Rocket UniVerse

UniVerse BASIC Commands Reference

Version 11.3.1

October 2016
UNV-1131-BASR-1

Notices

Edition

Publication date: October 2016
Book number: UNV-1131-BASR-1
Product version: Version 11.3.1

Copyright

© Rocket Software, Inc. or its affiliates 1985-2016. All Rights Reserved.

Trademarks

Rocket is a registered trademark of Rocket Software, Inc. For a list of Rocket registered trademarks go to: www.rocketsoftware.com/about/legal. All other products or services mentioned in this document may be covered by the trademarks, service marks, or product names of their respective owners.

Examples

This information might contain examples of data and reports. The examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

License agreement

This software and the associated documentation are proprietary and confidential to Rocket Software, Inc. or its affiliates, are furnished under license, and may be used and copied only in accordance with the terms of such license.

Note: This product may contain encryption technology. Many countries prohibit or restrict the use, import, or export of encryption technologies, and current use, import, and export regulations should be followed when exporting this product.

Corporate information

Rocket Software, Inc. develops enterprise infrastructure products in four key areas: storage, networks, and compliance; database servers and tools; business information and analytics; and application development, integration, and modernization.

Website: www.rocketsoftware.com

Rocket Global Headquarters
77 4th Avenue, Suite 100
Waltham, MA 02451-1468
USA

To contact Rocket Software by telephone for any reason, including obtaining pre-sales information and technical support, use one of the following telephone numbers.

Country	Toll-free telephone number
United States	1-855-577-4323
Australia	1-800-823-405
Belgium	0800-266-65
Canada	1-855-577-4323
China	800-720-1170
France	08-05-08-05-62
Germany	0800-180-0882
Italy	800-878-295
Japan	0800-170-5464
Netherlands	0-800-022-2961
New Zealand	0800-003210
South Africa	0-800-980-818
United Kingdom	0800-520-0439

Contacting Technical Support

The Rocket Customer Portal is the primary method of obtaining support. If you have current support and maintenance agreements with Rocket Software, you can access the Rocket Customer Portal and report a problem, download an update, or read answers to FAQs. To log in to the Rocket Customer Portal or to request a Rocket Customer Portal account, go to www.rocketsoftware.com/support.

In addition to using the Rocket Customer Portal to obtain support, you can use one of the telephone numbers that are listed above or send an email to support@rocketsoftware.com.

Contents

Notices ... 2

Corporate information ... 3

Chapter 1: Statements and functions ... 15

! statement ... 15
#INCLUDE statement ... 15
$* statement ... 16
$CHAIN statement ... 17
$COPYRIGHT statement ... 17
$DEFINE statement ... 18
$EJECT statement ... 20
$IFDEF statement ... 20
$IFNDEF statement ... 20
$INCLUDE statement ... 21
$INSERT statement ... 22
$MAP statement ... 23
$OPTIONS statement ... 23
$PAGE statement ... 30
$UNDEFINE statement ... 30
* statement ... 31
< > operator ... 31
@ function ... 32
[] operator ... 45
ABORT statement ... 47
ABS function ... 47
ABSS function ... 48
acceptConnection function ... 48
ACOS function ... 49
ACTIVATEKEY statement ... 50
addAuthenticationRule function ... 50
addCertificate function ... 52
addRequestParameter function ... 54
ADDS function ... 55
ALPHA function ... 56
amInitialize function ... 56
amReceiveMsg function ... 58
amReceiveRequest function ... 60
amSendMsg function ... 62
amSendRequest function ... 63
amSendResponse function ... 64
amTerminate function ... 65
analyzeCertificate function ... 66
ANDS function ... 66
ASCII function ... 67
ASIN function ... 68
ASSIGNED function ... 68
assignment statements ... 69
ATAN function ... 69
AuditLog() function ... 70
AUTHORIZATION statement ... 71
AUXMAP statement ... 72
BEGIN CASE statement ... 72

BEGIN TRANSACTION statement... 72
BITAND function... 73
BITNOT function... 74
BITOR function... 74
BITRESET function... 75
BITSET function... 75
BITTEST function... 76
BITXOR function... 76
BREAK statement... 77
BSCAN statement... 78
BYTE function... 79
BYTELEN function... 80
BYTETYPE function... 80
BYTEVAL function... 81
CALL statement... 81
CASE statements... 83
CATS function... 84
CENTURY.PIVOT function... 85
CHAIN statement... 86
CHANGE function... 86
CHAR function... 87
CHARS function... 88
CHECKSUM function... 88
CLEAR statement... 89
CLEARCOMMON... 89
CLEARDATA statement... 90
CLEARFILE statement... 90
CLEARPROMPTS statement... 92
CLEARSELECT statement... 92
CLOSE statement... 93
CLOSESEQ statement... 94
closeSocket function... 95
CloseXMLData function... 95
COL1 function... 96
COL2 function... 96
COMMAND.EDITOR... 97
COMMIT statement... 101
COMMON statement... 102
COMPARE function... 103
CONVERT function... 104
CONVERT statement... 105
COS function... 105
COSH function... 106
COUNT function... 106
COUNTS function... 107
CREATE statement... 108
createCertificate function... 109
createCertRequest function... 110
createRequest function... 112
createSecureRequest function... 113
createSecurityContext function... 115
CRT statement... 117
DATA statement... 118
DATE function... 119
DBTOXML function... 119
DCOUNT function... 120

DEACTIVATEKEY statement....121
DEBUG statement....121
DEFFUN statement....122
DEL statement....123
DELETE function....124
DELETE statements....126
DELETELIST statement....128
DESCRINFO function....128
DIGEST function....129
DIMENSION statement....130
DISABLEDEC statement....132
DISPLAY statement....133
DIV function....134
DIVS function....134
DOWNCASE function....135
DQUOTE function....135
DTX function....135
EBCDIC function....136
ECHO statement....137
ENABLEDEC statement....137
ENCODE function....138
ENCRYPT function....139
END statement....144
END CASE statement....145
END TRANSACTION statement....145
ENTER statement....145
EOF(ARG.) function....146
EQS function....146
EQUATE statement....147
EREPLACE function....148
ERRMSG statement....148
EXCHANGE function....149
EXECUTE statement....150
EXIT statement....152
EXP function....153
EXTRACT function....153
FADD function....155
FDIV function....155
FFIX function....156
FFLT function....156
FIELD function....156
FIELDS function....157
FIELDSTORE function....158
FILEINFO function....159
FILELOCK statement....163
FILEUNLOCK statement....165
FIND statement....166
FINDSTR statement....167
FIX function....167
FLUSH statement....168
FMT function....169
FMTDP function....172
FMTS function....172
FMTSDP function....173
FMUL function....174
FOLD function....174

FOLDDP function....175
FOOTING statement....175
FOR statement....178
FORMLIST statement....180
FSUB function....180
FUNCTION statement....181
generateKey function....182
GES function....183
GET statements....184
getCipherSuite function....186
getIpv....187
GETX statement....188
GET(ARG.) statement....188
getHTTPDefault function....189
GETLIST statement....189
GETLOCALE function....190
GETREM function....191
getSocketErrorMessage function....191
getSocketInformation function....192
getSocketMap function....192
getSocketOptions function....193
GOSUB statement....194
GOTO statement....195
GROUP function....196
GROUPSTORE statement....197
GTS function....198
HEADING statement....198
HMAC function....201
HUSH statement....202
ICHECK function....203
ICONV function....205
ICONVS function....206
IF statement....207
IFS function....208
ILPROMPT function....209
INCLUDE statement....210
INDEX function....211
INDEXS function....212
INDICES function....212
initSecureServerSocket function....215
initServerSocket function....215
INMAT function....216
INPUT statement....217
INPUTCLEAR statement....220
INPUTDISP statement....220
INPUTDP statement....221
INPUTERR statement....221
INPUTIF statement....221
INPUTNULL statement....222
INPUTTRAP statement....222
INS statement....222
INSERT function....224
INT function....226
ISNULL function....226
ISNULLS function....227
ITYPE function....227

KEYEDIT statement....229
KEYEXIT statement....233
KEYIN function....233
KEYTRAP statement....234
LEFT function....235
LEN function....235
LENDP function....236
LENS function....236
LENSDP function....237
LES function....237
LET statement....238
LN function....238
loadSecurityContext function....239
LOCALEINFO function....239
LOCATE statement (IDEAL and REALITY syntax)....240
LOCATE statement (INFORMATION syntax)....242
LOCATE statement (PICK syntax)....244
LOCK statement....247
LOOP statement....248
LOWER function....249
LTS function....250
MAT statement....251
MATBUILD statement....252
MATCH operator....253
MATCHFIELD function....254
MATPARSE statement....255
MATREAD statements....256
MATREADL statement....259
MATREADU statement....259
MATWRITE statements....259
MATWRITEU statement....262
MAXIMUM function....262
MINIMUM function....262
MOD function....263
MODS function....264
MQCLOSE function....264
MQCONN function....265
MQDISC function....266
MULS function....267
NAP statement....267
NEG function....268
NEGS function....268
NES function....268
NEXT statement....269
NOBUF statement....269
NOT function....270
NOTS function....270
NULL statement....271
NUM function....271
NUMS function....272
OCONV function....272
OCONVS function....274
ON statement....274
OPEN statement....276
OPENCHECK statement....279
OPENDEV statement....280

OPENPATH statement... 281
OPENSEQ statement... 283
openSecureSocket function... 286
openSocket function... 287
OpenXMLData function... 288
ORS function... 289
PAGE statement... 289
PERFORM statement... 290
PRECISION statement... 291
PrepareXML function... 292
PRINT statement... 293
PRINTER statement... 294
PRINTERR statement... 295
PROCREAD statement... 296
PROCWRITE statement... 296
PROGRAM statement... 297
PROMPT statement... 297
protocolLogging function... 298
PWR function... 298
PyCall function... 299
PyCallFunction function... 299
PyCallMethod function... 299
PyGetAttr function... 300
PyImport function... 300
PySetAttr function... 301
QUOTE function... 301
RAISE function... 301
RANDOMIZE statement... 302
READ statements... 303
READBLK statement... 307
READL statement... 308
READLIST statement... 308
READNEXT statement... 309
READSEQ statement... 310
readSocket function... 311
READT statement... 312
READU statement... 314
READV statement... 314
READVL statement... 314
READVU statement... 315
ReadXMLData function... 315
REAL function... 316
RECORDLOCK statements... 316
RECORDLOCKED function... 318
RELEASE statement... 319
ReleaseXML function... 320
REM function... 320
REM statement... 321
REMOVE function... 322
REMOVE statement... 323
REPEAT statement... 324
REPLACE function... 325
RETURN statement... 327
RETURN (value) statement... 328
REUSE function... 328
REVREMOVE statement... 329

REWIND statement..... 330
RIGHT function..... 331
RND function..... 331
ROLLBACK statement..... 332
RPC.CALL function..... 333
RPC.CONNECT function..... 334
RPC.DISCONNECT function..... 335
saveSecurityContext function..... 335
SADD function..... 336
SCMP function..... 337
SDIV function..... 337
SEEK statement..... 338
SEEK(ARG.) statement..... 339
SELECT statements..... 340
SELECTE statement..... 342
SELECTINDEX statement..... 342
SELECTINFO function..... 343
SEND statement..... 344
SENTENCE function..... 344
SEQ function..... 345
SEQS function..... 345
setAuthenticationDepth function..... 346
setCipherSuite function..... 347
setClientAuthentication function..... 349
setIpv..... 350
setPrivateKey function..... 351
setRandomSeed function..... 352
SET TRANSACTION ISOLATION LEVEL statement..... 353
setHTTPDefault function..... 354
setRequestHeader function..... 355
SETLOCALE function..... 356
SETREM statement..... 358
setSocketMap function..... 359
setSocketOptions function..... 359
showSecurityContext function..... 360
SIGNATURE function..... 361
SIN function..... 362
SINH function..... 363
SLEEP statement..... 363
SMUL function..... 364
SOAPCreateRequest function..... 364
SOAPCreateSecureRequest function..... 365
SOAPGetDefault function..... 367
SOAPGetFault function..... 367
SOAPGetResponseHeader function..... 368
SOAPSetRequestBody function..... 369
SOAPSetRequestContent function..... 369
SOAPSetRequestHeader function..... 370
SOAPRequestWrite function..... 371
SOAPSetDefault function..... 371
SOAPSetParameters function..... 372
SOAPSubmitRequest function..... 373
SOUNDEX function..... 375
SPACE function..... 375
SPACES function..... 376
SPLICE function..... 376

SQRT function....376
SQUOTE function....377
SSELECT statement....377
SSUB function....379
STATUS function....380
STATUS statement....384
STOP statement....387
STORAGE statement....388
STR function....388
STRS function....389
submitRequest function....389
SUBR function....390
SUBROUTINE statement....392
SUBS function....392
SUBSTRINGS function....393
SUM function....393
SUMMATION function....394
SWAP statement....395
SYSTEM function....395
TABSTOP statement....399
TAN function....399
TANH function....400
TERMINFO function....400
TIME function....415
TIMEDATE function....415
TIMEOUT statement....416
TPARM function....417
TPRINT statement....419
TRANS function....420
transaction statements....421
TRANSACTION ABORT statement....421
TRANSACTION COMMIT statement....423
TRANSACTION START statement....423
TRIM function....423
TRIMB function....424
TRIMBS function....425
TRIMF function....425
TRIMFS function....426
TRIMS function....426
TTYCTL statement....426
TTYGET statement....427
TTYSET statement....431
UDOArrayAppendItem....433
UDOArrayDeleteItem....433
UDOArrayGetItem....434
UDOArrayGetNextItem....434
UDOArrayGetSize....435
UDOArrayInsertItem....435
UDOArraySetItem....435
UDOClone....436
UDOCreate....436
UDODeleteProperty....437
UDOFree....437
UDOGetLastError....437
UDOGetNextProperty....438
UDOGetOption....438

UDOGetProperty ... 439
UDOGetPropertyNames ... 439
UDOGetType ... 439
UDOIsTypeOf ... 440
UDORead ... 440
UDOSetOption ... 440
UDOSetProperty ... 441
UDOWrite ... 441
UNASSIGNED function ... 442
UNICHAR function ... 442
UNICHARS function ... 442
UNISEQ function ... 443
UNISEQS function ... 443
UNLOCK statement ... 444
UPCASE function ... 444
UPRINT statement ... 445
USERINFO function ... 445
WEOF statement ... 447
WEOFSEQ statement ... 448
WRITE statements ... 449
WRITEBLK statement ... 453
WRITELIST statement ... 454
WRITESEQ statement ... 454
WRITESEQF statement ... 456
writeSocket function ... 457
WRITET statement ... 458
WRITEU statement ... 459
WRITEV statement ... 459
WRITEVU statement ... 459
XDOMAddChild function ... 459
XDOMAppend function ... 461
XDOMClone function ... 462
XDOMClose function ... 462
XDOMCreateNode function ... 463
XDOMCreateRoot function ... 464
XDOMEvaluate function ... 465
XDOMGetAttribute function ... 466
XDOMGetChildNodes function ... 466
XDOMGetElementById function ... 467
XDOMGetElementsByName function ... 468
XDOMGetElementsByTag function ... 469
XMLGetError function ... 470
XDOMGetNodeName function ... 470
XDOMGetNodeType function ... 471
XDOMGetNodeValue function ... 471
XDOMGetOwnerDocument function ... 472
XDOMGetUserData function ... 472
XDOMItem function ... 473
XDOMLength function ... 474
XDOMLocate function ... 474
XDOMLocateNode function ... 475
XDOMOpen function ... 477
XDOMQuery function ... 479
XDOMRemove function ... 479
XDOMReplace function ... 480
XDOMSetNodeValue function ... 481

XDOMSetUserData function.... 481
XDOMTransform function.... 482
XDOMValidate function.... 483
XDOMWrite function.... 483
XLATE function.... 484
XMAPAppendRec.... 485
XMAPClose function.... 486
XMAPCreate Function.... 486
XMAPOpen function.... 487
XMAPReadNext function.... 488
XMAPToXMLDoc function.... 489
XMLError function.... 489
XMLExecute function.... 490
XMLTODB function.... 492
XTD function.... 493

Appendix A: Quick reference.... 494
Compiler directives.... 494
Declarations.... 495
Assignments.... 495
Program flow control.... 496
File I/O.... 497
Sequential file I/O.... 498
Printer and terminal I/O.... 499
Tape I/O.... 500
Select lists.... 500
String handling.... 501
Data conversion and formatting.... 503
NLS.... 504
Mathematical functions.... 505
Relational functions.... 506
System.... 507
Remote procedure calls.... 508
Miscellaneous.... 508

Appendix B: ASCII and hex equivalents.... 509

Appendix C: Correlative and conversion codes.... 513
A code: algebraic functions.... 515
BB and BX codes: bit conversion.... 517
C code: concatenation.... 518
D code: date conversion.... 519
DI code: international date conversion.... 523
ECS code: extended character set conversion.... 524
F code: mathematical functions.... 524
G code: group extraction.... 526
L code: length function.... 526
MC Codes: masked character conversion.... 526
MD code: masked decimal conversion.... 528
MM code: monetary conversion.... 530
ML and MR codes: formatting numbers.... 531
MP code: packed decimal conversion.... 532
MT code: time conversion.... 533
MX, MO, MB, and MU0C codes: radix conversion.... 534
MY code: ASCII conversion.... 534
NL code: Arabic numeral conversion.... 535
NLSmapname code: NLS map conversion.... 535
NR code: roman numeral conversion.... 536

P code: pattern matching....536
Q code: exponential notation....537
R code: range function....538
S (soundex) code....538
S (substitution) code....538
T code: text extraction....539
Tfile code: file translation....539
TI code: international time conversion....540

Appendix D: BASIC reserved words....541

Appendix E: @Variables....552

Appendix F: BASIC subroutines....556
! ASYNC subroutine....557
!EDIT.INPUT subroutine....558
!ERRNO subroutine....563
!FCMP subroutine....563
!GET.KEY subroutine....563
!GET.PARTNUM subroutine....564
!GET.PATHNAME subroutine....566
!GETPU subroutine....566
!GET.USER.COUNTS subroutine....569
!GET.USERS subroutine....569
!INLINE.PROMPTS subroutine....570
!INTS subroutine....571
!MAKE.PATHNAME subroutine....572
!MATCHES subroutine....573
!MESSAGE subroutine....573
!PACK.FNKEYS subroutine....574
!REPORT.ERROR subroutine....577
!SET.PTR subroutine....578
!SETPU subroutine....579
!TIMDAT subroutine....581
!USER.TYPE subroutine....582
!VOC.PATHNAME subroutine....583

Appendix G: Socket function error return codes....584

Chapter 1: Statements and functions

This chapter describes the UniVerse BASIC statements and functions.

! statement

Use the ! statement to insert a comment in a UniVerse BASIC program. Comments explain or document various parts of a program. They are part of the source code only and are nonexecutable. They do not affect the size of the object code.

A comment must be a separate BASIC statement and can appear anywhere in a program. A comment must begin with one of the following comment designators:

- REM
- *
- !
- $*

Any text that appears between a comment designator and the end of a physical line is treated as part of the comment, not as part of the executable program. If a comment does not fit on one physical line, you can continue it on the next physical line only by starting the new line with a comment designator. If a comment appears at the end of a physical line containing an executable statement, you must put a semicolon (;) before the comment designator.

Syntax

```
! [comment.text]
```

Example

The PRINT statement at the end of the third line is not executed because it follows the exclamation point on the same line and is treated as part of the comment. Lines 4, 5, and 6 show how to include a comment in the same sequence of executable statements.

```
001: vi PRINT "HI THERE"; ! Anything after the ! is a comment.
 002: ! This line is also a comment and does not print.
 003: IF 5<6 THEN PRINT "YES"; ! A comment; PRINT "PRINT ME"
 004: IF 5<6 THEN
 005: PRINT "YES"; ! A comment
 006: PRINT "PRINT ME"
 007: END
```

This is the program output:

```
HI THERE
 YES
 YES
 PRINT ME
```

#INCLUDE statement

Use the #INCLUDE statement to direct the compiler to insert the source code in the record program and compile it with the main program. The #INCLUDE statement differs from the $CHAIN statement in

that the compiler returns to the main program and continues compiling with the statement following the #INCLUDE statement.

When *program* is specified without *filename*, *program* must be a record in the same file as the program containing the #INCLUDE statement.

If *program* is a record in a different file, the file name must be specified in the #INCLUDE statement, followed by the name of the program. The file name must specify a type 1 or type 19 file defined in the VOC file.

You can nest #INCLUDE statements.

The #INCLUDE statement is a synonym for the $INCLUDE and INCLUDE statements.

Syntax

#INCLUDE [*filename*] *program*

#INCLUDE *program* FROM *filename*

Example

```
PRINT "START"
 #INCLUDE END
 PRINT "FINISH"
```

When this program is compiled, the #INCLUDE statement inserts code from the program END (see the example on the END statement, on page 144). This is the program output:

```
START
 THESE TWO LINES WILL PRINT ONLY
 WHEN THE VALUE OF 'A'  IS 'YES'.

 THIS IS THE END OF THE PROGRAM
```

$* statement

Use the $* statement to insert a comment in UniVerse BASIC object code. Comments explain or document various parts of a program. They are nonexecutable.

A comment must be a separate UniVerse BASIC statement and can appear anywhere in a program.

Any text appearing between the $* and the end of a physical line is treated as part of the comment, not as part of the executable program. If a comment does not fit on one physical line, you can continue it on the next physical line only by starting the new line with another $*. If a comment appears at the end of a physical line containing an executable statement, you must put a semicolon (;) before the $*.

Syntax

$* [*comment.text*]

Example

The PRINT statement at the end of the third line is not executed because it follows the exclamation point on the same line and is treated as part of the comment. Lines 4, 5, and 6 show how to include a comment in the same sequence of executable statements.

```
001: PRINT "HI THERE"; $* Anything after the $* is a comment.
 002: $* This line is also a comment and does not print.
```

```
003: IF 5<6 THEN PRINT "YES"; $* A comment; PRINT "PRINT ME"
004: IF 5<6 THEN
005: PRINT "YES"; $* A comment
006: PRINT "PRINT ME"
007: END
```

This is the program output:

```
HI THERE
 YES
 YES
 PRINT ME
```

$CHAIN statement

Use the $CHAIN statement to direct the compiler to read source code from program and compile it as if it were part of the current program. The $CHAIN statement differs from the $INCLUDE statement, #INCLUDE statement, and INCLUDE statement in that the compiler does not return to the main program. Any statements appearing after the $CHAIN statement are not compiled or executed.

When the program name is specified without a file name, the source code to insert must be in the same file as the current program.

If the source code to insert is in a different file, the $CHAIN statement must specify the name of the remote file followed by the program name. *filename* must specify a type 1 or type 19 file defined in the VOC file.

When statements in *program* generate error messages, the messages name the program containing the $CHAIN statement.

Syntax

$CHAIN [*filename*] *program*

Example

```
PRINT "START"
 $CHAIN END
 PRINT "FINISH"
```

When this program is compiled, the $CHAIN statement inserts code from the program END (see the example in END statement, on page 144). This is the program output:

```
START
THESE TWO LINES WILL PRINT ONLY
WHEN THE VALUE OF 'A'    IS 'YES'.

THIS IS THE END OF THE PROGRAM
```

$COPYRIGHT statement

Use the $COPYRIGHT statement to specify copyright information in UniVerse BASIC object code. *copyright.notice* is inserted in the copyright field at the end of the object code.

copyright.notice must be enclosed in single or double quotation marks.

The copyright field in the object code is set to the empty string at the beginning of compilation. It remains empty until the program encounters a $COPYRIGHT statement.

If more than one $COPYRIGHT statement is included in the program, only the information included in the last one encountered is inserted in the object code.

This statement is included for compatibility with existing software.

Syntax

```
$COPYRIGHT "copyright.notice"
```

$DEFINE statement

Use the $DEFINE statement to define identifiers that control program compilation. $DEFINE has two functions:

- Defining an identifier
- Supplying replacement text for an identifier

Syntax

```
$DEFINE identifier [replacement.text]
```

Parameters

Parameter	Description
identifier	The symbol to be defined. It can be any valid identifier.
replacement.text	A string of characters that the compiler uses to replace *identifier* everywhere it appears in the program containing the $DEFINE statement.

When used as a replacement text supplier, $DEFINE adds the specified identifier and its associated *replacement.text* to the symbol table. Each time *identifier* is found in the program following the $DEFINE statement in which its value was set, it is replaced by *replacement.text*. If *replacement.text* is not specified, *identifier* is defined and has a null value.

Separate *replacement.text* from *identifier* with one or more blanks. Every character typed after this blank is added to *replacement.text* up to, but not including, the Return character that terminates the *replacement.text*.

Note: Do not use comments when supplying *replacement.text* because any comments after *replacement.text* are included as part of the replacement text. Any comments added to *replacement.text* can cause unexpected program behavior.

UniVerse does not supported nested $DEFINE/$UNDEFINE statements.

The $UNDEFINE statement removes the definition of an identifier.

Conditional compilation

You can use $DEFINE with the $IFDEF statement or $IFNDEF statement to define an identifier that controls conditional compilation. The syntax is as follows:

```
$DEFINE identifier [replacement.text]
  .
  .
```

```
 .
{ $IFDEF | $IFNDEF } identifier
[statements]
$ELSE
[statements]
$ENDIF
```

The $IFDEF or $IFNDEF statement that begins the conditional compilation block tests *identifier* to determine whether it is defined by a $DEFINE statement. If you use $IFDEF and *identifier* is defined, the statements between the $IFDEF and the $ELSE statements are compiled. If *identifier* is not defined, the statements between the $ELSE and $ENDIF statements are compiled.

If you use $IFNDEF, on the other hand, and *identifier* is defined, the statements between $ELSE and $ENDIF are compiled. If *identifier* is not defined, the statements between the $IFDEF and $ELSE statements are compiled.

Conditional compiler directives

Conditional compiler directives allow the inclusion of code and features available in later releases of UniVerse to be included in programs used in earlier releases. The newer, unavailable features are ignored by the compiler on older UniVerse releases. This helps developers avoid maintaining multiple code streams for the various releases of UniVerse.

The following compiler definitions are available in UniVerse BASIC. Note that the directives with "UNIVERSE" in the name use a double underscore.

- U2__UNIVERSE
- U2__UNIVERSEv11
- U2__UNIVERSEv11.2
- U2__UNIVERSEv11.3
- U2_LOCALCALL

For example, specifying $IFDEF U2__UNIVERSEv11.2, allows the use of 11.2 functionality within the $IFDEF statement. The U2_LOCALCALL identifier can be used for local subroutines and variables without being specific to 11.2. Using $IFDEF with the UniVerse supplied identifiers allows for compiling a program on an earlier release where the code contained in the $IFDEF clause will be ignored.

Note: The $UNDEFINE statement cannot be used to remove the UniVerse supplied identifiers.

Example

In this example the identifier NAME.SUFFIX is defined to have a value of PROGRAM.NAME[5]. When the compiler processes the next line, it finds the symbol NAME.SUFFIX, substitutes PROGRAM.NAME[5] in its place and continues processing with the first character of the replacement text.

```
$DEFINE NAME.SUFFIX PROGRAM.NAME[5]
 IF NAME.SUFFIX = '.B' THEN
    .
    .
    .
 END
    .
    .
    .
```

$EJECT statement

Use the $EJECT statement to begin a new page in the listing record.

Syntax

$EJECT

This statement is a synonym for the $PAGE statement.

$IFDEF statement

Use the $IFDEF statement to test for the definition of a compile-time symbol. $IFDEF tests to see if *identifier* is currently defined (that is, has appeared in a $DEFINE statement and has not been undefined).

If *identifier* is currently defined and the $ELSE clause is omitted, the statements between the $IFDEF and $ENDIF statements are compiled. If the $ELSE clause is included, only the statements between $IFDEF and $ELSE are compiled.

If *identifier* is not defined and the $ELSE clause is omitted, all the lines between the $IFDEF and $ENDIF statements are ignored. If the $ELSE clause is included, only the statements between $ELSE and $ENDIF are compiled.

Both the IFDEF statement and $IFNDEF statement can be nested up to 10 deep.

Syntax

```
$IFDEF identifier
[statements]
[[$ELSE]
[statements]]
$ENDIF
```

Example

The following example determines if the identifier “modified” is defined:

```
$DEFINE modified 0
 $IFDEF modified
 PRINT "modified is defined."
 $ELSE
 PRINT "modified is not defined."
 $ENDIF
```

$IFNDEF statement

Use the $IFNDEF statement to test for the definition of a compile-time symbol. The $IFNDEF statement complements the $IFDEF statement.

If *identifier* is currently not defined and the $ELSE clause is omitted, the statements between the $IFNDEF and $ENDIF statements are compiled. If the $ELSE clause is included, only the statements between $IFNDEF and $ELSE are compiled.

If *identifier* is defined and the $ELSE clause is omitted, all the lines between the $IFNDEF and $ENDIF statements are ignored. If the $ELSE clause is included, only the statements between $ELSE and $ENDIF are compiled.

$IFDEF and $IFNDEF statements can be nested up to 10 deep.

Syntax

```
$IFNDEF identifier
[statements]
[[$ELSE]
[statements]]
$ENDIF
```

Example

The following example determines if the identifier “modified” is not defined:

```
$DEFINE modified 0
 $IFNDEF modified
 PRINT "modified is not defined."
 $ELSE
 PRINT "modified is defined."
 $ENDIF
```

$INCLUDE statement

Use the $INCLUDE statement to direct the compiler to insert the source code in the record program and compile it with the main program. The $INCLUDE statement differs from the $CHAIN statement in that the compiler returns to the main program and continues compiling with the statement following the $INCLUDE statement.

When *program* is specified without *filename*, *program* must be a record in the same file as the program currently containing the $INCLUDE statement.

If *program* is a record in a different file, the file name must be specified in the $INCLUDE statement, followed by the name of the program. The file name must specify a type 1 or type 19 file defined in the VOC file.

You can nest $INCLUDE statements.

The $INCLUDE statement is a synonym for the #INCLUDE and INCLUDE statements.

Syntax

```
$INCLUDE [filename] program
```

```
$INCLUDE program FROM filename
```

Example

```
PRINT "START"
 $INCLUDE END
 PRINT "FINISH"
```

When this program is compiled, the $INCLUDE statement inserts code from the program END (see the example in END statement, on page 144). This is the program output:

```
START
THESE TWO LINES WILL PRINT ONLY
WHEN THE VALUE OF 'A'    IS 'YES'.

THIS IS THE END OF THE PROGRAM
```

$INSERT statement

Use the $INSERT statement to direct the compiler to insert the source code contained in the file specified by *primos.pathname* and compile it with the main program. The difference between the $INSERT statement and $INCLUDE statement (and its synonyms #INCLUDE and INCLUDE) is that $INSERT takes a PRIMOS path name as an argument, whereas $INCLUDE takes a UniVerse file name and record ID. The PRIMOS path is converted to a path; any leading *> is ignored.

$INSERT is included for compatibility with Prime INFORMATION programs; the $INCLUDE statement is recommended for general use.

Syntax

$INSERT *primos.pathname*

If *primos.pathname* is the name of the program only, it is interpreted as a relative path. In this case, the program must be a file in the same directory as the program containing the $INSERT statement.

You can nest $INSERT statements.

primos.pathname is converted to a valid path using the following conversion rules:

Conversion rules
/ is converted to ?\
? is converted to ??
ASCII CHAR 0 (NUL) is converted to ?0
. (period) is converted to ?.

If you specify a full path name, the > between directory names changes to a / to yield:

[pathname/] program

$INSERT uses the transformed argument directly as a path of the file containing the source to be inserted. It does not use the file definition in the VOC file.

Example

```
PRINT "START"
 $INSERT END
 PRINT "FINISH"
```

When this program is compiled, the $INSERT statement inserts code from the program END (see the example in END statement, on page 144). This is the program output:

```
START
 THESE TWO LINES WILL PRINT ONLY
 WHEN THE VALUE OF 'A'  IS 'YES'.
```

```
THIS IS THE END OF THE PROGRAM
FINISH
```

$MAP statement

In NLS mode, use the $MAP statement to direct the compiler to specify the map for the source code. Use the $MAP statement if you use embedded literal strings that contain non-ASCII characters.

Syntax

$MAP *mapname*

mapname must be the name of a map that has been built and installed.

You can use only one $MAP statement during compilation.

> **Note:** You can execute programs that contain only ASCII characters whether NLS mode is on or off. You cannot execute programs that contain non-ASCII characters that were compiled in NLS mode if NLS mode is switched off.

For more information, see the *NLS Guide*.

Example

The following example assigns a string containing the three characters alpha, beta, and gamma to the variable GREEKABG:

```
$MAP MNEMONICS
 .
.
.GREEKABG = "<A*><B*><G*>"
```

$OPTIONS statement

Use the $OPTIONS statement to set compile-time emulation of any UniVerse flavor. This does not allow object code compiled in one flavor to execute in another flavor. You can select individual options in a program to override the default setting.

> **Note:** You must specify $OPTIONS for each internal subroutine.

Syntax

$OPTIONS [*flavor*] [*options*]

Flavor keywords

Use the following keywords to specify *flavor*:

Keyword	Flavor
PICK	Generic Pick emulation
INFORMATION	Prime INFORMATION emulation

Keyword	Flavor
REALITY	REALITY emulation
IN2	Intertechnique emulation
DEFAULT	IDEAL UniVerse
PIOPEN	PI/open emulation

For instance, the following statement instructs the compiler to treat all UniVerse BASIC syntax as if it were running in a PICK flavor account:

```
$OPTIONS PICK
```

Another way to select compile-time emulation is to specify one of the following keywords in field 6 of the VOC entry for the BASIC command:

```
INFORMATION.FORMAT
 PICK.FORMAT
 REALITY.FORMAT
 IN2.FORMAT
 PIOPEN.FORMAT
```

By default the VOC entry for the BASIC command corresponds with the account flavor specified when your UniVerse account was set up.

Options keywords

options are specified by the keywords listed in following table. To turn off an option, prefix it with a minus sign (-).

Option name	Option letter	Description
CASE	*none*	Differentiates between uppercase and lowercase identifiers and keywords.
COMP.PRECISION	*none*	Rounds the number at the current precision value in any comparison.
COUNT.OVLP	O	For the INDEX function and the COUNT function, the count overlaps.

Option name	Option letter	Description

DIM,IN.SUM		By default, arrays passed as arguments in a subroutine call cannot be redimensioned in the subroutine. An attempt to redimension the array is simply ignored. If you set the DIM.IN.SUB option through the $OPTIONS statement, you can redimension the array in a subroutine. See the following example:
		
		`>AE BP CALLER SUBTEST`
		
		`CALLER`
		`0001 DIM A(10)`
		`0002 CALL SUBTEST(MAT A)`
		`0003 CRT A(100)`
		`0004 END`
		
		`SUBTEST`
		`0001 SUBROUTINE SUBTEST(MAT A)`
		`0002 $OPTIONS DIM.IN.SUB`
		`0003 DIM A(100)`
		`0004 A(100) = 100`
		`0005 RETURN`
		`0006 END`
		
		`>RUN BP CALLER`
		`100`

END.WARN	R	Prints a warning message if there is no final END statement.

EXEC.EQ.PERF	P	Compiles the EXECUTE statement as the PERFORM statement.
		
		Note: If the syntax of the EXECUTE statement is changed so it is no longer compatible with the PERFORM statement, UniVerse ignores EXEC.EQ.PERF. For example, UniVerse ignores EXEC.EQ.PERF in the following program:
		
		`0001 "$OPTIONS EXEC.EQ.PERF`
		`0002 EXECUTE 'DATE' CAPTURING RESULTS`
		`0003 END`

EXTRA.DELIM	W	For the INSERT function and the REPLACE function, the compiler handles fields, values, and subvalues that contain the empty string differently from the way they are handled in the IDEAL flavor. In particular, if you specify a negative one (-1) parameter, INFORMATION and IN2 flavors add another delimiter, except when starting with an empty string.
FOR.INCR.BEF	F	Increments the index for FOR…NEXT loop before instead of after the bound checking.
FORMAT.OCONV	*none*	Lets output conversion codes be used as format masks (see the FMT function, on page 169).

Option name	Option letter	Description
FSELECT	*none*	Makes the SELECT statements return the total number of records selected to the @SELECTED variable. Using this option can result in slower performance for the SELECT statement.
HEADER.BRK	*none*	Specifies the PIOPEN flavor for the I and P options to the HEADING statement and FOOTING statement. This is the default for the PIOPEN flavor.
HEADER.DATE	D	Displays times and dates in headings or footings in fixed format (that is, they do not change from page to page). Dates are displayed in 'D2-' format instead of 'D' format. Allows page number field specification by multiple invocations of 'P' in a single set of quotation marks.
HEADER.EJECT	H	HEADING statement causes initial page eject.
IN2.SUBSTR	T	Uses IN2 definitions for UniVerse BASIC substring handling (*string*[*n*,*m*]). If a single parameter is specified, a length of 1 is assumed. The size of the string expands or contracts according to the length of the replacement string.
INFO.ABORT	J	ABORT statement syntax follows Prime INFORMATION instead of PICK.
INFO.CONVERT	*none*	Specifies that the `FMT`, `ICONV`, and `OCONV` functions perform PI/open style conversions.
INFO.ENTER	*none*	Specifies the PIOPEN flavor of the ENTER statement.
INFO.INCLUDE	none	Processes any PRIMOS paths specified with the $INSERT statement.
INFO.LOCATE	L	LOCATE syntax follows Prime INFORMATION instead of REALITY. The Pick format of the LOCATE statement is always supported in all flavors.
INFO.MARKS	*none*	Specifies that the LOWER, RAISE, and REMOVE functions use a smaller range of delimiters for PI/open compatibility.
INFO.MOD	*none*	Specifies the PIOPEN flavor for the MOD function. This is the default for the PIOPEN flavor.
INPUTAT	none	Specifies the PIOPEN flavor for the INPUT @ statement. This is the default for the PIOPEN flavor.
INPUT.ELSE	Y	Accepts an optional THEN…ELSE clause on INPUT statement.
INT.PRECISION	none	Rounds the integer at the current precision value in an INT function.
LOCATE.R83	none	A LOCATE statement returns an “AR” or “DR” sequence value compatible with Pick, Prime INFORMATION, and PI/open systems.
NO.CASE	*none*	Does not differentiate between uppercase and lowercase in identifiers or keywords. This is the default for the PIOPEN flavor.
NO.RESELECT	U	For the SELECT statements and SSELECT statement, active select list 0 remains active; another selection or sort is not performed. The next READNEXT statement uses select list 0.

Option name	Option letter	Description
NO.RETURN.WARN	none	Suppresses display of warning messages from ambiguous RETURN statements.
ONGO.RANGE	G	If the value used in an ON…GOTO or ON…GOSUB is out of range, executes the next statement rather than the first or last branch.
PCLOSE.ALL	Z	The PRINTER CLOSE statement closes all print channels.
PERF.EQ.EXEC	C	The PERFORM statement compiles as the EXECUTE statement.
PIOPEN.EXECUTE	none	EXECUTE behaves similarly to the way it does on PI/open systems.
PIOPEN.INCLUDE	none	Processes any PRIMOS paths specified with the $INSERT statement and the $INCLUDE statement.
PIOPEN.MATREAD	none	Sets the elements of the matrix to empty strings when the record ID is not found. MATREAD, MATREADL, and MATREADU will behave as they do on PI/open systems.
PIOPEN.SELIDX	*none*	In the SELECTINDEX statement, removes multiple occurrences of the same record ID in an index with a multivalued field.
RADIANS	*none*	Calculates trigonometric operations using radians instead of degrees.
RAW.OUTPUT	*none*	Suppresses automatic mapping of system delimiters on output. When an application handles terminal control directly, RAW.OUTPUT turns off this automatic mapping.
READ.RETAIN	Q	If READ statements, READU statement, READV statement, READVL statement, or a READVU statement fail, the resulting variable retains its value. The variable is not set to an empty string.
REAL.SUBSTR	K	Uses REALITY flavor definitions for substring handling (*string*[*n*,*m*]). If *m* or *n* is less than 0, the starting position for substring extraction is defined as the right side (the end) of the string.
RNEXT.EXPL	X	A READNEXT statement returns an exploded select list.
SEQ.255	N	SEQ(" ") = 255 (instead of 0).
STATIC.DIM	M	Creates arrays at compile time, not at run time. The arrays are not redimensioned, and they do not have a zero element.
STOP.MSG	E	Causes a STOP statement and an ABORT statement to use the ERRMSG file to produce error messages instead of using the specified text.
STRING.MATH	*none*	Causes UniVerse BASIC to automatically use the SADD, SSUB, SDIV, and SMUL functions rather than +, -, /, and *. This option also applies to the INT, ABS, NEG, and MOD functions.
SUPP.DATA.ECHO	I	Causes input statements to suppress echo from data.

Option name	Option letter	Description
TIME.MILLISECOND	*none*	Causes the SYSTEM (12) function to return the current system time in milliseconds, and the TIME function to return the current system time in seconds.
ULT.FORMAT	*none*	Format operations are compatible with Ult/ix. For example, FMT("","MR2") returns an empty string, not 0.00.
USE.ERRMSG	B	The PRINTERR statement prints error messages from ERRMSG.
VAR.SELECT	S	SELECT TO *variable* creates a local select variable instead of using numbered select lists, and the READLIST statement reads a saved select list instead of an active numbered select list.
VEC.MATH	V	Uses vector arithmetic instructions for operating on multivalued data. For performance reasons the IDEAL flavor uses singlevalued arithmetic.
WIDE.IF	*none*	Testing numeric values for true or false uses the wide zero test. In Release 6 of UniVerse, the WIDE.IF option is OFF by default. In Release 7, WIDE.IF is ON by default.

You can also set individual options by using special versions of some statements to override the current setting. These are listed as follows:

Statement	Equal to...
ABORTE	The ABORT statement with $OPTIONS STOP.MSG
ABORTM	ABORT with $OPTIONS -STOP.MSG
HEADINGE	The HEADING statement with $OPTIONS HEADER.EJECT
HEADINGN	HEADING with $OPTIONS -HEADER.EJECT
SELECTV	The SELECT statements with $OPTIONS VAR.SELECT
SELECTN	SELECT with $OPTIONS -VAR.SELECT
STOPE	The STOP statement with $OPTIONS STOP.MSG
STOPM	STOP with $OPTIONS -STOP.MSG

The default settings for each flavor are listed in the following table:

	IDEAL	PICK	INFO	REALITY	IN2	PIOPEN
CASE			X			
COMP.PRECISION						
COUNT.OVLP		X		X	X	
END.WARN			X	X		X
EXEC.EQ.PERF			X			X
EXTRA.DELIM			X		X	X
FOR.INC.REF	X	X		X	X	
FORMAT.OCONV				X		
FSELECT						
HEADER.BRK						X
HEADER.DATE			X			X

	IDEAL	PICK	INFO	REALITY	IN2	PIOPEN
HEADER.EJECT			X			X
IN2.SUBSTR			X		X	X
INFO.ABORT						X
INFO.CONVERT						
INFO.ENTER						X
INFO.LOCATE			X			X
INFO.MARKS						X
INFO.MOD						X
INPUTAT						X
INPUT.ELSE		X	X			
INT.PRECISION						
LOCATE.R83						
NO.CASE						X
NO.RESELECT		X	X		X	X
NO.SMA.COMMON						
ONGO.RANGE		X			X	
PCLOSE.ALL		X		X	X	
PERF.EO.EXEC				X		X
PIOPEN.EXECUTE						
PIOPEN.INCLUDE						X
PIOPEN.MATREAD						
PIOPEN.SELIDX						X
RADIANS					X	
RAW.OUTPUT						
READ.RETAIN		X		X	X	
REAL.SUBSTR			X	X		X
RNEXT.EXPL			X			
SEQ.255		X		X	X	
STATIC.DIM		X		X	X	
STOP.MSG		X		X	X	
SUPP.DATA.ECHO		X		X	X	
ULT.FORMAT						
USE.ERRMSG				X		
VAR.SELECT		X		X	X	
VEC.MATH			X			X
WIDE.IF	X	X	X	X	X	

Example

```
>ED BP OPT
4 lines long.
 ----: P
 0001: $OPTIONS INFORMATION
 0002: A='12'
 0003: B='14'
```

```
 0004: PRINT A,B
 Bottom at line 4
 ----: Q
>BASIC BP OPT
Compiling: Source = 'BP/OPT', Object = 'BP.O/OPT'

 @EOF           WARNING: Final 'END' statement not found.

 Compilation Complete.
 >ED BP OPT
4 lines long.
 ----: P
 0001: $OPTIONS PICK
 0002: A='12'
 0003: B='14'
 0004: PRINT A,B
 Bottom at line 4
 ----: Q
>BASIC BP OPT
Compiling: Source = 'BP/OPT', Object = 'BP.O/OPT'
 Compilation Complete.
```

$PAGE statement

The $PAGE statement is a synonym for the $EJECT statement, on page 20.

$UNDEFINE statement

Use the $UNDEFINE statement to remove the definition of identifiers set with the $DEFINE statement. The $UNDEFINE statement removes the definition of *identifier* from the symbol table if it appeared in a previous $DEFINE statement. If the identifier was not previously defined, $UNDEFINE has no effect.

Syntax

$UNDEFINE *identifier*

identifier is the identifier whose definition is to be deleted from the symbol table.

You can use $UNDEFINE with the $IFDEF statement or $IFNDEF statement to undefine an identifier that controls conditional compilation. The syntax is as follows:

```
$UNDEFINE identifier
  .
  .
  .
{ $IFDEF | $IFNDEF }identifier
[statements]
$ELSE
[statements]
$ENDIF
```

The $IFDEF statement that begins the conditional compilation block tests *identifier* to determine whether it is currently defined. Using this syntax, the $UNDEFINE statement deletes the definition of *identifier* from the symbol table, and the statements between the $ELSE and the $ENDIF statements are compiled.

If you use the $IFNDEF statement, on the other hand, and *identifier* is undefined, the statements between $IFDEF and $ENDIF are compiled. If *identifier* is not defined, the statements between $IFDEF and $ELSE are compiled.

Note: UniVerse does not support nested $DEFINE/$UNDEFINE statements.

* statement

Use the * statement to insert a comment in a UniVerse BASIC program. Comments explain or document various parts of a program. They are part of the source code only and are nonexecutable. They do not affect the size of the object code.

A comment must be a separate UniVerse BASIC statement, and can appear anywhere in a program. A comment must begin with one of the following comment designators:

- REM
- *
- !
- $*

Any text that appears between a comment designator and the end of a physical line is treated as part of the comment, not as part of the executable program. If a comment does not fit on one physical line, you can continue it on the next physical line only by starting the new line with a comment designator. If a comment appears at the end of a physical line containing an executable statement, you must put a semicolon (;) before the comment designator.

Syntax

***** [*comment.text*]

Example

The PRINT statement at the end of the third line is not executed because it follows the asterisk on the same line and is treated as part of the comment. Lines 4, 5, and 6 show how to include a comment in the same sequence of executable statements.

```
PRINT "HI THERE"; * Anything after the * is a comment
* This line is also a comment and does not print.
IF 5<6 THEN PRINT "YES"; * A comment; PRINT "PRINT ME"
IF 5<6 THEN
PRINT "YES"; * A comment
PRINT "PRINT ME"
END
```

This is the program output:

```
HI THERE
YES
YES
PRINT ME
```

< > operator

Use the < > operator (angle brackets) to extract or replace elements of a dynamic array.

Syntax

```
variable < field# [ ,value# [,subvalue#]] >
```

Parameters

Parameter	Description
variable	Specifies the dynamic array containing the data to be changed.
field#, value#, subvalue #	Delimiter expressions.

Angle brackets to the left of an assignment operator change the specified data in the dynamic array according to the assignment operator. For examples, see the REPLACE function, on page 325. Angle brackets to the right of an assignment operator indicate that an `EXTRACT` function is to be performed. For examples, see the FADD function, on page 155.

@ function

Use the @ function with the PRINT statement to control display attributes, screen display, and cursor positioning.

Note: You can save processing time by assigning the result of a commonly used @ function, such as @ (–1), to a variable, rather than reevaluating the function each time it is used.

Syntax

```
@ (column [,row])
@(–code [,arg ])
```

Parameters

Parameter	Description
column	Defines a screen column position.
row	Defines a screen row position.
–code	The terminal control code that specifies a particular screen or cursor function.
arg	Specifies further information for the screen or cursor function specified in *–code*.

Cursor positioning

You position the cursor by specifying a screen column and row position using the syntax @ (*column* [,*row*]). If you do not specify a row, the current row is the default. The top line is row 0, the leftmost column is column 0. If you specify a column or row value that is out of range, the effect of the function is undefined.

If you use the @ function to position the cursor, automatic screen pagination is disabled.

Screen and cursor controls

You can use the @ function with terminal control codes to specify various cursor and display operations using the syntax @ (*–code* [,*arg*]).

If you want to use mnemonics rather than the code numbers, you can use an insert file of equate names by specifying either of the following options when you compile your program:

```
$INCLUDE UNIVERSE.INCLUDE ATFUNCTIONS.H
```

```
$INCLUDE SYSCOM ATFUNCTIONS.INS.IBAS
```
(PIOPEN flavor only)

Note: Not all terminal control codes are supported by all terminal types. If the current terminal type does not support the code you specified, the function returns an empty string. You can use this to test whether your program operates correctly on a particular terminal, and whether you need to code any alternative actions.

If you issue multiple video attributes (such as blink and reverse video) at the same time, the result is undefined. See the description of the @ function, on page 32 for details of additive attributes.

The following table summarizes the characteristics of the terminal control codes, and the sections following the table give more information on each equate name:

Integer	Equate name	Function	Argument
–1	IT$CS	Screen clear and home	
–2	IT$CAH	Cursor home	
–3	IT$CLEOS	Clear to end of screen	
–4	IT$CLEOL	Clear to end of line	
–5	IT$SBLINK	Start blink	
–6	IT$EBLINK	Stop blink	
–7	IT$SPA	Start protect	
–8	IT$EPA	Stop protect	
–9	IT$CUB	Back space one character	Number of characters to back space
–10	IT$CUU	Move up one line	Number of lines to move
–11	IT$SHALF	Start half-intensity	
–12	IT$EHALF	Stop half-intensity	
–13	IT$SREV	Start reverse video	
–14	IT$EREV	Stop reverse video	
–15	IT$SUL	Start underlining	
–16	IT$EUL	Stop underlining	
–17	IT$IL	Insert line	Number of lines to insert
–18	IT$DL	Delete line	Number of lines to delete
–19	IT$ICH	Insert character	Number of lines to insert
–20	IT$SIRM	Set insert/replace mode	
–21	IT$RIRM	Reset insert/ replace mode	
–22	IT$DCH	Delete character	Number of characters to delete

Integer	Equate name	Function	Argument
–23	IT$AUXON	Auxiliary port on	
–24	IT$AUXOFF	Auxiliary port off	
–25	IT$TRON	Transparent auxiliary port on	
–26	IT$TROFF	Transparent auxiliary port off	
–27	IT$AUXDLY	Auxiliary port delay time	
–28	IT$PRSCRN	Print screen	
–29	IT$E80	Enter 80-column mode	
–30	IT$E132	Enter 132-column mode	
–31	IT$RIC	Reset inhibit cursor	
–32	IT$SIC	Set inhibit cursor	
–33	IT$CUD	Cursor down	Number of lines to move cursor
–34	IT$CUF	Cursor forward	Number of places to move cursor forward
–35	IT$VIDEO	Set video attributes	Additive attribute value
–36	IT$SCOLPR	Set color pair	Predefined color pairing
–37	IT$FCOLOR	Set foreground color	Foreground color code
–38	IT$BCOLOR	Set background color	Background color code
–39	IT$SLINEGRFX	Start line graphics	
–40	IT$ELINEGRFX	End line graphics	
–41	IT$LINEGRFXCH	Line graphics character	The required graphics character
–42	IT$DMI	Disable manual input	
–43	IT$EMI	Enable manual input	
–44	IT$BSCN	Blank screen	
–45	IT$UBS	Unblank screen	
–48	IT$SU	Scroll up	Number of lines to scroll
–49	IT$SD	Scroll down	Number of lines to scroll
–50	IT$SR	Scroll right	Number of columns to scroll
–51	IT$SL	Scroll left	Number of columns to scroll
–54	IT$SLT	Set line truncate	
–55	IT$RLT	Reset line truncate	
–56	IT$SNK	Set numeric keypad	
–57	IT$RNK	Reset numeric keypad	

Integer	Equate name	Function	Argument
–58	IT$SBOLD	Start bold	
–59	IT$EBOLD	End bold	
–60	IT$SSECUR	Start secure mode	
–61	IT$ESECUR	End secure mode	
–62	IT$SSCRPROT	Start screen protect mode	
–63	IT$ESCRPROT	End screen protect mode	
-64	IT$SLD	System line display	
–65	IT$SLR	System line reset	
–66	IT$SLS	System line set	
–70	IT$CHA	Cursor horizontal absolute	Column number to position cursor
–71	IT$ECH	Erase character	Number of characters to erase
–74	IT$NPC	Character to substitute for nonprinting character	
–75	IT$DISPLAY	EDFS main display attributes	
–76	IT$MINIBUF	EDFS mini-buffer display attributes	
–77	IT$LOKL	Lock line	The line number
–78	IT$UNLL	Unlock line	The line number
–79	IT$MARKSUBS	Display marks	
–80 through –100		Reserved for U2	
–101 through –128	IT$USERFIRST IT$USERLAST	Available for general use	

Screen clear and home @(IT$CS)

Clears the screen and positions the cursor in the upper-left corner.

Cursor home @(IT$CAH)

Moves the cursor to the upper-left corner of the screen.

Clear to end of screen @(IT$CLEOS)

Clears the current screen line starting at the position under the cursor to the end of that line and clears all lines below that line. The cursor does not move.

Clear to end of line @(IT$CLEOL)

Clears the current screen line starting at the position under the cursor to the end of that line. The cursor does not move.

Start blink @(IT$SBLINK)

Causes any printable characters that are subsequently displayed to blink. If you move the cursor before issuing the stop blink function, @(IT$EBLINK), the operation of the @(IT$SBLINK) code is undefined.

Stop blink @(IT$EBLINK)

Stops blink mode. If a start blink function, @(IT$SBLINK), was not transmitted previously, the effect of this sequence is undefined.

Start protect @(IT$SPA)

Protects all printable characters that are subsequently displayed from update until the characters are erased by one of the clear functions @(IT$CS), @(IT$CLEOS), or @(IT$CLEOL). If you move the cursor before issuing the stop protect function, @(IT$EPA), the operation of this code is undefined. The start protect function is useful only for terminals that are in block mode.

Stop protect @(IT$EPA)

Stops the protect mode. If a start protect string was not previously transmitted, the effect of this sequence is undefined. The stop protect function is useful only for terminals that are in block mode.

Back space one char @(IT$CUB)

Moves the cursor one position to the left without deleting any data. For *m* greater than 0, the function @(IT$CUB, *m*) moves the cursor *m* positions to the left. In moving to the left, the cursor cannot move beyond the start of the line.

Move up one line @(IT$CUU)

Moves the cursor up one line toward the top of the screen. For *m* greater than 0, the function @(IT $CUU, *m*) moves the cursor up *m* lines. The cursor remains in the same column, and cannot move beyond the top of the screen.

Start half-intensity @(IT$SHALF)

Causes all printable characters that are subsequently displayed to be displayed at reduced intensity. If a cursor-positioning sequence is used before the stop half-intensity function, @(IT$EHALF), the operation of this function is undefined.

Stop half-intensity @(IT$EHALF)

Terminates half-intensity mode. The effect of this sequence is unspecified if a start half-intensity string was not previously transmitted.

Start reverse video @(IT$SREV)

Causes printable characters that are subsequently displayed to be displayed with all pixels inverted. If a cursor-positioning sequence is used before the stop reverse video function, @(IT$EREV), the operation of this function is undefined.

Stop reverse video @(IT$EREV)

Terminates reverse video mode. If a start reverse video function, @(IT$SREV), was not previously transmitted, the effect of this sequence is undefined.

Start underlining @(IT$SUL)

Causes all subsequent printable characters to be underlined when displayed. If a cursor-positioning sequence is used before the stop underlining function, @(IT$EUL), the operation of this function is undefined.

Stop underlining @(IT$EUL)

Terminates the underlining mode established by a start underlining function, @(IT$SUL). The effect of this sequence is unspecified if a start underlining string was not previously transmitted.

Insert line @(IT$IL)

Inserts a blank line at the current cursor position. For *m* greater than 0, the function @(IT$IL, *m*) inserts m blank lines at the current cursor position. If *m* is omitted, the default is 1. The effect when *m* is less than 1 is undefined. All lines from the current cursor position to the end of the screen scroll down. The bottom *m* lines on the screen are lost.

Delete line @(IT$DL)

Deletes the line at the current cursor position; the function @(IT$DL, 1) has the same effect. For *m* greater than 1, the lines above the current line are deleted until *m* minus 1 lines have been deleted or the top of the file has been reached, whichever occurs first. All lines below the current cursor position scroll up. The last lines on the screen are cleared.

Insert character @(IT$ICH)

Inserts a space at the current cursor position. All characters from the cursor position to the right edge of the screen are shifted over one character to the right. Any character at the rightmost edge of the screen is lost. For *m* greater than 0, the function @(IT$ICH, *m*) inserts m spaces at the current cursor position, shifting the other characters accordingly.

Set insert/replace mode @(IT$SIRM)

Starts insert character mode. Characters sent to the terminal screen are inserted at the current cursor position instead of overwriting the character under the cursor. The characters under and to the right of the cursor are shifted over one character to the right for each character transmitted, and any character at the rightmost edge of the screen is lost.

Reset insert/replace mode @(IT$RIRM)

Turns off insert character mode. Characters sent to the terminal screen overwrite the characters at the current cursor position.

Delete character @(IT$DCH)

Deletes the character at the current cursor position. All characters to the right of the cursor move one space to the left, and the last character position on the line is made blank. For *m* greater than 1, the function @(IT$DCH, *m*) deletes further characters, to the right of the original position, until *m* characters have been deleted altogether or until the end of the display has been reached, whichever occurs first.

Auxiliary port on @(IT$AUXON)

Enables the auxiliary (printer) port on the terminal. All characters sent to the terminal are displayed on the screen and also copied to the auxiliary port.

Auxiliary port off @(IT$AUXOFF)

Disables the auxiliary (printer) port on the terminal, and stops the copying of the character stream to the auxiliary port.

Transparent auxiliary port on @(IT$TRON)

Places the auxiliary (printer) port on the terminal in transparent mode. All characters sent to the terminal are sent only to the auxiliary port and are not displayed on the terminal screen.

Transparent auxiliary port off @(IT$TROFF)

Disables the auxiliary (printer) port on the terminal and enables the display of the character stream on the terminal screen.

Auxiliary delay time @(IT$AUXDLY)

Sets a time, in milliseconds, that an application should pause after enabling or disabling the auxiliary port. The value of this function is an integer in the range 0 through 32,767. The function is used in conjunction with the !SLEEP$ subroutine; for example:

```
PRINT @(IT$AUXON)::CALL !SLEEP$(@(IT$AUXDLY))
```

Print screen @(IT$PRSCRN)

Copies the contents of the screen to the auxiliary port. The function does not work for some terminals while echo delay is enabled.

Enter 80-column mode @(IT$E80)

Starts 80-column mode. On some terminals it can also clear the screen.

Enter 132-column mode @(IT$E132)

Starts 132-column mode. On some terminals it can also clear the screen.

Reset inhibit cursor @(IT$RIC)

Turns the cursor on.

Set inhibit cursor @(IT$SIC)

Turns the cursor off.

Cursor down @(IT$CUD)

Moves the cursor down one line. For *m* greater than 0, the function @(IT$CUD, *m*) moves the cursor down *m* lines. The cursor remains in the same column, and cannot move beyond the bottom of the screen.

Cursor forward @(IT$CUF)

Moves the cursor to the right by one character position without overwriting any data. For *m* greater than 0, the function @(IT$CUF, *m*) moves the cursor *m* positions to the right. The cursor cannot move beyond the end of the line.

Set video attributes @(IT$VIDEO)

Is an implementation of the ANSI X3.64-1979 and ISO 6429 standards for the video attribute portion of Select Graphic Rendition. It always carries an argument *m* that is an additive key consisting of one or more of the following video attribute keys:

Value	Name	Description
0	IT$NORMAL	Normal
1	IT$BOLD	Bold
2	IT$HALF	Half-intensity
4	IT$STANDOUT	Enhanced
4	IT$ITALIC	Italic
8	IT$ULINE	Underline
16	IT$SLOWBLINK	Slow blink
32	IT$FASTBLINK	Fast blink
64	IT$REVERSE	Reverse video
128	IT$BLANK	Concealed
256	IT$PROTECT	Protected
572	IT$ALTCHARSET	Alternative character set

For example:

```
PRINT @(IT$VIDEO,IT$HALF+IT$ULINE+IT$REVERSE)
```

In this example, *m* is set to 74 (2 + 8 + 64) for half-intensity underline display in reverse video. Bold, italic, fast blink, and concealed are not supported on all terminals. To set the video attributes half-intensity and underline, specify the following:

```
@(-35,10)
```

In this example, 10 is an additive key composed of 2 (half-intensity) plus 8 (underline).

Set color pair @(IT$SCOLPR)

Sets the background and foreground colors to a combination that you have previously defined in your system *terminfo* file.

Set foreground color @(IT$FCOLOR)

Sets the color that is used to display characters on the screen. @(IT$FCOLOR,*arg*) always takes an argument that specifies the foreground color to be chosen, as follows:

Value	Name	Description
0	IT$63	Black
1	IT$RED	Red
2	IT$GREEN	Green
3	IT$YELLOW	Yellow
4	IT$BLUE	Blue

Value	Name	Description
5	IT$MAGENTA	Magenta
6	IT$CYAN	Cyan
7	IT$WHITE	White
8	IT$DARK.RED	Dark red
9	IT$CERISE	Cerise
10	IT$ORANGE	Orange
11	IT$PINK	Pink
12	IT$DARK.GREEN	Dark green
13	IT$SEA.GREEN	Sea green
14	IT$LIME.GREEN	Lime green
15	IT$PALE.GREEN	Pale green
16	IT$BROWN	Brown
17	IT$CREAM	Cream
18	IT$DARK.BLUE	Dark blue
19	IT$SLATE.BLUE	Slate blue
20	IT$VIOLET	Violet
21	IT$PALE.BLUE	Pale blue
22	IT$PURPLE	Purple
23	IT$PLUM	Plum
24	IT$DARK.CYAN	Dark cyan
25	IT$SKY.BLUE	Sky blue
26	IT$GREY	Grey

The color attributes are not additive. Only one foreground color at a time can be displayed. If a terminal does not support a particular color, a request for that color should return an empty string.

Set background color @(IT$BCOLOR)

Sets the background color that is used to display characters on the screen. The @(IT$BCOLOR, *arg*) function always has an argument that specifies the background color to be chosen. (See Set foreground color @(IT$FCOLOR) on page 65 for a list of available colors.)

Start line graphics @(IT$SLINEGRFX)

Switches on the line graphics mode for drawing boxes or lines on the screen.

End line graphics @(IT$ELINEGRFX)

Switches off the line graphics mode.

Line graphics character @(IT$LINEGRFXCH)

Specifies the line graphics character required. The argument can be one of the following:

Value	Token	Description
0	IT$GRFX.CROSS	Cross piece
1	IT$GRFX.H.LINE	Horizontal line
2	IT$GRFX.V.LINE	Vertical line

Value	Token	Description
3	IT$GRFX.TL.CORNER	Top-left corner
4	IT$GRFX.TR.CORNER	Top-right corner
5	IT$GRFX.BL.CORNER	Bottom-left corner
6	IT$GRFX.BR.CORNER	Bottom-right corner
7	IT$GRFX.TOP.TEE	Top-edge tee piece
8	IT$GRFX.LEFT.TEE	Left-edge tee piece
9	IT$GRFX.RIGHT.TEE	Right-edge tee piece
10	IT$GRFX.BOTTOM.TEE	Bottom-edge tee piece

Disable manual input @(IT$DMI)

Locks the terminal's keyboard.

Enable manual input @(IT$EMI)

Unlocks the terminal's keyboard.

Blank screen @(IT$BSCN)

Blanks the terminal's display. Subsequent output to the screen is not visible until the unblank screen function, @(IT$UBS), is used.

Unblank screen @(IT$UBS)

Restores the terminal's display after it was blanked. The previous contents of the screen, and any subsequent updates, become visible.

Scroll up @(IT$SU)

Moves the entire contents of the display up one line. For *m* greater than 0, the function @(IT$SU, *m*) moves the display up *m* lines or until the bottom of the display is reached, whichever occurs first. For each line that is scrolled, the first line is removed from sight and another line is moved into the last line. This function works only if the terminal is capable of addressing character positions that do not all fit on the screen, such that some lines are not displayed. This normally requires the terminal to be set to vertical two-page mode in the initialization string. The effect of attempting to scroll the terminal too far is undefined.

Scroll down @(IT$SD)

Moves the entire contents of the display down one line. For *m* greater than 0, the function @(IT$SD, *m*) moves the display down *m* lines or until the top of the display is reached, whichever occurs first. For each line that is scrolled, the last line is removed from sight and another line is moved into the top line. This function works only if the terminal is capable of addressing character positions that do not all fit on the screen, such that some lines are not displayed. This normally requires the terminal to be set to vertical two-page mode in the initialization string. The effect of attempting to scroll the terminal too far is undefined.

Scroll right @(IT$SR)

Moves the entire contents of the display one column to the right. For *m* greater than 0, the function @(IT$SR, *m*) moves the display *m* columns to the right or until the left edge of the display is reached, whichever occurs first. For each column scrolled, the rightmost column is removed from sight and another leftmost column appears. This function works only if the terminal is capable of addressing

character positions that do not fit on the screen, such that some columns are not displayed. This normally requires the terminal to be set to horizontal two-page mode in the initialization string. The effect of attempting to scroll the terminal too far is undefined.

Scroll left @(IT$SL)

Moves the entire contents of the display one column to the left. For *m* greater than 0, the function @(IT$SL, *m*) moves the display *m* columns to the left or until the right edge of the display is reached, whichever happens first. For each column scrolled, the leftmost column is removed from sight and another rightmost column appears. This function works only if the terminal is capable of addressing character positions that do not fit on the screen, such that some columns are not displayed. This normally requires the terminal to be set to horizontal two-page mode in the initialization string. The effect of attempting to scroll the terminal too far is undefined.

Set line truncate @(IT$SLT)

Makes the cursor stay in the last position on the line when characters are printed past the last position.

Reset line truncate @(IT$RLT)

Makes the cursor move to the first position on the next line down when characters are printed past the last position.

Set numeric keypad @(IT$SNK)

Sets keys on the numeric keypad to the labeled functions instead of numbers.

Reset numeric keypad @(IT$RNK)

Resets keys on the numeric keypad to numbers.

Start bold @(IT$SBOLD)

Starts bold mode; subsequently, any characters entered are shown more brightly on the screen.

End bold @(IT$EBOLD)

Ends bold mode; characters revert to normal screen brightness.

Start secure mode @(IT$SSECUR)

Characters entered in this setting are not shown on the screen. This function can be used when entering passwords, for example.

End secure mode @(IT$ESECURE)

Switches off secure mode; characters appear on the screen.

Start screen protect mode @(IT$SSCRPROT)

Switches on start protect mode. Characters entered in this mode are not removed when the screen is cleared.

End screen protect mode @(IT$ESCRPROT)

Switches off screen protect mode.

System line display @(IT$SLD)

Redisplays the user-defined characters that were sent by the system line set function, @(IT$SLS). The system line is defined as an extra line on the terminal display but is addressable by the normal cursor positioning sequence. On most terminals the system line normally contains a terminal status description. The number of usable lines on the screen does not change.

System line reset @(IT$SLR)

Removes from the display the characters that were set by the @(IT$SLS) function and replaces them with the default system status line. The number of usable lines on the screen does not change.

System line set @(IT$SLS)

Displays the user-defined status line, and positions the cursor at the first column of the status line. Subsequent printing characters sent to the terminal are displayed on the status line. Issuing a system line reset function, @(IT$SLR), terminates printing on the status line, and leaves the cursor position undefined. The characters printed between the issuing of @(IT$SLS) and @(IT$SLR) can be recalled subsequently and displayed on the line by issuing an @(IT$SLD) function.

Cursor horizontal absolute @(IT$CHA)

Positions the cursor at column *m* of the current line. If *m* is omitted, the default is 0. The @(IT$CHA, *m*) function must have the same effect as @(*m*).

Erase character @(IT$ECH)

Erases the character under the cursor and replaces it with one or more spaces, determined by the argument *m*. If you do not specify *m*, or you specify a value for *m* that is less than 2, only the character under the cursor is replaced. If you specify an argument whose value is greater than 1, the function replaces the character under the cursor, and *m* –1 characters to the right of the cursor, with spaces. The cursor position is unchanged.

IT$NPC, IT$DISPLAY, and IT$MINIBUF

Reserved for EDFS attributes.

Lock line @(IT$LOKL)

Locks line *n* of the screen display (top line is 0). The line cannot be modified, moved, or deleted from the screen until it is unlocked.

Unlock line @(IT$UNLL)

Unlocks line *n* of the screen display allowing it to be modified, moved, or deleted.

Display marks @(IT$MARKSUBS)

Returns the characters used to display UniVerse delimiters on screen. From left to right, the delimiters are: item, field, value, subvalue, and text.

Allocated for U2 @(–80) to @(–100)

These functions are reserved for U2.

Allocated for general use @(–101) to @(–128)

These functions are available for any additional terminal definition strings that you require.

Video attributes: points to note

Terminals whose video attributes are described as embedded or on-screen use a character position on the terminal screen whenever a start or stop video attribute is received. Programs driving such terminals must not change an attribute in the middle of a contiguous piece of text. You must leave at least one blank character position at the point where the attribute changes. The field in the terminal definition record called *xmc* is used to specify the number of character positions required for video attributes. A program can examine this field, and take appropriate action. To do this, the program must execute GET.TERM.TYPE and examine the @SYSTEM.RETURN.CODE variable, or use the definition VIDEO.SPACES from the TERM INFO.H file.

Many terminals do not clear video attributes automatically when the data on a line is cleared or deleted. The recommended programming practice is to reposition to the point at which a start attribute was emitted, and overwrite it with an end attribute, before clearing the line.

On some terminals you can set up the Clear to End of Line sequence to clear both data and video attributes. This is done by combining the strings for erase data from active position to end of line, selecting Graphic Rendition normal, and changing all video attributes from active position to end of line. Sending the result of the @(IT$CLEOL) function causes both the visible data on the line to be cleared, and all video attributes to be set to normal, after the cursor position.

Note: Where possible, you should try to ensure that any sequences that clear data also clear video attributes. This may not be the case for all terminal types.

An exception is @(IT$CS) clear screen. The sequence associated with this function should always clear not only all data on the screen but also reset any video attributes to normal.

Examples

The following example displays “Demonstration” at column 5, line 20:

```
PRINT @(5,20):"Demonstration"
```

In the next example, the PRINT statement positions the cursor to home, at the top-left corner of the screen, and clears the screen:

```
PRINT @(IT$CS):
```

The $INCLUDE statement is used to include the ATFUNCTIONS insert file of equate names. Assignment statements are used to assign the evaluated @ functions to variables. The variables are used in PRINT statements to produce code that clears the screen and returns the cursor to its original position; positions the cursor at column 5, line 20; turns on the reverse video mode; prints the string; and turns off the reverse video mode.

```
$INCLUDE UNIVERSE.INCLUDE ATFUNCTIONS.H
 CLS = @(IT$CS)
 REVERSE.ON = @(IT$SREV)
 REVERSE.OFF = @(IT$EREV)
 .
 .
 .
 PRINT CLS: @(5,20):
 PRINT REVERSE.ON:"THIS IS REVERSE VIDEO":REVERSE.OFF
```

The next example displays any following text in yellow letters:

```
PRINT @(IT$FCOLOR, IT$YELLOW)
```

The next example displays any following text on a cyan background:

```
PRINT @(IT$BCOLOR, IT$CYAN)
```

The next example gives a yellow foreground, not a green foreground, because color changes are not additive:

```
PRINT @(IT$FCOLOR, IT$BLUE):@(IT$FCOLOR, IT$YELLOW)
```

If you have a terminal that supports colored letters on a colored background, the next example displays the text “Hello” in yellow on a cyan background. All subsequent output is in yellow on cyan until another color @ function is used. If your color terminal cannot display colored foreground on colored background, only the last color command is used, so that this example displays the text “Hello” in yellow on a black background.

```
PRINT @(IT$BCOLOR,IT$CYAN):@(IT$FCOLOR,IT$YELLOW):"Hello"
```

If your color terminal cannot display colored foreground on colored background, the previous example displays the text “Hello” in black on a cyan background.

The next example gives the same result as the previous example for a terminal that supports colored letters on a colored background. Strings containing the @ functions can be interpreted as a sequence of instructions, which can be stored for subsequent frequent reexecution.

```
PRINT @(IT$FCOLOR,IT$YELLOW):@(IT$BCOLOR,IT$CYAN):"Hello"
```

In the last example, the screen is cleared, the cursor is positioned to the tenth column in the tenth line, and the text “Hello” is displayed in foreground color cyan. The foreground color is then changed to white for subsequent output. This sequence of display instructions can be executed again, whenever it is required, by a further PRINT SCREEN statement.

```
SCREEN = @(IT$CS):@(10,10):@(IT$FCOLOR,IT$CYAN):"Hello"
 SCREEN = SCREEN:@(IT$FCOLOR,IT$WHITE)
 PRINT SCREEN
```

[] operator

Use the [] operator (square brackets) to extract a substring from a character string. The bold brackets are part of the syntax and must be typed.

Syntax

expression **[** [*start*,] *length*]

expression **[** *delimiter*, *occurrence*, *fields*]

Parameters

Parameter	Description
expression	Evaluates to any character string.
start	An expression that evaluates to the starting character position of the substring. If *start* is 0 or a negative number, the starting position is assumed to be 1. If you omit *start*, the starting position is calculated according to the following formula: *string.length* – *substring.length* + 1 This lets you specify a substring consisting of the last *n* characters of a string without having to calculate the string length.

Parameter	Description
length	is an expression that evaluates to the length of the substring.

If *start* exceeds the number of characters in *expression*, an empty string results. An empty string also results if *length* is 0 or a negative number. If the sum of *start* and *length* exceeds the number of characters in the string, the substring ends with the last character of the string.

Use the second syntax to return a substring located between the specified number of occurrences of the specified delimiter. This syntax performs the same function as the FIELD function, on page 156.

Parameter	Description
delimiter	Any string, including field mark, value mark, and subvalue mark characters. It delimits the start and end of the substring (all that appears within the two delimiters). If *delimiter* consists of more than one character, only the first character is used.
occurrence	Specifies which occurrence of the delimiter is to be used as a terminator. If *occurrence* is less than 1, 1 is assumed.
fields	Specifies the number of successive fields after the delimiter specified by *occurrence* that are to be returned with the substring. If the value of *fields* is less than 1, 1 is assumed. The delimiter is part of the returned value in the successive fields.

If the delimiter or the occurrence specified does not exist within the string, an empty string is returned. If *occurrence* specifies 1 and no delimiter is found, the entire string is returned.

If *expression* is the null value, any substring extracted from it will also be the null value.

Examples

In the following example (using the second syntax) the fourth # is the terminator of the substring to be extracted, and one field is extracted:

```
A="###DHHH#KK"
 PRINT A["#",4,1]
```

This is the result:

```
DHHH
```

The following syntaxes specify substrings that start at character position 1:

expression [0, *length*]*expression* [–1, *length*]

The following example specifies a substring of the last five characters:

```
"1234567890" [5]
```

This is the result:

```
67890
```

All substring syntaxes can be used in conjunction with the assignment operator (=). The new value assigned to the variable replaces the substring specified by the [] operator. For example:

```
A='12345'
 A[3]=1212
 PRINT "A=",A
```

returns the following:

```
A=        12121
```

A[3] replaces the last three characters of A (345) with the newly assigned value for that substring (1212).

The FIELDSTORE function provides the same functionality as assigning the three-argument syntax of the [] operator.

ABORT statement

Use the ABORT statement to terminate execution of a BASIC program and return to the UniVerse prompt. ABORT differs from STOP in that a STOP statement returns to the calling environment (for example, a menu, a paragraph, another UniVerse BASIC program following an EXECUTE statement, and so on), whereas ABORT terminates all calling environments as well as the UniVerse BASIC program. You can use it as part of an IF…THEN statement to terminate processing if certain conditions exist.

Syntax

ABORT [*expression* …]

ABORTE [*expression* …]

ABORTM [*expression* …]

If *expression* is used, it is printed when the program terminates. If *expression* evaluates to the null value, nothing is printed.

The ABORTE statement is the same as the ABORT statement except that it behaves as if $OPTIONS statement STOP.MSG were in force. This causes ABORT to use the ERRMSG file to produce error messages instead of using the specified text. If *expression* in the ABORTE statement evaluates to the null value, the default error message is printed:

```
Message ID is NULL: undefined error
```

For information about the ERRMSG file, see the ERRMSG statement, on page 148.

The ABORTM statement is the same as the ABORT statement except that it behaves as if $OPTIONS -STOP.MSG were in force. This causes ABORT to use the specified text instead of text from the ERRMSG file.

Example

```
PRINT "DO YOU WANT TO CONTINUE?":
INPUT A
IF A="NO" THEN ABORT
```

This is the program output:

```
DO YOU WANT TO CONTINUE?
```

ABS function

Use the ABS function to return the absolute value of any numeric expression. The absolute value of an expression is its unsigned magnitude. If *expression* is negative, the value returned is:

-expression

For example, the absolute value of -6 is 6.

If *expression* is positive, the value of *expression* is returned. If *expression* evaluates to the null value, null is returned.

Syntax

ABS (*expression*)

Example

```
Y = 100
X = ABS(43-Y)
PRINT X
```

This is the program output:

```
57
```

ABSS function

Use the ABSS function to return the absolute values of all the elements in a dynamic array. If an element in *dynamic.array* is the null value, null is returned for that element.

Syntax

ABSS (*dynamic.array*)

Example

```
Y = REUSE(300)
Z = 500:@VM:400:@VM:300:@SM:200:@SM:100
A = SUBS(Z,Y)
PRINT A
PRINT ABSS(A)
```

This is the program output:

```
200V100V0S-100S-200
 200V100V0S100S200
```

acceptConnection function

Use the acceptConnection() function to accept an incoming connection attempt on the server side socket.

Syntax

acceptConnection(*svr_socket, blocking_mode, timeout, in_addr, in_name, socket_handle*)

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
svr_socket	The handle to the server side socket which is returned by `initServerSocket`().
blocking_mode	*blocking_mode* is one of the following: ▪ 0: using current mode. ▪ 1: blocking mode (default). If this mode and the current blocking mode of *svr_socket* is set to blocking, `acceptConnection`() blocks the caller until a connection request is received or the specified *time_out* has expired. ▪ 2: non-blocking mode. In this mode, if there are no pending connections present in the queue, `acceptConnection`() returns an error status code. If this mode, *time_out* is ignored.
time_out	Timeout in milliseconds.
in_addr	The buffer that receives the address of the incoming connection. If NULL, it will return nothing.
in_name	The variable that receives the name of the incoming connection. If NULL, it will return nothing.
socket_handle	The handle to the newly created socket on which the actual connection will be made. The server will use `readSocket`(), `writeSocket`(), and so forth with this handle to communicate with the client.

Return codes

The following table describes the status of each return code.

Return code	Description
0	Success.
1-41	See Socket function error return codes, on page 584.
102	SSL Handshake failure.
103	No client certificate.
105	Client authentication failure.
106	Peer not speaking SSL.

ACOS function

Use the `ACOS` function to return the trigonometric arc-cosine of *expression*. *expression* must be a numeric value. The result is expressed in degrees. If *expression* evaluates to the null value, null is returned. The `ACOS` function is the inverse of the `COS` function.

Syntax

ACOS (*expression*)

Example

```
PRECISION 5
```

```
PRINT "ACOS(0.707106781) = ":ACOS(0.707106781):" degrees"
```

This is the program output:

```
ACOS(0.707106781) = 45 degrees
```

ACTIVATEKEY statement

Use the ACTIVATEKEY command to activate a key. It is necessary to activate a key if you want to supply a password for key protection.

Syntax

ACTIVATEKEY *<key.id>, <password>* [ON *<hostname>*]

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
key.id	The key ID to activate.
password	The password corresponding to *key.id*.
ON *hostname*	The name of the remote host on which you want to activate the encryption key.

Note: You can activate only keys with password protection with this command. Keys that do not have password protection are automatically activated. Also, you can activate only keys to which you are granted access.

Use the STATUS function after an ACTIVATEKEY statement is executed to determine the result of the operation, as follows:

Value	Description
0	Operation successful.
1	Key is already activated. This applies to a single key, not a wallet operation.
2	Operation failed. This applies to a single key, not a wallet operation.
3	Invalid key or wallet ID or password.
4	No access to wallet.
5	Invalid key ID or password in a wallet.
6	No access to one of the keys in the wallet.
9	Other error.

addAuthenticationRule function

The addAuthenticationRule() function adds an authentication rule to a security context. The rules are used during SSL negotiation to determine whether the peer is to be trusted.

Syntax

addAuthenticationRule(*context, ServerOrClient, Rule, RuleString*)

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
context	The security context handle.
ServerOrClient	Flag: 1- Server (SSL_SERVER) 2- Client (SSL_CLIENT) Any other value is treated as a value of 1.
Rule	The rule name string. Valid settings are: ▪ SSL_RULE_STRENGTH ▪ SSL_RULE_PEER_NAME ▪ SSL_RULE_CERTPATH ▪ SSL_RULE_SERVER_NAME
RuleString	Rule content string. Can be attribute-mark separated.

Return code status

The following table describes the status of each return code.

Return code	Status
0	Success.
1	Invalid security context handle.
2	Invalid rule name.
3	Invalid rule content.

VerificationStrength rule

This VerificationStrength rule (SSL_RULE_STRENGTH) governs the SSL negotiation and determines whether an authentication process is considered successful. There are two levels of security: generous and strict. If you specify generous, the certificate need only contain the subject name (common name) that matches one specified by “PeerName”, to be considered valid. There is no need to have its complete certificate chain established. If you specify strict, the incoming certificate must pass a number of checks, including signature check, expiry check, purpose check, and issuer check. A complete certificate chain must be established.

Note: Setting the rule to **generous** is recommended only for development or testing purposes.

PeerName rule

By specifying the PeerName rule (SSL_RULE_PEER_NAME) and attribute-mark separated common names in ruleString, trusted server/client names will be stored into the context.

During the SSL handshake negotiation, the server will send its certificate to the client. By specifying trusted server names, the client can control which server or servers it should communicate with. During the handshake, once the server certificate has been verified by way of the establishment of the complete certificate chain, the subject name contained in the certificate will be compared against the trusted server names set in the context. If the server subject name matches one of the trusted names, communication will continue, otherwise the connection will not be established.

If no trusted peername is set, then any peer is considered legitimate.

CertificatePath rule

The CertificatePath rule (SSL_RULE_CERTPATH) enables you to specify locations in which to search for certificates. From the list of options, choose a CertificatePath rule to specify the search path:

- Default – When you add a certificate to a security context record, the full path for that certificate is registered in the security context record. This path is derived from the current directory in which UniData or UniVerse is running. When the certificate is loaded into memory to establish the SSL connection, UniData or UniVerse by default uses this registered full path to retrieve the certificate.
- Relative – With this option, UniData or UniVerse looks for the certificate in the current directory in which it is running.

  Note: Some of the UniData or UniVerse processes, such as the Telnet server processes, run from the system directory.

- Path – With this option, UniData or UniVerse uses the path you specify for loading the certificate added to this security context record. You can specify either an absolute path or a relative path.
- Env – If you select this option, enter an environment variable name in the **Env** text box. With this option, the UniData or UniVerse process first obtains the value of the environment variable you specify, and then uses that value as the path to load the certificates.

  Note: UniData or UniVerse evaluates the environment variable only when the first SSL connection is made. The value is cached for later reference.

ServerName rule

The ServerName rule (SSL_RULE_SERVER_NAME or Server Name Indication - SNI) is an extension to the TLS computer networking protocol by which a client indicates which hostname it is attempting to connect to at the start of the handshaking ("client hello") process. This rule allows a server to present multiple certificates on the same IP address and TCP port number. As a result, it allows multiple secure (HTTPS) websites or any other service over TLS to be served off the same IP address without requiring all those sites to use the same certificate.

See the RFC 6066 standard for more information. In order to provide any of the server names, clients can include an extension of type "server_name" in the extended "client hello."

If a secure HTTP request is requested and the specified protocols include at least one TLS version, and a ServerName rule exists in the SCR, then an SNI extension will be added to the protocol handshake, allowing users to connect to a server that serves different virtual hosts on a single IP address.

addCertificate function

The `addCertificate`() function stores a certificate (or multiple certificates) into a security context to be used as a UniData or UniVerse server or client certificate. Alternatively, it can specify a certificate or a directory which contains the certificates that are either used as CA (Certificate Authority) certificates to verify incoming certificates or act as a Revocation list to check against expired or revoked certificates.

There are three kinds of certificates:

- Self-signed root certificate, or root CA certificate – these certificates are used to sign other certificates as a means to vouch for the authenticity of holders of those certificates.

- Intermediate CA certificates – these certificates are signed by a root CA certificate or another intermediate CA certificate and are used to sign other certificates.
- Server/client certificates – these certificates are signed by root CA or intermediate CA certificates, and are used by a server or client to provide its identity.

Root CA or Intermediate certificates are sometimes also called Issuer certificates.

For a server/client certificate, a complete certificate chain contains all the certificates starting from the server/client certificate to its immediate intermediate CA certificate (and the intermediate CA certificate's immediate intermediate CA certificates, if any), up to the root CA certificate. To verify a server/client certificate, the complete certificate chain needs to be established. For UniData and UniVerse, this means that all intermediate root CA certificates must be specified in the security context record. Note that sometimes the intermediate CA certificates can be sent from a server or client, along with the server client certificate. In this case, you only need to add the root CA certificate to the security context record.

A certificate's purpose is to bind an entity's name with its public key. It is a means of distributing public keys. A certificate always contains three pieces of information: a name that identifies the owner of this certificate, a public key of this owner, and a digital signature signed by a trusted third party called a Certificate Authority (CA) with its private key. If you have the CA's public key, you can verify that the certificate is authentic, that is, whether the public key contained in the certificate is indeed associated with the entity specified with the name in the certificate. In practice, a certificate can and often does contain more information, for example, the period of time the certificate is valid.

SSL protocol specifies that when two parties start an SSL handshake, the server must always send its certificate to the client for authentication. It might optionally require the client to send its certificate to the server for authentication as well. Therefore, UniData and UniVerse applications that act as HTTPS clients are not required to maintain a client certificate. The application should work with web servers that do not require client authentication, while UniData and UniVerse applications that do act as SSL servers must install a server certificate.

Regardless of which role the application is going to assume, it needs to install a CA certificate or a CA certificate chain to be able to verify an incoming certificate.

Syntax

addCertificate(*certPath*, *usedAs*, *format*, *algorithm*, *context*, *p12pass*)

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
certPath	A string containing the name of the OS level file that holds the certificate, or the directory containing certificates.
usedAs	Flag 1- Used as a client/server certificate (SSL_CERT_SELF) 2- Used as an issuer certificate (SSL_CERT_CA) 3- Used as a certificate revocation list (SSL_CERT_CRL)
format	1 - PEM (Base64 encoded) format (SSL_FMT_PEM) 2 - DER (ASN.1 binary) format (SSL_FMT_DER) 3 - PKCS #12 format (SSL_FMT_P12)

Parameter	Description
algorithm	Flag 1- RSA key (SSL_KEY_RSA) 2- DSA key (SSL_KEY_DSA)
context	The security context handle.
p12pass	Optional. Sets a password on the PKCS #12 file. This parameter should only be included if using a PKCS #12 certificate that has a password. Otherwise the parameter should be omitted.

Note: To use the predefined constants, you must include SSL.H in your program. The value for PKCS #12 file format is 3. If you include the SSL.H shipped with UniVerse in your BASIC program, you can also use the predefined format constant SSL_FMT_P12.

Return code status

The following table describes the status of each return code.

Return code	Status
0	Success.
1	Invalid security context handle.
2	Certificate file could not be opened or directory does not exist.
3	Unrecognized format.
4	Corrupted or unrecognized certificate contents.
5	Invalid parameter value(s).

addRequestParameter function

The `addRequestParameter` function adds a parameter to the request.

Syntax

addRequestParameter(*request_handle, parameter_name, parameter_value, content_handling*)

Parameters

Parameter	Description
request_handle	The handle to the request.
parameter_name	The name of the parameter.
parameter_value	The value of the parameter.
content_handling	The dynamic MIME type for the parameter value.

Return codes

The following table describes the status of each return code.

Return code	Status
0	Success.

Return code	Status
1	Invalid request handle.
2	Invalid parameter.
3	Bad content type.

Note: For a GET request, *content_handling* is ignored.

For a POST request with default content type, the default for content handling is “ContentType:text/plain” if *content_handling* is not specified. For a POST request with “Multipart/*” content-type, content_handling is a dynamic array containing Content-* strings separated by field marks (@FM). They will be included in the multipart message before the data contained in *parameter_value* is sent. An example of *content_handling*:

```
Content-Type: application/XML @FM
 Content-Dispostion: attachment; file=”C:\drive\test.dat @FM
 Content-Length: 1923
```

Specifically, for a POST request with content type “multipart/form-data,” a “Content-Disposition:form-data” header will be created (or, in the case of Content-Dispostion already in *content_handling*, “form-data” will be added to it).

For both a GET and a POST request with either no content type specified or specified as “application/x-www-form-urlencoded,” as described in createRequest(), URL encoding is performed on data in *parameter_value* automatically. Basically, any character other than alphanumeric is considered “unsafe” and will be replaced by %HH, where HH is the ASCII value of the character in question. For example, “#” is replaced by %23, and “/” is replaced by %2F, and so forth. One exception is that by convention, spaces (‘ ‘) are converted into “+”.

For a POST method with other MIME-type specified, no encoding is done on data contained in *parameter_value*.

ADDS function

Use the ADDS function to create a dynamic array of the element-by-element addition of two dynamic arrays.

Each element of *array1* is added to the corresponding element of *array2*. The result is returned in the corresponding element of a new dynamic array. If an element of one array has no corresponding element in the other array, the existing element is returned. If an element of one array is the null value, null is returned for the sum of the corresponding elements.

If you use the subroutine syntax, the resulting dynamic array is returned as *return.array*.

Syntax

ADDS (*array1, array2*)

CALL **-ADDS** (*return.array, array1, array2*)

CALL **!ADDS** (*return.array, array1, array2*)

Example

```
A = 2:@VM:4:@VM:6:@SM:10
B = 1:@VM:2:@VM:3:@VM:4
PRINT ADDS(A,B)
```

This is the program output:

```
3V6V9S10V4
```

ALPHA function

Use the ALPHA function to determine whether *expression* is an alphabetic or non-alphabetic string. If *expression* contains the characters a through z or A through Z, it evaluates to true and a value of 1 is returned. If *expression* contains any other character or an empty string, it evaluates to false and a value of 0 is returned. If *expression* evaluates to the null value, null is returned.

If NLS is enabled, the ALPHA function uses the characters in the Alphabetics field in the NLS.LC.CTYPE file. For more information, see the *NLS Guide*.

Syntax

ALPHA (*expression*)

Example

```
PRINT "ALPHA('ABCDEFG') = ":ALPHA('ABCDEFG')
PRINT "ALPHA('abcdefg') = ":ALPHA('abcdefg')
PRINT "ALPHA('ABCDEFG.') = ":ALPHA('ABCDEFG.')
PRINT "ALPHA('SEE DICK') = ":ALPHA('SEE DICK')
PRINT "ALPHA('4 SCORE') = ":ALPHA('4 SCORE')
PRINT "ALPHA('') = ":ALPHA('')
```

This is the program output:

```
ALPHA('ABCDEFG') = 1
ALPHA('abcdefg') = 1
ALPHA('ABCDEFG.') = 0
ALPHA('SEE DICK') = 0
ALPHA('4 SCORE') = 0
ALPHA('') = 0
```

amInitialize function

The amInitialize function creates and opens an AMI session. The *hSession* output parameter is a session handle which is valid unless the session is terminated. The function returns a status code indicating success, warning, or failure. You can also use the STATUS() function to obtain this value.

Syntax

amInitialize(*hSession, appName, policyName, reasonCode*)

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
hSession	Upon successful return, holds a handle to a session. You can then use this handle in other UniData and UniVerse WebSphere MQ API calls. [OUT]
appName	An optional name you can use to identify the session. [IN]

Parameter	Description
policyName	The name of a policy. If you specify "" (null), the system default policy name is used. [IN]
reasonCode	Holds an AMI Reason Code when the function returns a status indicating an AMI warning or an AMI error occurred. The AMI Reason Code can be used to obtain more information about the cause of the warning or error. See the *MQSeries Application Messaging Interface* manual for a list of AMI Reason Codes and their descriptions. [OUT]

Return codes

The following table describes the status of each return code.

Return code	Status
0 – AMCC_SUCCESS	Function completed successfully.
1 – AMCC_WARNING	A warning was returned from AMI. The *reasonCode* output parameter contains an AMI reason code with further details about the warning.
2 – AMCC_FAILED	An error was returned from AMI. The *reasonCode* output parameter contains an AMI reason code with further details about the error.
99 – IPHANTOM_LICN_ERROR	Failed to get an IPHANTOM license.
100 – U2AMI_ERR_MQUNAVAILABLE	MQ AMI libraries are not available.
101 – U2AMI_ERR_UNKNOWN	Unknown error.
102 – U2AMI_ERR_NOBINDIR	The UVBIN environment variable was not found.
103 – U2AMI_ERR_FORK	Error during the execution of AMI pipes to AMI process.
104 – U2AMI_ERR_PIPECREATE	Error creating pipes to AMI process.
105 – U2AMI_ERR_PIPEWRITETOAMI	Error writing to pipe of AMI process.
106 – U2AMI_ERR_PIPEREADFROMAMI	Error reading pipe from AMI process.
107 – U2AMI_ERR_PIPEWRITETOU2	Error writing to pipe of U2 process.
108 – U2AMI_ERR_PIPEREADFROMU2	Error reading pipe from U2 process.
109 – U2AMI_ERR_EXEC	Error during execution of AMI process.
110 – U2AMI_ERR_INVALIDFORMAT	Variable does not match required format.
111 – U2AMI_ERR_NOT_HANDLE	Variable not of type MQShandle.
112 – U2SAMI_ERR_NULL_HANDLE	Uninitialized handle.
113 – U2AMI_ERR_INVALID_HANDLE	Handle has been closed with amTermnate.
114 – U2AMI_ERR_UNKNOWN_HANDLE	Unexpected handle value reported.
115 – U2AMI_ERR_SESSION_IN_USE	An active session already exists (under a different *hSession* variable than the one being passed in. See Usage Notes for more details).

Return code	Status
116 – U2AMI_ERR_CREATE_HANDLE	Error creating U2AMI session handle.
117 – U2AMI_ERR_DL_OPEN	Error opening MQ AMI library.
118 – U2AMI_ERR_DL_FUNC	Error calling function in MQ AMI library.
119 – U2AMI_ERR_RCVMSGOPTS	Invalid amRcvMsgOptions passed in.
Other	A non-AMI error occurred.

Usage Notes

Only one session can be active at one time. If you call amInitialize while another session is active, AMI returns an error code of U2AMI_ERR_SESSION_IN_USE. The one exception to this case is if the subsequent call to amInitialize uses the same *hSession* variable from the first call. In this case, the session is automatically terminated using the same AMI policy with which it was initialized, and a new session is started in its place.

amReceiveMsg function

The `amReceiveMsg` function receives a message sent by the `amSendMsg` function.

Syntax

amReceiveMsg(*hSession, receiverName, policyName, selMsgName, maxMsgLen, dataLen, data, rcvMsgName, reasonCode*[,*recMsgOption*])

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
hSession	The session handle returned by the `amInitialize` function. [IN]
receiverName	The name of a receiver service. If you specify "" (null), the system default name is used. [IN]
policyName	The name of a policy. If you specify "" (null), the system default policy name is used. [IN]
selMsgName	An optional parameters specifying the name of a message object containing information (such as a Correl ID) that will be used to retrieve the required message from the queue. See Usage Notes for additional information about the use of this parameter. [IN]
maxMsgLen	The maximum message length the application will accept. Specify as -1 to accept messages of any length, or use the optional parameter U2AMI_RESIZE BUFFER. See Usage Notes for additional information about the use of this parameter. [IN]
dataLen	The length of the received message data, in bytes. If this parameter is not required, specify as "" (null). [OUT]
data	The received message data. [OUT]
rcvMsgName	The name of a message object for the retrieved message. If you specify "" (null) for this parameter, the system default name (constant AMSD_RCV_MSG) is used. See Usage notes for additional information about the use of this parameter. [IN]

Parameter	Description	
reasonCode	Holds an AMI Reason Code when the function returns a status indicating an AMI warning or an AMI error occurred. The AMI Reason Code can be used to obtain more information about the cause of the warning or error. See the *MQSeries Application Messaging Interface* manual for a list of AMI Reason Codes and their descriptions. [OUT]	
recMsgOption	U2AMI_RECEIVEMSG	This is the default behavior. It returns both the message and the message length into the respective output parameters of the amReceiveMsg function.
	U2AMI_LEAVEMSG	If you specify U2AMI_LEAVEMSG for this parameter, and Accept Truncated Messages is not set in the policy receive attributes, UniVerse returns the message length in the *dataLen* parameter, but the message itself remains on the queue.
	U2AMI_DISCARDMSG	If you specify U2AMI_DISCARDMSG for this parameter and Accept Truncated Messages is set in the policy receive attributes, UniVerse discards the message at the MQSeries level with an AMRC_MSG_TRUNCATED warning. This behavior is preferable to discarding the message at the UniVerse level.
	U2AMI_RESIZEBUFFER	If you specify U2AMI_RESIZEBUFFER for this parameter, UniVerse handles the details of the buffer size used to retrieve the message. If you do not specify this parameter, you must specify the buffer size. See Usage Notes for more information about this option.

Return codes

The following table describes the status of each return code.

Return code	Description
0 – AMCC_SUCCESS	Function completed successfully.
1 – AMCC_WARNING	A warning was returned from AMI. The *reasonCode* output parameter contains an AMI reason code with further details about the warning.
2 – AMCC_FAILED	An error was returned from AMI. The *reasonCode* output parameter contains an AMI reason code with further details about the error.
Other	A non-AMI error occurred.

Usage notes

The *selMsgtName* parameter:

You can use the *selMsgName* parameter in Request/Reply messaging to tell amReceiveMsg to retrieve only those messages from the queue that correlate with a message previously placed on the queue with the amSendRequest function. When you use *selMsgName* in this manner, you should use the *sndMsgName* parameter of the amSendRequest call as the value for *selMsgName* in amReceiveMsg. Message correlation occurs here due to the following:

- The underlying message object created when the request message was sent, and referenced by the name *sndMsgName*, holds information about the sent message, such as its Correlation ID and Message ID.
- When you use this message object (*sndMsgName*) as the *selMsgName* parameter to `amReceiveMsg`, the information held in this message object is used to ensure that the function retrieves only correlating response messages from the queue.

The *maxMsgLen* parameter:

You can use the *maxMsgLen* parameter to define the maximum length message that the `amReceiveMsg` function retrieves from the queue. If the value of *maxMsgLen* is less than the length of the message to retrieve, behavior depends on the value of the *Accept Truncated Message* parameter in the policy receive attribute. If *Accept Truncated Message* is set to true, the `amReceiveMsg` function truncates the data, and there is an AMRC_MSG_TRUNCATED warning in the *reasonCode* parameter. If *Accept Truncated Message* is set to false, the default, the `amReceiveMsg` function fails with return status AMCC_FAILED(2), and the reason code is AMRC_RECEIVE_BUFF_LEN_ERR.

Note: If `amReceiveMsg` returns AMRC_RECEIVE_BUFF_LEN_ERR as the *reasonCode*, the *dataLen* parameter contains the message length, even though the call failed with return value MQCC_FAILED.

If you do not specify the U2AMI_RESIZE BUFFER optional parameter and the buffer size you specify with the *maxMsgLen* parameter is too small, the function fails with the AMRC_RECEIVE_BUFF_LEN_ERR. If this happens, UniVerse returns the necessary buffer size in the *dataLen* parameter so you can reissue the request with the correct size.

If you specify the U2AMI_RESIZEBUFFER parameter, UniVerse uses a default buffer size of 8K. If this buffer size is too small, UniVerse automatically reissues the request with the correct buffer size. While convenient, this behavior can result in performance degradation for the following reasons:

- If the default buffer size is larger than necessary for the received message, UniVerse incurs an unnecessary overhead.
- If the default buffer size is too small for the received message, UniVerse must issue to requests to the queue before successfully retrieving the message.

For performance reasons, we recommend you set the *maxMsgLen* parameter to the expected size of the message whenever possible.

The *rcvMsgName* parameter:

The *rcvMsgName* parameter enables the application to attach a name to the underlying message object used to retrieve the message. Although UniVerse supports this parameter, it is mainly intended for future additions to the WebSphere MQ for UniData and UniVerse API.

amReceiveRequest function

The `amReceiveRequest` function receives a request message.

Syntax

```
amReceiveRequest(hSession, receiverName, policyName, maxMsgLen,
dataLen, data, rcvMsgName, senderName, reasonCode [,recReqOption])
```

Parameters

The following table describes each parameter of the syntax.

Parameter	Description	
hSession	The session handle returned by the `amInitialize` function. [IN]	
receiverName	The name of a receiver service. If you specify "" (null), the system default name is used. [IN]	
policyName	The name of a policy. If you specify "" (null), the system default policy name is used. [IN]	
maxMsgLen	The maximum message length the application will accept. Specify as -1 to accept messages of any length, or use the optional parameter U2AMI_RESIZE BUFFER. See Usage Notes for additional information about the use of this parameter. [IN]	
dataLen	The length of the received message data, in bytes. If this parameter is not required, specify as "" (null). [OUT]	
data	The received message data. [OUT]	
rcvMsgName	The name of the message object for the retrieved message. If you specify "" (null), the system default receiver name is used. amReceiveRequest uses the value of *rcvMsgName* in the subsequent call to the amSendResponse function. [OUT]	
senderName	The name of a special type of sender service known as a response sender, to which the response message will be sent. If you do not specify a name, the system default response sender service is used. [IN] **Note:** The sender name you specify must not exist in your AMI repository.	
reasonCode	Holds an AMI Reason Code when the function returns a status indicating an AMI warning or an AMI error occurred. The AMI Reason Code can be used to obtain more information about the cause of the warning or error. See the *MQSeries Application Messaging Interface* manual for a list of AMI Reason Codes and their descriptions. [OUT]	
recReqOption	U2AMI_RECEIVEMSG	This is the default behavior. It returns both the message and the message length into the respective output parameters of the amReceiveReq function.
	U2AMI_LEAVEMSG	If you specify U2AMI_LEAVEMSG for this parameter, and Accept Truncated Messages is not set in the policy receive attributes, UniVerse returns the message length in the *dataLen* parameter, but the message itself remains on the queue.
	U2AMI_DISCARDMSG	If you specify U2AMI_DISCARDMSG for this parameter and Accept Truncated Messages is set in the policy receive attributes, UniVerse discards the message at the MQSeries level with an AMRC_MSG_TRUNCATED warning. This behavior is preferable to discarding the message at the UniVerse level.
	U2AMI_RESIZEBUFFER	If you specify U2AMI_RESIZEBUFFER for this parameter, UniVerse handles the details of the buffer size used to retrieve the message. If you do not specify this parameter, you must specify the buffer size. See Usage Notes for more information about this option.

Return codes

The following table describes the status of each return code.

Return code	Description
0 – AMCC_SUCCESS	Function completed successfully.
1 – AMCC_WARNING	A warning was returned from AMI. The *reasonCode* output parameter contains an AMI reason code with further details about the warning.
2 – AMCC_FAILED	An error was returned from AMI. The *reasonCode* output parameter contains an AMI reason code with further details about the error.
Other	A non-AMI error occurred.

Usage notes

The *maxMsgLen* parameter:

You can use the *maxMsgLen* parameter to define the maximum length message the amReceiveRequest will retrieve from the queue. If the value of *maxMsgLen* is less than the length of the message to retrieve, behavior depends on the value of *Accept Truncated Message* in the policy receive attributes. If the value of *Accept Truncated Message* is true, amReceiveRequest truncates the data and there is an AMRC_MSG_TRUNCATED warning in the *reasonCode* parameter. If the value of *Accept Truncated Message* is false, the default, amReceiveRequest fails with a return status AMCC_FAILED (2), and reason code AMRC_RECEIVE_BUFF_LEN_ERR.

> **Note:** If amReceiveRequest returns AMRC_RECEIVE_BUFF_LEN_ERR as the *reasonCode*, the message length is contained in the *dataLen* parameter, even thought the call failed with return value MQCC_FAILED.

If you do not specify the U2AMI_RESIZE BUFFER optional parameter and the buffer size you specify with the *maxMsgLen* parameter is too small, the function fails with the AMRC_RECEIVE_BUFF_LEN_ERR. If this happens, UniVerse returns the necessary buffer size in the *dataLen* parameter so you can reissue the request with the correct size.

If you specify the U2AMI_RESIZEBUFFER parameter, UniVerse uses a default buffer size of 8K. If this buffer size is too small, UniVerse automatically reissues the request with the correct buffer size. While convenient, this behavior can result in performance degradation for the following reasons:

- If the default buffer size is larger than necessary for the received message, UniVerse incurs an unnecessary overhead.
- If the default buffer size is too small for the received message, UniVerse must issue to requests to the queue before successfully retrieving the message.

For performance reasons, we recommend you set the *maxMsgLen* parameter to the expected size of the message whenever possible.

amSendMsg function

The `amSendMsg` function sends a datagram (send and forget) message.

Syntax

`amSendMsg``(hSession, senderName, policyName, data, sndMsgName, reasonCode)`

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
hSession	The session handle returned by the `amInitialize` function. [IN]
senderName	The name of a sender service. If you specify "" (null), the system default sender name is used. [IN]
policyName	The name of a policy. If you specify "" (null), the system default policy name is used. [IN]
data	The message data to be sent. [IN]
sndMsgName	The name of a message object for the message being sent. If you specify "" (null), the system default policy name is used. [IN]
reasonCode	Holds an AMI Reason Code when the function returns a status indicating an AMI warning or an AMI error occurred. The AMI Reason Code can be used to obtain more information about the cause of the warning or error. See the *MQSeries Application Messaging Interface* manual for a list of AMI Reason Codes and their descriptions. [OUT]

Return codes

The following table describes the status of each return code.

Return code	Description
0 – AMCC_SUCCESS	Function completed successfully.
1 – AMCC_WARNING	A warning was returned from AMI. The *reasonCode* output parameter contains an AMI reason code with further details about the warning.
2 – AMCC_FAILED	An error was returned from AMI. The *reasonCode* output parameter contains an AMI reason code with further details about the error.
Other	A non-AMI error occurred.

You can also use the UniVerse BASIC `STATUS()` function to obtain the return status from the function.

amSendRequest function

The `amSendRequest` function sends a request message.

Syntax

amSendRequest(*hSession, senderName, policyName, responseName, data, sndMsgName, reasonCode*)

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
hSession	The session handle returned by the `amInitialize` function. [IN]
senderName	The name of a sender service. If you specify "" (null), the system default sender name is used. [IN]

Parameter	Description
policyName	The name of a policy. If you specify "" (null), the system default policy name is used. [IN]
responseName	The name of the receiver service to which the response to this send request should be sent. Specify as “” (null) if no response is required. [IN]
data	The message data to be sent. [IN]
sndMsgName	The name of a message object for the message being sent. If you specify “” (null), amSendRequest uses the system default message name (constant AMSD_SND_MSG). [IN]
reasonCode	Holds an AMI Reason Code when the function returns a status indicating an AMI warning or an AMI error occurred. The AMI Reason Code can be used to obtain more information about the cause of the warning or error. See the *MQSeries Application Messaging Interface* manual for a list of AMI Reason Codes and their descriptions. [OUT]

Return codes

The following table describes the status of each return code.

Return code	Description
0 – AMCC_SUCCESS	Function completed successfully.
1 – AMCC_WARNING	A warning was returned from AMI. The *reasonCode* output parameter contains an AMI reason code with further details about the warning.
2 – AMCC_FAILED	An error was returned from AMI. The *reasonCode* output parameter contains an AMI reason code with further details about the error.
Other	A non-AMI error occurred.

You can also use the UniVerse BASIC `STATUS()` function to obtain the return status from the function.

amSendResponse function

The `amSendResponse` function sends a request message.

Syntax

amSendResponse(*hSession, senderName, policyName, rcvMsgName, data, sndMsgName, reasonCode*)

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
hSession	The session handle returned by the `amInitialize` function. [IN]
senderName	The name of a sender service. If you specify "" (null), the system default sender name is used. [IN]
policyName	The name of a policy. If you specify "" (null), the system default policy name is used. [IN]
rcvMsgName	The name of the received message to which this message is a response. You must set this parameter to the *rcvMsgName* specified for the amReceiveRequest function. [IN]

Parameter	Description
data	The message data to be sent. [IN]
sndMsgName	The name of a message object for the message being sent. If you specify "" (null), the system default message name (constant AMSD_SND_MSG) is used. [IN]
reasonCode	Holds an AMI Reason Code when the function returns a status indicating an AMI warning or an AMI error occurred. The AMI Reason Code can be used to obtain more information about the cause of the warning or error. See the *MQSeries Application Messaging Interface* manual for a list of AMI Reason Codes and their descriptions. [OUT]

Return codes

The following table describes the status of each return code.

Return code	Description
0 – AMCC_SUCCESS	Function completed successfully.
1 – AMCC_WARNING	A warning was returned from AMI. The *reasonCode* output parameter contains an AMI reason code with further details about the warning.
2 – AMCC_FAILED	An error was returned from AMI. The *reasonCode* output parameter contains an AMI reason code with further details about the error.
Other	A non-AMI error occurred.

You can also use the UniVerse BASIC `STATUS()` function to obtain the return status from the function.

amTerminate function

The `amTerminate` function closes a session.

Syntax

amTerminate(*hSession, policyName, reasonCode*)

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
hSession	The session handle returned by the amInitialize function. [IN/OUT]
policyName	The name of a policy. If you specify "" (null), the system default policy name is used. [IN]
reasonCode	Holds an AMI Reason Code when the function returns a status indicating an AMI warning or an AMI error occurred. The AMI Reason Code can be used to obtain more information about the cause of the warning or error. See the *MQSeries Application Messaging Interface* manual for a list of AMI Reason Codes and their descriptions. [OUT]

Return codes

The following table describes the status of each return code.

Return code	Description
0 – AMCC_SUCCESS	Function completed successfully.
1 – AMCC_WARNING	A warning was returned from AMI. The *reasonCode* output parameter contains an AMI reason code with further details about the warning.
2 – AMCC_FAILED	An error was returned from AMI. The *reasonCode* output parameter contains an AMI reason code with further details about the error.
Other	A non-AMI error occurred.

You can also use the UniVerse BASIC `STATUS()` function to obtain the return status from the function.

analyzeCertificate function

The `analyzeCertificate()` function decodes a certificate and puts plain text into the *result* parameter. The *result* parameter will then contain such information as the subject name, location, institute, issuer, public key, other extensions, and the issuer's signature.

Syntax

analyzeCertificate(*cert, format, result, p12pass*)

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
cert	A string containing the certificate file name.
format	1 - PEM (Base64 encoded) format (SSL_FMT_PEM) 2 - DER (ASN.1 binary) format (SSL_FMT_DER) 3 - PKCS #12 format (SSL_FMT_P12)
result	A dynamic array containing parsed cert data, in ASCII format.
p12pass	Optional. Sets a password on the PKCS #12 file. This parameter should only be included if using a PKCS #12 certificate that has a password. Otherwise the parameter should be omitted.

Return code status

The following table describes the status of each return code.

Return code	Status
0	Success.
1	Failed to open cert file.
2	Invalid format.
3	Unrecognized cert.
4	Other errors.

ANDS function

Use the `ANDS` function to create a dynamic array of the logical AND of corresponding elements of two dynamic arrays.

Each element of the new dynamic array is the logical AND of the corresponding elements of *array1* and *array2*. If an element of one dynamic array has no corresponding element in the other dynamic array, a false (0) is returned for that element.

If you use the subroutine syntax, the resulting dynamic array is returned as *return.array*.

If both corresponding elements of *array1* and *array2* are the null value, null is returned for those elements. If one element is the null value and the other is 0 or an empty string, a false is returned for those elements.

Syntax

ANDS (*array1, array2*)

CALL **-ANDS** (*return.array, array1, array2*)

CALL **!ANDS** (*return.array, array1, array2*)

Example

```
A = 1:@SM:4:@VM:4:@SM:1
B = 1:@SM:1-1:@VM:2
PRINT ANDS(A,B)
```

This is the program output:

```
1S0V1S0
```

ASCII function

Use the ASCII function to convert each character of *expression* from its EBCDIC representation value to its ASCII representation value. If *expression* evaluates to the null value, null is returned.

The ASCII function and the EBCDIC function perform complementary operations.

Syntax

ASCII (*expression*)

Example

```
X = EBCDIC('ABC 123')
 Y = ASCII(X)
 PRINT "EBCDIC", "ASCII", " Y "
 PRINT "------", "-----", "---"
 FOR I = 1 TO LEN (X)
 PRINT SEQ(X[I,1]) , SEQ(Y[I,1]),Y[I,1]
 NEXT I
```

This is the program output:

```
EBCDIC    ASCII      Y
------    -----     ---
193       65        A
194       66        B
195       67        C
64        32
241       49        1
242       50        2
```

```
243        51         3
```

ASIN function

Use the ASIN function to return the trigonometric arc-sine of *expression*. *expression* must be a numeric value. The result is expressed in degrees. If *expression* evaluates to the null value, null is returned. The ASIN function is the inverse of the SIN function.

Syntax

ASIN (*expression*)

Example

```
PRECISION 5
PRINT "ASIN(0.707106781) = ":ASIN(0.707106781):" degrees"
```

This is the program output:

```
ASIN(0.707106781) = 45 degrees
```

ASSIGNED function

Use the ASSIGNED function to determine if *variable* is assigned a value. ASSIGNED returns 1 (true) if *variable* is assigned a value, including common variables and the null value. It returns 0 (false) if *variable* is not assigned a value.

Syntax

ASSIGNED (*variable*)

PICK Flavor

When you run UniVerse in a PICK flavor account, all common variables are initially unassigned. ASSIGNED returns 0 (false) for common variables until the program explicitly assigns them a value.

Example

```
A = "15 STATE STREET"
C = 23
X = ASSIGNED(A)
Y = ASSIGNED(B)
Z = ASSIGNED(C)
PRINT X,Y,Z
```

This is the program output:

```
1          0          1
```

assignment statements

Use assignment statements to assign a value to a variable. The variable can be currently unassigned (that is, one that has not been assigned a value by an assignment statement, READ statements, or any other statement that assigns values to variables) or have an old value that is to be replaced. The assigned value can be a constant or an expression. It can be any data type (that is, numeric, character string, or the null value).

Syntax

```
variable = expression
variable += expression
variable -= expression
variable := expression
```

Use the operators += , -= , and := to alter the value of a variable. The += operator adds the value of *expression* to *variable*. The -= operator subtracts the value of *expression* from *variable*. The := operator concatenates the value of *expression* to the end of *variable*.

Use the system variable @NULL to assign the null value to a variable:

variable = @NULL

Use the system variable @NULL.STR to assign a character string containing only the null value (more accurately, the character used to represent the null value) to a variable:

variable = @NULL.STR

Example

```
EMPL=86
A="22 STAGECOACH LANE"
X='$4,325'
B=999
PRINT "A= ":A,"B= ":B,"EMPL= ":EMPL
B+=1
PRINT "X= ":X,"B= ":B
```

This is the program output:

```
A= 22 STAGECOACH LANE         B= 999    EMPL= 86
X= $4,325 B= 1000
```

ATAN function

Use the `ATAN` function to return the trigonometric arc-tangent of *expression*. *expression* must be a numeric value. The result is expressed in degrees. If *expression* evaluates to the null value, null is returned. The `ATAN` function is the inverse of the `TAN` function.

Syntax

ATAN (*expression*)

Examples

The following example prints the numeric value 135 and the angle, in degrees, that has an arc-tangent of 135:

```
PRINT 135, ATAN(135)
```

This is the program output:

```
 135        89.5756
```

The next example finds what angle has an arc-tangent of 1:

```
X = ATAN(1)
PRINT 1, X
```

This is the program output:

```
1          45
```

AuditLog() function

To reduce unnecessary or excessive logging, the UniVerse BASIC `AuditLog()` function has been added to allow for application-driven audit. For example, instead of enabling system-wide UniVerse BASIC `READ` auditing, which could create a huge number of audit log records, you can choose to have your application call this function at a strategic point to have the action recorded in the system audit file.

Syntax

`AuditLog``(`*`Originator, Action, File, Record, Info, Status, {OldData}, {NewData}`*`)`

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
Originator	The ID of the originator of the event.
Action	The action taken.
File	The file name to audit.
Record	The record ID to audit.
Info	Additional details about this logged action. The content specified in this parameter goes into the Details field of the audit log file.
Status	An integer indicating the status of the logged actions. 0 usually indicates success, and nonzero values indicate errors.
OldData	Optional. The data before the change was made.
NewData	Optional. The data after the change was made.

All parameters are expressions that evaluate to text strings (*Originators*, *Action*, *File*, and *Info*) or dynamic arrays (*Record*, *OldData*, *NewData*), except for *Status*, which must be an integer.

Note: *OldData* and *NewData* are optional parameters. You can omit them if you do not need to store these values. Also, if you do not need File or RecordID, you can supply empty strings instead. For example:

```
OldAddr = Rec.addr
NewAddr = "1234 Main St Cape Town MA 02021"
CALL ChangeAddr("file1", "ID1", Rec, NewAddr)
Status = AuditLog("myapp1", "ChangeAddr", "file1", "ID1","replaced
billing address", 0, OldAddr, NewAddr)
```

AUTHORIZATION statement

Use the AUTHORIZATION statement to specify or change the effective runtime user of a program. After an AUTHORIZATION statement is executed, any SQL security checking acts as if *username* is running the program.

Syntax

AUTHORIZATION "*username*"

username is a valid login name on the machine where the program is run. *username* must be a constant. *username* is compiled as a character string whose user identification (UID) number is looked up in the */etc/passwd* file at run time.

If your program accesses remote files across UVNet, *username* must also be a valid login name on the remote machine.

An AUTHORIZATION statement changes only the user name that is used for SQL security checking while the program is running. It does not change the actual user name, nor does it change the user's effective UID at the operating system level. If a program does not include an AUTHORIZATION statement, it runs with the user name of the user who invokes it.

You can change the effective user of a program as many times as you like. The *username* specified by the most recently executed AUTHORIZATION statement remains in effect for a subsequent EXECUTE statement and PERFORM statement as well as for subroutines.

When a file is opened, the effective user's permissions are stored in the file variable. These permissions apply whenever the file variable is referenced, even if a subsequent AUTHORIZATION statement changes the effective user name.

The effective user name is stored in the system variable @AUTHORIZATION.

A program using the AUTHORIZATION statement must be compiled on the machine where the program is to run. To compile the AUTHORIZATION statement, SQL DBA privilege is required. If the user compiling the program does not have DBA privilege, the program will not be compiled. You cannot run the program on a machine different from the one where it was compiled. If you try, the program terminates with a fatal error message.

Example

```
AUTHORIZATION "susan"
 OPEN "","SUES.FILE" TO FILE.S ELSE PRINT "CAN'T OPEN SUES.FILE"
 AUTHORIZATION "bill"
 OPEN "","BILLS.FILE" TO FILE.B ELSE PRINT "CAN'T OPEN BILLS.FILE"
 FOR ID = 5000 TO 6000
 READ SUE.ID FROM FILE.S, ID THEN PRINT ID ELSE NULL
 READ BILL.ID FROM FILE.B, ID THEN PRINT ID ELSE NULL
```

```
NEXT ID
```

AUXMAP statement

In NLS mode, use the AUXMAP statement to associate an auxiliary device with a terminal.

Syntax

```
AUXMAP { ON | OFF | expression }
```

AUXMAP ON causes a subsequent PRINT statement directed to print channel 0 to use the auxiliary map. If no auxiliary map is defined, the terminal map is used. AUXMAP OFF causes subsequent PRINT statements to use the terminal map. OFF is the default. If *expression* evaluates to true, AUXMAP is turned on. If *expression* evaluates to false, AUXMAP is turned off.

A program can access the map for an auxiliary device only by using the AUXMAP statement. Other statements used for printing to the terminal channel, such as a CRT statement, a PRINT statement, or a INPUTERR statement, use the terminal map.

If NLS is not enabled and you execute the AUXMAP statement, the program displays a run-time error message. For more information, see the *NLS Guide*.

BEGIN CASE statement

Use the BEGIN CASE statement to begin a set of CASE statements.

For details, see CASE statements, on page 83.

BEGIN TRANSACTION statement

Use the BEGIN TRANSACTION statement to indicate the beginning of a transaction.

Syntax

```
BEGIN TRANSACTION [ISOLATION LEVELlevel]
[statements]
```

The ISOLATION LEVEL clause sets the transaction for isolation level for the duration of that transaction. The isolation level reverts to the original value at the end of the transaction.

level is an expression that evaluates to one of the following:

- An integer from 0 through 4
- One of the following keywords

Integer	Keyword	Effect on this transaction
0	NO.ISOLATION	Prevents lost updates. Lost updates are prevented if the ISOMODE configurable parameter is set to 1 or 2.
1	READ.UNCOMMITTED	Prevents lost updates.
2	READ.COMMITTED	Prevents lost updates and dirty reads.
3	REPEATABLE.READ	Prevents lost updates, dirty reads, and nonrepeatable reads.
4	SERIALIZABLE	Prevents lost updates, dirty reads, nonrepeatable reads, and phantom writes.

Examples

The following examples both start a transaction at isolation level 3:

```
BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE.READ
BEGIN TRANSACTION ISOLATION LEVEL 3
```

BITAND function

Use the `BITAND` function to perform the bitwise AND comparison of two integers specified by numeric expressions. The bitwise AND operation compares two integers bit by bit. It returns a bit of 1 if both bits are 1; otherwise it returns a bit of 0.

Syntax

BITAND (*expression1, expression2*)

If either *expression1* or *expression2* evaluates to the null value, null is returned.

Noninteger values are truncated before the operation is performed.

The `BITAND` operation is performed on a 32-bit twos-complement word.

Note: Differences in hardware architecture can make the use of the high-order bit nonportable.

Example

```
PRINT BITAND(6,12)
* The binary value of 6    =   0110
* The binary value of 12 =     1100
```

This results in 0100, and the following output is displayed:

```
4
```

BITNOT function

Use the BITNOT function to return the bitwise negation of an integer specified by any numeric expression.

Syntax

BITNOT (*expression* [,*bit#*])

bit# is an expression that evaluates to the number of the bit to invert. If *bit#* is unspecified, BITNOT inverts each bit. It changes each bit of 1 to a bit of 0 and each bit of 0 to a bit of 1. This is equivalent to returning a value equal to the following:

(-*expression*)-1

If *expression* evaluates to the null value, null is returned. If *bit#* evaluates to the null value, the BITNOT function fails and the program terminates with a run-time error message.

Noninteger values are truncated before the operation is performed.

The BITNOT operation is performed on a 32-bit twos-complement word.

Note: Differences in hardware architecture can make the use of the high-order bit nonportable.

Example

```
PRINT BITNOT(6),BITNOT(15,0),BITNOT(15,1),BITNOT(15,2)
```

This is the program output:

```
 -7        14        13        11
```

BITOR function

Use the BITOR function to perform the bitwise OR comparison of two integers specified by numeric expressions. The bitwise OR operation compares two integers bit by bit. It returns the bit 1 if the bit in either or both numbers is 1; otherwise it returns the bit 0.

Syntax

BITOR (*expression1, expression2*)

If either *expression1* or *expression2* evaluates to the null value, null is returned.

Noninteger values are truncated before the operation is performed.

The BITOR operation is performed on a 32-bit twos-complement word.

Note: Differences in hardware architecture can make the use of the high-order bit nonportable.

Example

```
PRINT BITOR(6,12)
* Binary value of 6    = 0110
* Binary value of 12 = 1100
```

This results in 1110, and the following output is displayed:

```
14
```

BITRESET function

Use the BITRESET function to reset to 0 the bit number of the integer specified by *expression*. Bits are counted from right to left. The number of the rightmost bit is 0. If the bit is 0, it is left unchanged.

Syntax

BITRESET (*expression, bit#*)

If *expression* evaluates to the null value, null is returned. If *bit#* evaluates to the null value, the BITRESET function fails and the program terminates with a run-time error message.

Noninteger values are truncated before the operation is performed.

Example

```
PRINT BITRESET(29,0),BITRESET(29,3)
* The binary value of 29 = 11101
* The binary value of 28 = 11100
* The binary value of 21 = 10101

PRINT BITRESET(2,1),BITRESET(2,0)
* The binary value of 2 = 10
* The binary value of 0 =  0
```

This is the program output:

```
28          21
0           2
```

BITSET function

Use the BITSET function to set to 1 the bit number of the integer specified by *expression*. The number of the rightmost bit is 0. If the bit is 1, it is left unchanged.

Syntax

BITSET (*expression, bit#*)

If *expression* evaluates to the null value, null is returned. If *bit#* evaluates to the null value, the BITSET function fails and the program terminates with a run-time error message.

Noninteger values are truncated before the operation is performed.

Example

```
PRINT BITSET(20,0),BITSET(20,3)
* The binary value of 20 = 10100
* The binary value of 21 = 10101
* The binary value of 28 = 11100
```

```
PRINT BITSET(2,0),BITSET(2,1)
* The binary value of 2 = 10
* The binary value of 3 = 11
```

This is the program output:

```
21          28
3           2
```

BITTEST function

Use the BITTEST function to test the bit number of the integer specified by *expression*. The function returns 1 if the bit is set; it returns 0 if it is not. Bits are counted from right to left. The number of the rightmost bit is 0.

Syntax

BITTEST (*expression, bit#*)

If *expression* evaluates to the null value, null is returned. If *bit#* evaluates to null, the BITTEST function fails and the program terminates with a runtime error message.

Noninteger values are truncated before the operation is performed.

Example

```
PRINT BITTEST(11,0),BITTEST(11,1),BITTEST(11,2),BITTEST(11,3)
* The binary value of 11 = 1011
```

This is the program output:

```
1           1           0           1
```

BITXOR function

Use the BITXOR function to perform the bitwise XOR comparison of two integers specified by numeric expressions. The bitwise XOR operation compares two integers bit by bit. It returns a bit 1 if only one of the two bits is 1; otherwise it returns a bit 0.

Syntax

BITXOR (*expression1, expression2*)

If either *expression1* or *expression2* evaluates to the null value, null is returned.

Noninteger values are truncated before the operation is performed.

The BITXOR operation is performed on a 32-bit twos-complement word.

Note: Differences in hardware architecture can make the use of the high-order bit nonportable.

Example

```
PRINT BITXOR(6,12)
* Binary value of 6    = 0110
* Binary value of 12 = 1100
```

This results in 1010, and the following output is displayed:

```
10
```

BREAK statement

Use the BREAK statement to enable or disable the Intr, Quit, and Susp keys on the keyboard.

Syntax

BREAK [KEY] { ON | OFF | *expression* }

When the BREAK ON statement is in effect, pressing Intr, Quit, or Susp causes operations to pause.

When the BREAK OFF statement is in effect, pressing Intr, Quit, or Susp has no effect. This prevents a break in execution of programs that you do not want interrupted.

When *expression* is used with the BREAK statement, the value of *expression* determines the status of the Intr, Quit, and Susp keys. If *expression* evaluates to false (0, an empty string, or the null value), the Intr, Quit, and Susp keys are disabled. If *expression* evaluates to true (not 0, an empty string, or the null value), the Intr, Quit, and Susp keys are enabled.

A counter is maintained for the BREAK statement. It counts the number of executed BREAK ON and BREAK OFF commands. When program control branches to a subroutine, the value of the counter is maintained; it is not set back to 0. For each BREAK ON statement executed, the counter decrements by 1; for each BREAK OFF statement executed, the counter increments by 1. The counter cannot go below 0. The Intr, Quit, and Susp keys are enabled only when the value of the counter is 0. The following example illustrates the point:

Statement from	Command	Counter	Key status
—	—	0	ON
Main program	BREAK OFF	+1	OFF
Subroutine	BREAK OFF	+2	OFF
Subroutine	BREAK ON	+1	OFF
Main program	BREAK ON	0	ON

Examples

The following example increases the counter by 1:

```
BREAK KEY OFF
```

The following example decreases the counter by 1:

```
BREAK KEY ON
```

The following example disables the Intr, Quit, and Susp keys if QTY is false, 0, an empty string, or the null value; it enables them if QTY is true, not 0, not an empty string, or not the null value:

```
BREAK QTY ;*
```

BSCAN statement

Use the BSCAN statement to scan the leaf nodes of a B-tree file (type 25) or of a secondary index. The record ID returned by the current scan operation is assigned to *ID.variable*. If you specify *rec.variable*, the contents of the record whose ID is *ID.variable* is assigned to it.

Syntax

```
BSCAN ID.variable [, rec.variable] [FROM file.variable [, record]]
[USING indexname] [RESET] [BY seq] {THEN statements [ELSE statements] |
ELSE statements}
```

file.variable specifies an open file. If *file.variable* is not specified, the default file is assumed (for more information on default files, see the OPEN statement, on page 276). If the file is neither accessible nor open, the program terminates with a runtime error message.

record is an expression that evaluates to a record ID of a record in the B-tree file. If the USING clause is used, *record* is a value in the specified index. *record* specifies the relative starting position of the scan.

record need not exactly match an existing record ID or value. If it does not, the scan finds the next or previous record ID or value, depending on whether the scan is in ascending or descending order. For example, depending on how precisely you want to specify the starting point at or near the record ID or value SMITH, *record* can evaluate to SMITH, SMIT, SMI, SM, or S.

If you do not specify *record*, the scan starts at the leftmost slot of the leftmost leaf, or the rightmost slot of the rightmost leaf, depending on the value of the *seq* expression. The scan then moves in the direction specified in the BY clause.

indexname is an expression that evaluates to the name of a secondary index associated with the file.

RESET resets the internal B-tree scan pointer. If the scanning order is ascending, the pointer is set to the leftmost slot of the leftmost leaf; if the order is descending, the pointer is set to the rightmost slot of the rightmost leaf. If you do not specify *seq*, the scan is done in ascending order. If you specify *record* in the FROM clause, RESET is ignored.

seq is an expression that evaluates to A or D; it specifies the direction of the scan. "A", the default, specifies ascending order. "D" specifies descending order.

If the BSCAN statement finds a valid record ID, or a record ID and its associated data, the THEN statements are executed; the ELSE statements are ignored. If the scan does not find a valid record ID, or if some other error occurs, any THEN statements are ignored, and the ELSE statements are executed.

Any file updates executed in a transaction (that is, between a BEGIN TRANSACTION statement and a COMMIT statement) are not accessible to the BSCAN statement until after the COMMIT statement has been executed.

The STATUS function returns the following values after the BSCAN statement is executed:

Value	Description
0	The scan proceeded beyond the leftmost or rightmost leaf node. *ID.variable* and *rec.variable* are set to empty strings.
1	The scan returned an existing record ID, or a record ID that matches the record ID specified by *record*.
2	The scan returned a record ID that does not match *record*. *ID.variable* is either the next or the previous record ID in the B-tree, depending on the direction of the scan.

Value	Description
3	The file is not a B-tree (type 25) file, or, if the USING clause is used, the file has no active secondary indexes.
4	*indexname* does not exist.
5	*seq* does not evaluate to A or D.
6	The index specified by *indexname* needs to be built, or is currently being built concurrently.
10	An internal error was detected.

If NLS is enabled, the BSCAN statement retrieves record IDs in the order determined by the active collation locale; otherwise, BSCAN uses the default order, which is simple byte ordering that uses the standard binary value for characters; the Collate convention as specified in the NLS.LC.COLLATE file for the current locale is ignored. For more information about collation, see the *NLS Guide*.

Example

The following example shows how you might indicate that the ELSE statements were executed because the contents of the leaf nodes were exhausted:

```
BSCAN ID,REC FROM FILE,MATCH USING "PRODUCT" BY "A" THEN
PRINT ID,REC
END ELSE
ERR = STATUS()
BEGIN CASE
CASE ERR = 0
PRINT "Exhausted leaf node contents."
CASE ERR = 3
PRINT "No active indices, or file is not type 25."
CASE ERR = 4
PRINT "Index name does not exist."
CASE ERR = 5
PRINT "Invalid BY clause value."
CASE ERR = 6
PRINT "Index must be built."
CASE ERR = 10
PRINT "Internal error detected."
END CASE
GOTO EXIT.PROGRAM:
END
```

BYTE function

In NLS mode, use the `BYTE` function to generate a byte from the numeric value of *expression*. `BYTE` returns a string containing a single byte.

Syntax

BYTE (*expression*)

If *expression* evaluates to a value in the range 0 to 255, a single-byte character is returned. If *expression* evaluates to a value in the range 0x80 to 0xF7, a byte that is part of a multibyte character is returned.

If NLS is not enabled, `BYTE` works like the CHAR function, on page 87. For more information, see the *NLS Guide*.

Example

When NLS is enabled, the BYTE and CHAR functions return the following:

Function	Value
BYTE(32)	Returns a string containing a single space.
CHAR(32)	Returns a string containing a single space.
BYTE(230)	Returns a string containing the single byte 0xe6.
CHAR(230)	Returns a string containing the multibyte characters æ (small ligature Æ).

BYTELEN function

In NLS mode, use the BYTELEN function to generate the number of bytes contained in the ASCII string value in *expression*.

Syntax

BYTELEN (*expression*)

The bytes in *expression* are counted, and the count is returned. If *expression* evaluates to the null value, null is returned.

If NLS is not enabled, BYTELEN works like the LEN function, on page 235. For more information, see the *NLS Guide*.

BYTETYPE function

In NLS mode, use the BYTETYPE function to determine the function of a byte in *value*.

Syntax

BYTETYPE (*value*)

If *value* is from 0 to 255, the BYTETYPE function returns a number that corresponds to the following:

Return value	Description
–1	*value* is out of bounds
0	Trailing byte of a 2-, 3-, or > 3-byte character
1	Single-byte character
2	Leading byte of a 2-byte character
3	Leading byte of a 3-byte character
4	Reserved for the leading byte of a 4-byte character
5	System delimiter

If *value* evaluates to the null value, null is returned.

BYTETYPE behaves the same whether NLS is enabled or not. For more information, see the *NLS Guide*.

BYTEVAL function

In NLS mode, use the BYTEVAL function to examine the bytes contained in the internal string value of *expression*. The BYTEVAL function returns a number from 0 through 255 as the byte value of *n* in *expression*. If you omit *n*, 1 is assumed.

Syntax

```
BYTEVAL (expression [, n] )
```

If an error occurs, the BYTEVAL function returns –1 if *expression* is the empty string or has fewer than *n* bytes, or if *n* is less than 1. If *expression* evaluates to the null value, BYTEVAL returns null.

BYTEVAL behaves the same whether NLS is enabled or not. For more information, see the *NLS Guide*.

CALL statement

Use the CALL statement to transfer program control from the calling program to an external subroutine or program that has been compiled and cataloged.

Syntax

```
CALL name [([MAT] argument [, [MAT] argument …])]

 variable = 'name'
 CALL @variable [([MAT] argument [, [MAT] argument …])]
```

Locally cataloged subroutines can be called directly. Specify *name* using the exact name under which it was cataloged. For more details, see the CATALOG command.

External subroutines can be called directly or indirectly. To call a subroutine indirectly, the name under which the subroutine is cataloged must be assigned to a variable or to an element of an array. This variable name or array element specifier, prefixed with an at sign (@), is used as the operand of the CALL statement.

The first time a CALL is executed, the system searches for the subroutine in a cataloged library and changes a variable that contains the subroutine name to contain its location information instead. This procedure eliminates the need to search the catalog again if the same subroutine is called later in the program. For indirect calls, the variable specified in the CALL as the @variable is used; for direct calls, an internal variable is used. With the indirect method, it is best to assign the subroutine name to the variable only once in the program, not every time the indirect CALL statement is used.

arguments are variables, arrays, array variables, expressions, or constants that represent actual values. You can pass one or more arguments from the calling program to a subroutine. The number of arguments passed in a CALL statement must equal the number of arguments specified in the SUBROUTINE statement, on page 392 that identifies the subroutine. If multiple arguments are passed, they must be separated by commas. If an argument requires more than one physical line, use a comma at the end of the line to indicate that the list continues.

If *argument* is an array, it must be preceded by the MAT keyword, and the array should be named and dimensioned in both the calling program and the subroutine before using this statement. If the array is not dimensioned in the subroutine, it must be declared using the MAT keyword in the SUBROUTINE statement. Other arguments can be passed at the same time regardless of the size of the array.

The actual values of *arguments* are not passed to the subroutine. Instead, a pointer to the location of each argument is passed. Passing a pointer instead of the values is more efficient when many values

need to be passed to the subroutine. This method of passing arguments is called passing *by reference*; passing actual values is called passing *by value*.

All scalar and matrix variables are passed to subroutines by reference. If you want to pass variables by value, enclose them in parentheses. When data is passed by value, the contents of the variable in the main program do not change as a result of manipulations to the data in the subroutine. When data is passed by reference, the memory location of the variable is changed by manipulations in both the main program and the subroutines. Constants are passed to subroutines by value.

When an array is passed to an external subroutine as an argument in a CALL statement, any dimensions assigned to the array in the subroutine are ignored. The dimensions of the original array as it exists in the calling program are maintained. Therefore, it is a common and acceptable practice to dimension the array in the subroutine with subscripts or indices of one. For example, you could dimension the arrays in the subroutine as follows:

```
DIM A (1), B (1, 1), C (1, 1)
```

When the corresponding array arguments are passed from the calling program to the subroutine at run time, arrays A, B, and C inherit the dimensions of the arrays in the calling program. The indices in the DIMENSION statement are ignored.

A better way to declare array arguments in a subroutine is to use the MAT keyword of the SUBROUTINE statement in the first line of the subroutine. The following example tells the subroutine to expect the three arrays A, B, and C:

```
SUBROUTINE X(MAT A, MAT B, MAT C)
```

When a RETURN statement is encountered in the subroutine, or when execution of the subroutine ends without encountering a RETURN statement, control returns to the statement following the CALL statement in the calling program. For more details, see the RETURN statement, on page 327.

Examples

The following example calls the local subroutine SUB. It has no arguments.

```
CALL SUB
```

The following example calls the local subroutine QTY.ROUTINE with three arguments:

```
CALL QTY.ROUTINE(X,Y,Z)
```

The following example calls the subroutine cataloged as *PROGRAM.1 with six arguments. The argument list can be expressed on more than one line.

```
AAA="*PROGRAM.1"
 CALL @AAA(QTY,SLS,ORDER,ANS,FILE.O,SEQ)
```

The following example calls the subroutine *MA with three arguments. Its index and three arguments are passed.

```
STATE.TAX(1,2)='*MA'
 CALL @STATE.TAX(1,2)(EMP.NO,GROSS,NET)
```

The following example calls the subroutine cataloged as *SUB and two matrices are passed to two subroutine matrices. A third, scalar, argument is also passed.

```
GET.VALUE="*SUB"
 DIM QTY(10)
 DIM PRICE(10)
 CALL @GET.VALUE(MAT QTY,MAT PRICE,COST )
```

The following example shows the SUBROUTINE statement in the subroutine SUB that is called by the preceding example. The arrays Q and P need not be dimensioned in the subroutine.

```
SUBROUTINE SUB(MAT Q,MAT P,C )
```

CASE statements

Use the CASE statement to alter the sequence of instruction execution based on the value of one or more expressions. If *expression* in the first CASE statement is true, the following statements up to the next CASE statement are executed. Execution continues with the statement following the END CASE statement.

Syntax

```
BEGIN CASE
CASE expression
statements
[CASE expression
statements
 .
 .
.]
END CASE
```

If the expression in a CASE statement is false, execution continues by testing the expression in the next CASE statement. If it is true, the statements following the CASE statement up to the next CASE or END CASE statement are executed. Execution continues with the statement following the END CASE statement.

If more than one CASE statement contains a true expression, only the statements following the first such CASE statement are executed. If no CASE statements are true, none of the statements between the BEGIN CASE and END CASE statements are executed.

If an expression evaluates to the null value, the CASE statement is considered false.

Use the ISNULL function with the CASE statement when you want to test whether the value of a variable is the null value. This is the only way to test for the null value since null cannot be equal to any value, including itself. The syntax is:

```
CASE ISNULL (expression)
```

Use an expression of the constant "1" to specify a default CASE to be executed if none of the other CASE expressions evaluate to true.

Examples

In the following example NUMBER is equal to 3. CASE 1 is always true, therefore control is transferred to subroutine 30. Once the subroutine RETURN is executed, control proceeds to the statement following the END CASE statement.

```
NUMBER=3
BEGIN CASE
CASE NUMBER=1
GOTO 10
CASE 1
GOSUB 30
CASE NUMBER<3
GOSUB 20
```

```
END CASE
PRINT 'STATEMENT FOLLOWING END CASE'
GOTO 50
10*
PRINT 'LABEL 10'
STOP
20*
PRINT 'LABEL 20'
RETURN
30*
PRINT 'LABEL 30'
RETURN
50*
```

This is the program output:

```
LABEL 30
STATEMENT FOLLOWING END CASE
```

In the following example, control proceeds to the statement following the END CASE because 'NAME' does not meet any of the conditions:

```
NAME="MICHAEL"
BEGIN CASE
CASE NAME[1,2]='DA'
PRINT NAME
GOTO 10
CASE NAME[1,2]='RI'
PRINT NAME
GOSUB 20
CASE NAME[1,2]='BA'
PRINT NAME
GOSUB 30
END CASE
PRINT 'NO MATCH'
STOP
```

This is the program output:

```
NO MATCH
```

CATS function

Use the CATS function to create a dynamic array of the element-by-element concatenation of two dynamic arrays.

Syntax

CATS (*array1*, *array2*)

CALL **-CATS** (*return.array*, *array1*, *array2*)

CALL **!CATS** (*return.array*, *array1*, *array2*)

Each element of *array1* is concatenated with the corresponding element of *array2*. The result is returned in the corresponding element of a new dynamic array. If an element of one dynamic array has no corresponding element in the other dynamic array, the existing element is returned. If an element of one dynamic array is the null value, null is returned for the concatenation of the corresponding elements.

If you use the subroutine syntax, the resulting dynamic array is returned as *return.array*.

Example

```
A="A":@VM:"B":@SM:"C"
B="D":@SM:"E":@VM:"F"
PRINT CATS(A,B)
```

This is the program output:

```
ADSEVBFSC
```

CENTURY.PIVOT function

Use the `CENTURY.PIVOT` function to override the system-wide century pivot year defined in the `uvconfig` file.

Syntax

`CENTURY.PIVOT` (*`year | nn`*)

In UniVerse, when you enter as input a year in two-digit format (for example, 99 or 01), UniVerse by default assumes the following:

- Years entered in the range 30 through 99 stand for 1930 through 1999
- Years entered in the range 00 through 29 stand for 2000 through 2029

Administrators can change these default ranges in three ways:

- Setting or changing the CENTURYPIVOT configurable parameter in the `uvconfig` file (for information about configurable parameters, see *Administering UniVerse*).
- Using the `CENTURY.PIVOT` UniVerse command (see *User Reference Guide*).
- Using the `CENTURY.PIVOT` function

The CENTURYPIVOT configurable parameter sets the system-wide century pivot year for UniVerse. You can use the `CENTURY.PIVOT` command to override the century pivot year for the current session.

You can set the century pivot year in two ways:

Static century pivot year

If you specify the century pivot year with four digits, the first two digits specify the century, and the last two digits specify the pivot year.

For example, if you specify *year* as 1940, two-digit years specified in the range of 40 through 99 stand for 1940 though 1999, and two-digit years specified in the range of 00 through 29 stand for 2000 through 2039. These ranges remain fixed until you explicitly change them.

Sliding century pivot year

If you enter the century pivot year as a two-digit code (*nn*), the century pivot year changes relative to the current year. The formula for determining the century pivot year is as follows:

current.year – (100 – *nn*)

For example, if the current year is 2000 and *nn* is 05, the century pivot year is 1905. This means that two-digit years specified in the range of 05 through 99 stand for 1905 through 1999, and two-digit years specified in the range of 00 through 04 stand for 2000 through 2004.

If the current year is 2005 and *nn* is 05, the century pivot year is 1910. Two-digit years specified in the range of 10 through 99 stand for 1910 through 1999, and two-digit years specified in the range of 00 through 09 stand for 2000 through 2009.

If the current year is 2001 and *nn* is 30, the century pivot year is 1931. Two-digit years specified in the range of 31 through 99 stand for 1931 through 1999, and two-digit years specified in the range of 00 through 30 stand for 2000 through 2030.

CHAIN statement

Use the CHAIN statement to terminate execution of a UniVerse BASIC program and to execute the value of command. command is an expression that evaluates to any valid UniVerse command. If command evaluates to the null value, the CHAIN statement fails and the program terminates with a runtime error message.

Local variables belonging to the current program are lost when you chain from one program to another. Named and unnamed common variables are retained.

CHAIN differs from the EXECUTE statement or PERFORM statement in that CHAIN does not return control to the calling program. If a program chains to a proc, any nested calling procs are removed.

Syntax

CHAIN *command*

PICK, IN2, and REALITY flavors

Unnamed common variables are lost when a chained program is invoked in a PICK, IN2, or REALITY flavor account. If you want to save the values of variables in unnamed common, use the KEEP.COMMON keyword to the `RUN` command at execution.

Example

The following program clears the screen, initializes the common area, and then runs the main application:

```
PRINT @(-1)
PRINT "INITIALIZING COMMON, PLEASE WAIT"
GOSUB INIT.COMMON
CHAIN "RUN BP APP.MAIN KEEP.COMMON"
```

CHANGE function

Use the `CHANGE` function to replace a substring in *expression* with another substring. If you do not specify *occurrence*, each occurrence of the substring is replaced.

Syntax

CHANGE (*expression, substring, replacement* [,*occurrence* [,*begin*]])

occurrence specifies the number of occurrences of *substring* to replace. To change all occurrences, specify *occurrence* as a number less than 1.

begin specifies the first occurrence to replace. If *begin* is omitted or less than 1, it defaults to 1.

If *substring* is an empty string, the value of *expression* is returned. If *replacement* is an empty string, all occurrences of *substring* are removed.

If *expression* evaluates to the null value, null is returned. If *substring*, *replacement*, *occurrence*, or *begin* evaluates to the null value, the CHANGE function fails and the program terminates with a run-time error message.

The CHANGE function behaves like the EREPLACE function except when *substring* evaluates to an empty string.

Example

```
A = "AAABBBCCCDDDBBB"
PRINT CHANGE (A,"BBB","ZZZ")
PRINT CHANGE (A,"","ZZZ")
PRINT CHANGE (A,"BBB","")
```

This is the program output:

```
AAAZZZCCCDDDZZZ
AAABBBCCCDDDBBB
AAACCCDDD
```

CHAR function

Use the CHAR function to generate an ASCII character from the numeric value of *expression*.

If *expression* evaluates to the null value, null is returned. If *expression* evaluates to 128, CHAR(128) is returned, not the null value. CHAR(128) is the equivalent of the system variable @NULL.STR.

The CHAR function is the inverse of the SEQ function.

If NLS mode is enabled, and if *expression* evaluates to a number from 129 through 247, the CHAR function generates Unicode characters from x0081 through x00F7. These values correspond to the equivalent ISO 8859-1 (Latin 1) multibyte characters. The evaluation of numbers from 0 through 127, 128, and 248 through 255 remains the same whether NLS is enabled or not.

The UNICHAR function is the recommended method for generating Unicode characters. For more information, see the *NLS Guide*.

Note: In order to run programs using the CHAR function in NLS mode, you must first recompile them in NLS mode.

Syntax

CHAR (*expression*)

Example

```
X = CHAR(38)
Y = CHAR(32)
PRINT X:Y:X
```

CHAR(38) is an ampersand (&). CHAR(32) is a space. This is the program output:

```
& &
```

CHARS function

Use the CHARS function to generate a dynamic array of ASCII characters from the decimal numeric value of each element of *dynamic.array*.

If you use the subroutine syntax, the resulting dynamic array is returned as *return.array*.

If any element in the dynamic array is the null value, null is returned for that element. If any element in the dynamic array evaluates to 128, CHAR(128) is returned, not the null value. CHAR(128) is the equivalent of the system variable @NULL.STR.

If NLS mode is enabled, and if any element in the dynamic array evaluates to a number from 129 through 247, the CHARS function generates Unicode characters from x0081 through x00F7. These values correspond to the equivalent ISO 8859-1 (Latin 1) multibyte characters. The evaluation of numbers from 0 through 127, 128, and 248 through 255 remains the same whether NLS is enabled or not.

The UNICHARS function is the recommended method for generating a dynamic array of Unicode characters. For more information, see the *NLS Guide*.

Syntax

```
CHARS (dynamic.array)
CALL -CHARS (return.array, dynamic.array)
CALL !CHARS (return.array, dynamic.array)
```

Example

```
X = CHARS(38:@VM:32:@VM:38)
PRINT X
```

The dynamic array X comprises three elements: CHAR(38) (an ampersand (&)), CHAR(32) (a space), and another CHAR(38). The program prints a dynamic array of these elements separated by value marks:

```
&V V&
```

CHECKSUM function

Use the CHECKSUM function to return a cyclical redundancy code (a checksum value).

If *string* is the null value, null is returned.

Syntax

```
CHECKSUM (string)
```

Example

```
A = "THIS IS A RECORD TO BE SENT VIA SOME PROTOCOL"
REC = A:@FM:CHECKSUM(A)
PRINT REC
```

This is the program output:

```
THIS IS A RECORD TO BE SENT VIA SOME PROTOCOLF30949
```

CLEAR statement

Use the CLEAR statement at the beginning of a program to set all assigned and unassigned values of variables outside of the common area of the program to 0. This procedure avoids run-time errors for unassigned variables. If you use the CLEAR statement later in the program, any values assigned to noncommon variables (including arrays) are lost.

Use the COMMON option to reset the values of all the variables in the unnamed common area to 0. Variables outside the common area or in the named common area are unaffected.

Syntax

CLEAR [COMMON]

Example

```
A=100
PRINT "The value of A before the CLEAR statement:"
PRINT A
CLEAR
PRINT "The value of A after the CLEAR statement:"
PRINT A
PRINT
*
COMMON B,C,D
D="HI"
PRINT "The values of B, C, and D"
PRINT B,C,D
CLEAR COMMON
PRINT B,C,D
```

This is the program output:

```
The value of A before the CLEAR statement: 100
The value of A after the CLEAR statement:          0
The values of B, C, and D
0                          0                               HI
0                          0                               0
```

CLEARCOMMON

The UniVerse BASIC CLEARCOMMON command sets all variables in a named common area to zero. If you do not specify *common.label*, CLEARCOMMON sets all variables specified in the unnamed common area to zero.

Syntax

CLEARCOMMON [/*common.label*/]

Examples

In the following example, the program statement sets to zero all variables named in COM_1:

```
CLEARCOMMON /COM_1/
```

In the next example, the program statement sets to zero all variables held in common areas if the variable INITIALIZE.COMMON is true:

```
IF INITIALIZE.COMMON THEN CLEAR COMMON
```

CLEARDATA statement

Use the CLEARDATA statement to flush all data that has been loaded in the input stack by the DATA statement. No expressions or spaces are allowed with this statement. Use the CLEARDATA statement when an error is detected, to prevent data placed in the input stack from being used incorrectly.

Syntax

CLEARDATA

Example

The following program is invoked from a paragraph. A list of file names and record IDs is passed to it from the paragraph with DATA statements. If a file cannot be opened, the CLEARDATA statement clears the data stack since the DATA statements would no longer be valid to the program.

```
TEN:
 INPUT FILENAME
 IF FILENAME="END" THEN STOP
 OPEN FILENAME TO FILE ELSE
 PRINT "CAN'T OPEN FILE ":FILENAME
 PRINT "PLEASE ENTER NEW FILENAME "
 CLEARDATA
 GOTO TEN:
 END
 TWENTY:
 INPUT RECORD
 READ REC FROM FILE,RECORD ELSE GOTO TEN:
 PRINT REC<1>
 GOTO TEN:
TEST.FILE.
 0 records listed.
```

CLEARFILE statement

Use the CLEARFILE statement to delete all records in an open dictionary or data file. You cannot use this statement to delete the file itself. Each file to be cleared must be specified in a separate CLEARFILE statement.

Syntax

CLEARFILE [*file.variable*] [ON ERROR *statements*] [LOCKED *statements*]

file.variable specifies an open file. If *file.variable* is not specified, the default file is assumed (for more information on default files, see the OPEN statement).

The CLEARFILE statement fails and the program terminates with a runtime error message if:

- The file is neither accessible nor open.
- *file.variable* evaluates to the null value.

- A distributed file contains a part file that cannot be accessed, but the CLEARFILE statement clears those part files still available.
- A transaction is active. That is, you cannot execute this statement between a BEGIN TRANSACTION statement (or TRANSACTION START statement and the COMMIT statement (or TRANSACTION START statement) or ROLLBACK statement that ends the transaction.

The ON ERROR clause

The ON ERROR clause is optional in the CLEARFILE statement. The ON ERROR clause lets you specify an alternative for program termination when a fatal error is encountered during processing of the CLEARFILE statement.

If a fatal error occurs and the ON ERROR clause was not specified or was ignored, the following occurs:

- An error message appears.
- The current program terminates.
- Processing continues with the next statement of the previous execution environment, or the program returns to the UniVerse prompt.

If the ON ERROR clause is used, the value returned by the `STATUS` function is the error number. If a CLEARFILE statement is used when any portion of a file is locked, the program waits until the file is released. The ON ERROR clause is not supported if the CLEARFILE statement is within a transaction.

The LOCKED clause

The LOCKED clause is optional, but recommended.

The LOCKED clause handles a condition caused by a conflicting lock (set by another user) that prevents the CLEARFILE statement from processing. The LOCKED clause is executed if one of the following conflicting locks exists:

- Exclusive file lock
- Intent file lock
- Shared file lock
- Update record lock
- Shared record lock

If the CLEARFILE statement does not include a LOCKED clause, and a conflicting lock exists, the program pauses until the lock is released.

If a LOCKED clause is used, the value returned by the STATUS function is the terminal number of the user who owns the conflicting lock.

Example

```
OPEN "","TEST.FILE" ELSE PRINT "NOT OPEN"
EXECUTE "LIST TEST.FILE"
CLEARFILE
CHAIN "LIST TEST.FILE"
```

This is the program output:

```
LIST TEST.FILE 11:37:45am   03-22-94 PAGE          1
TEST.FILE
ONE
TWO
THREE
```

```
3 records listed.
LIST TEST.FILE 11:37:46am  03-22-94 PAGE          1
TEST.FILE.
0 records listed.
```

CLEARPROMPTS statement

Use the CLEARPROMPTS statement to clear the value of the inline prompt. Once a value is entered for an in-line prompt, the prompt continues to have that value until a CLEARPROMPTS statement is executed, unless the in-line prompt control option A is specified. CLEARPROMPTS clears all values that have been entered for inline prompts.

Syntax

CLEARPROMPTS

CALL !**CLEAR.PROMPTS**

For information about in-line prompts, see the ILPROMPT function, on page 209.

CLEARSELECT statement

Use the CLEARSELECT statement to clear an active select list. This statement is normally used when one or more select lists have been generated but are no longer needed. Clearing select lists prevents remaining select list entries from being used erroneously.

Syntax

CLEARSELECT [ALL | *list.number*]

Use the keyword ALL to clear all active select lists. Use *list.number* to specify a numbered select list to clear. *list.number* must be a numeric value from 0 through 10. If neither ALL nor *list.number* is specified, select list 0 is cleared.

If *list.number* evaluates to the null value, the CLEARSELECT statement fails and the program terminates with a run-time error message.

PICK, REALITY, and IN2 flavors

PICK, REALITY, and IN2 flavor accounts store select lists in list variables instead of numbered select lists. In those accounts, and in programs that use the VAR.SELECT option of the $OPTIONS statement, the syntax of CLEARSELECT is:

CLEARSELECT [ALL | *list.variable*]

Example

The following program illustrates the use of CLEARSELECT to clear a partially used select list. The report is designed to display the first 40-odd hours of lessons. A CLEARSELECT is used so that all the selected records are not printed. Once the select list is cleared, the READNEXT statement ELSE clause is executed.

```
OPEN 'SUN.SPORT' TO FILE ELSE STOP "CAN'T OPEN FILE"
HOURS=0
*
```

```
EXECUTE 'SSELECT SUN.SPORT BY START BY INSTRUCTOR'
*
START:
READNEXT KEY ELSE
PRINT 'FIRST WEEK', HOURS
STOP
END
READ MEMBER FROM FILE,KEY ELSE GOTO START:
HOURS=HOURS+MEMBER<4>
PRINT MEMBER<1>,MEMBER<4>
IF HOURS>40 THEN
******
CLEARSELECT
******
GOTO START:
END
GOTO START:
END
```

This is the program output:

```
14 records selected to Select List #0
4309               1
6100               4
3452               3
6783               12
5390               9
4439               4
6203               14
FIRST WEEK                              47
```

CLOSE statement

Use the CLOSE statement after opening and processing a file. Any file locks or record locks are released.

Syntax

CLOSE [*file.variable*] [ON ERROR *statements*]

file.variable specifies an open file. If *file.variable* is not specified, the default file is assumed. If the file is neither accessible nor open, or if *file.variable* evaluates to the null value, the CLOSE statement fails and the program terminates with a run-time error message.

The ON ERROR clause

The ON ERROR clause is optional in the CLOSE statement. The ON ERROR clause lets you specify an alternative for program termination when a fatal error is encountered during processing of the CLOSE statement.

If a fatal error occurs, and the ON ERROR clause was not specified, or was ignored (as in the case of an active transaction), the following occurs:

- An error message appears.
- Any uncommitted transactions begun within the current execution environment roll back.

- The current program terminates.
- Processing continues with the next statement of the previous execution environment, or the program returns to the UniVerse prompt.

A fatal error can occur if any of the following occur:

- A file is not open.
- *file.variable* is the null value.
- A distributed file contains a part file that cannot be accessed.

If the ON ERROR clause is used, the value returned by the STATUS function is the error number.

Example

```
CLEAR
 OPEN '','EX.BASIC' TO DATA ELSE STOP
 READ A FROM DATA, 'XYZ' ELSE STOP
 A<3>='*'
 WRITE A ON DATA, 'XYZ'
 CLOSE DATA
```

CLOSESEQ statement

Use the CLOSESEQ statement after opening and processing a file opened for sequential processing. CLOSESEQ makes the file available to other users.

Syntax

CLOSESEQ *file.variable* [ON ERROR *statements*]

file.variable specifies a file previously opened with an OPENSEQ statement. If the file is neither accessible nor open, the program terminates with a runtime error message. If *file.variable* is the null value, the CLOSESEQ statement fails and the program terminates with a run-time error message.

The ON ERROR clause

The ON ERROR clause is optional in the CLOSESEQ statement. The ON ERROR clause lets you specify an alternative for program termination when a fatal error is encountered during processing of the CLOSESEQ statement.

If a fatal error occurs, and the ON ERROR clause was not specified, or was ignored (as in the case of an active transaction), the following occurs:

- An error message appears.
- Any uncommitted transactions begun within the current execution environment roll back.
- The current program terminates.
- Processing continues with the next statement of the previous execution environment, or the program returns to the UniVerse prompt.

A fatal error can occur if any of the following occur:

- A file is not open.
- *file.variable* is the null value.
- A distributed file contains a part file that cannot be accessed.

If the ON ERROR clause is used, the value returned by the STATUS function is the error number.

Example

In this example, the CLOSESEQ statement closes FILE.E, making it available to other users:

```
OPENSEQ 'FILE.E', 'RECORD1' TO FILE ELSE ABORT
READSEQ A FROM FILE THEN PRINT A ELSE STOP
CLOSESEQ FILE
END
```

closeSocket function

Use the `closeSocket`() function to close a socket connection.

Syntax

`closeSocket`(*`socket_handle`*)

Where *socket_handle* is the handle to the socket you want to close.

Return codes

The following table describes the status of each return code.

Return code	Description
0	Success.
Non-zero	See Socket function error return codes, on page 584.

CloseXMLData function

After you finish using an XML data, use `CloseXMLData` to close the dynamic array variable.

Syntax

`Status=`**`CloseXMLData`**(*`xml_data_handle`*)

Parameter

The following table describes each parameter of the syntax.

Parameter	Description
xml_data_handle	The name of the XML data file handle created by the `OpenXMLData`() function.

Return values

The return value is one of the following:

XML.SUCCESS: Success.

XML.ERROR: Failed

XML.INVALID.HANDLE2: Invalid *xml_data_handle*

Example

The following example illustrates use of the CloseXMLData function:

```
status = CloseXMLData(STUDENT_XML)
```

COL1 function

Use the COL1 function after the execution of a FIELD function to return the numeric value for the character position that immediately precedes the selected substring. Although the COL1 function takes no arguments, parentheses are required to identify it as a function.

The value obtained from COL1 is local to the program or subroutine executing the FIELD function. Before entering a subroutine, the current value of COL1 in the main program is saved. The value of COL1 in the subroutine is initialized as 0. When control is returned to the calling program, the saved value of COL1 is restored.

If no FIELD function precedes the COL1 function, a value of 0 is returned. If the delimiter expression of the FIELD function is an empty string or the null value, or if the string is not found, the COL1 function returns a 0 value.

Syntax

COL1 ()

Examples

The FIELD function in the following example returns the substring CCC. COL1() returns 8, the position of the delimiter ($) that precedes CCC.

```
SUBSTRING=FIELD("AAA$BBB$CCC",'$',3)
POS=COL1()
PRINT POS
```

In the following example, the FIELD function returns a substring of 2 fields with the delimiter (.) that separates them: 4.5. COL1() returns 6, the position of the delimiter that precedes 4.

```
SUBSTRING=FIELD("1.2.3.4.5",'.',4,2)
POS=COL1()
PRINT POS
```

COL2 function

Use the COL2 function after the execution of a FIELD function to return the numeric value for the character position that immediately follows the selected substring. Although the COL2 function takes no arguments, parentheses are required to identify it as a function.

The value obtained from COL2 is local to the program or subroutine executing the FIELD function. Before entering a subroutine, the current value of COL2 in the main program is saved. The value of COL2 in the subroutine is initialized as 0. When control is returned to the calling program, the saved value of COL2 is restored.

If no FIELD function precedes the COL2 function, a value of 0 is returned. If the delimiter expression of the FIELD function is an empty string or the null value, or if the string is not found, the COL2 function returns a 0 value.

Syntax

COL2 ()

Examples

The FIELD function in the following example returns the substring 111. COL2() returns 4, the position of the delimiter (#) that follows 111.

```
SUBSTRING=FIELD("111#222#3","#",1)
P=COL2()
PRINT P
```

In the following example, the FIELD function returns a substring of two fields with the delimiter (&) that separates them: 7&8. COL2() returns 5, the position of the delimiter that follows 8.

```
SUBSTRING=FIELD("&7&8&B&","&",2,2)
S=COL2()
PRINT S
```

In the next example, FIELD() returns the whole string, because the delimiter (.) is not found. COL2() returns 6, the position after the last character of the string.

```
SUBSTRING=FIELD("9*8*7",".",1)
Y=COL2()
PRINT Y
```

In the next example, FIELD() returns an empty string, because there is no tenth occurrence of the substring in the string. COL2() returns 0 because the substring was not found.

```
SUBSTRING=FIELD("9*8*7","*",10)
O=COL2()
PRINT O
```

COMMAND.EDITOR

The COMMAND.EDITOR command enables or disables the Command Editor in PI/open. The Command Editor provides you with facilities for simple command line editing and command stack manipulation.

You can turn on the Command Editor in either insert or overlay mode, and you can specify a prompt to use in place of the PERFORM colon prompt while command editing is enabled.

> **Note:** If a single character only is expected by the prompt from a PERFORM command or by an INFO or BASIC INPUT statement, the Command Editor is not enabled for that prompt.

Syntax

COMMAND.EDITOR [{ON | OFF} [INSERT | OVERLAY] [VERBS | ALL] "*prompt*"}]

COMMAND.EDITOR [OFF]

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
ON | OFF	Enables or disables the Command Editor. If ON, you must specify the other parameters. If OFF, disables the Command Editor and restores the PERFORM colon prompt, if you previously used the command editor to define a different prompt string.
INSERT | OVERLAY	Specifies what mode to use - insert or overlay. INSERT is the default setting. In insert mode, you can edit the command line or manipulate the stack while at the PERFORM colon prompt.
VERBS | ALL	Specifies what type of editing to perform. If VERBS, it enables the editing of text in answer to a prompt from a PERFORM command. If ALL, it enables the full editing facilities, including editing of text entered in response to an INFO or BASIC INPUT statement.
"*prompt*"	Specifies the prompt that you want to replace the PERFORM colon prompt while the Command Editor is enabled. The *prompt* string must be enclosed in single or double quotation marks.

Command Editor line editing functions

When the Command Editor is enabled, an ordinary text character is accepted and displayed on the line that you are editing. You can use the keybindings supplied for your terminal or use the fundamental keybindings to execute any line-editing command.

The Command Editor supports the following line editing functions.

Function	Description
Backspace	The backspace function has no effect if the cursor is already at the beginning of the line. In insert mode, the backspace function moves one character position to the left and deletes the character in the new cursor position. Any text on the line to the right of the cursor is moved one position to the left with the cursor. In overlay mode, the backspace function replaces the character to the left of the cursor with a space. Text to the right of the cursor does not move.
Cursor left	Moves the cursor one position to the left unless the cursor is already at the beginning of the line.
Cursor right	Moves the cursor one position to the right unless the cursor is already at the end of the line.
Next word	Moves the cursor to the beginning of the next word or to the end of the line if there is no next word. Any combination of alphabetic characters is considered a word. Any nonalphabetic character terminates a word.
Previous word	Moves the cursor to the first character of the word that begins to the left of the current cursor position. It has no effect if the cursor is at the beginning of the line.
Toggle cursor start/ end of line	Moves the cursor to the beginning or end of the line, depending on the current cursor position within the line. The cursor always moves to whichever position is farthest from the current position.
Toggle insert/overlay mode	Switches between insert and overlay mode.
Insert space	Inserts a space at the current cursor position. Any text to the right of the cursor is moved to the right. Its main use is in overlay mode, in which typing a space overwrites the current character.

Function	Description
Delete character	Deletes the character at the current cursor position. All text to the right of the cursor is moved to the left. It has no effect if the cursor is at the end of the line.
Delete word	Deletes the text from the current cursor position to the end of the first word that terminates to the right of that position. It has no effect if the cursor is at the end of the line.
Delete line	Deletes the current line, and places the cursor at the start of the resulting blank line.
Truncate line	Deletes text from the current cursor position to the end of the line. It has no effect if the cursor is at the end of the line.
Exchange previous two characters	Swaps the positions of the two characters that immediately precede the current cursor position. If the cursor is placed on the first or second character of the line, the function has no effect.
Restore deleted text	If a delete word, delete line, or truncate line function has been used, this function restores the display of the deleted text beginning at the current cursor position. The cursor is placed immediately after the end of the restore text. Any text that was to the right of the current cursor position is moved to the right by the restored text. If a delete word, delete line, or truncate line function has not yet been used, this function has no effect.
Refresh	Rewrites the current command line on a new display line. It is useful when the current screen display has been disturbed, for example, by an operator message.
Convert line to uppercase	Redisplays the command line with each lowercase alphabetic character changed to an uppercase character. The cursor is then placed at the end of the line.
Cancel	Discards the line that you are editing and returns you to the PERFORM colon prompt. This function has no effect if you are editing input to a PERFORM command prompt or input into an INFO or BASIC INPUT statement.

Stack manipulation functions

The PERFORM command stack retains the last 98 commands numbered from 01, the most recent command, to 98. The command numbered 00 is the current line. The PERFORM command stack is a ring. When you cycle through the stack of the 98th command, the first command is then shown.

The following stack manipulation functions enable you to display, and recall commands from your stack so that you can use the Command Editor's line editing capabilities to edit those commands, if required, before re-executing them. All of the line editing functions described in the previous section are available to make changes to commands from your stack.

Re-execution copies the command to the bottom off the stack and places you at the bottom of the stack, unless if you execute command 01. In this case, no new stack entry is made even though you edited the command.

Function	Description
Cancel	When you are manipulating the stack, the Cancel function clears the command line, repositions to the bottom of the PERFORM command stack, and returns you to the PERFORM colon prompt. You can use this function to escape from a Next command or Previous command sequence without executing a command, and alto to abort Search and Goto commands.

Function	Description
Cycle up command stack function	Allows you display your PERFORM command stack one command at a time starting at the command numbered 01 and displaying the next command in ascending numerical order each time the function is executed.
Cycle down command stack function	Allows you to display the PERFORM command stack one command at a time starting at the command numbered 98 and displaying the next command in descending numerical order each time the function is executed.
Scroll up command stack	Displays the previous 20 PERFORM stack commands in ascending numerical order each time the function is executed. The stack is treated as a ring. During a consecutive sequence of scroll commands, the current position within the command stack is recorded, so that you can scroll through the entire stack. Any command other than a scroll command returns you to the bottom of the PERFORM command stack.
Scroll down command stack	Displays 20 PERFORM stack commands in descending numerical order each time the function is executed. It works similarly to the scroll up command stack function.
Goto command number	Prompts you for a command number. If you supply a valid number, the command is displayed for editing and/or execution. If the supplied number is invalid, an error message is displayed, and no further action is taken. Use the Cancel function to exit from the Goto function.
Search command stack	Prompts you for a search string, and conducts a case-sensitive search of the PERFORM command stack for a command which contains that string. The stack is treated as a ring, so that the complete stack is scanned if necessary. If a match is found, the command containing the search string is displayed for editing and/or execution. If no match is found, a warning message is displayed. If you press only Return in response to the search string prompt, the previous search string is used again. Use the Cancel function to exit from the Search function.

Stack commands

The following table provides a summary of stack commands that you might need when editing the stack.

Command	Description
`.D[n]`	Deletes a command or paragraph from the stack. *n* is the command number that you want to delete. If *n* is omitted, command number 01 is deleted by default.
`.?`	Displays a list of the stack commands.
`.D[paragraph.name | command.name]`	Deletes a sentence or paragraph from the VOC file. *paragraph.name* and *command.name* are the names of the paragraph or command to be deleted.

Command	Description
`.I[n] [any.text]`	Inserts a command into any location in the stack. You can insert a new command into the PERFORM command stack by specifying ? at the end of the command to be inserted or using the `.I` command. Inserting a command into the stack is often used to place a command within a group of related commands that you might want to save in the VOC as a paragraph. If ? is specified at the end of the command to be inserted, the command is inserted into the stack as command number 01. *n* is the command stack number that you want the inserted text to become. If *n* is omitted, *any.text* is inserted into the stack as command number 01 by default. A space must appear between the command stack number and the text that you want to insert. Commands already in the stack with a number equal to or greater than the number inserted have their number increased by one. All commands in the stack with numbers lower than the number inserted retain their original numbers. The `.I` command is often used with the `.S` command to place several stack commands into a paragraph in the VOC.
`.L [paragraph.name | command.name]`	Lists a paragraph or sentence from the VOC. *paragraph.name* and *command.name* are the record IDs of the paragraph or command in the VOC to be displayed.

`.R[n]`
 	
or
 	
`.R[paragraph.name | sentence.name]`
	Recalls or repositions to a command number 01 or paragraph. *n* is the stack number of the command that you want to recall. If you omit *n*, command number 01 is the default.
	
	The command `.R` and `.R1` do not change the order of commands in the stack. Stack command number 01 is not rewritten.
	
	Additionally, the `.R` command can be used to place a sentence or group of commands from a stored paragraph into the stack using the second syntax listed.

`.S name [s# [e#]]`	Saves the command or paragraph in the VOC as a sentence or paragraph. *name* is the name you are assigning to the command or group of commands that you want to save. If you only save one line, it is saved as a sentence, not as a paragraph. The command line number that you want to start with is s#, and the command line number you want to end with is e#.

COMMIT statement

Use the COMMIT statement to commit all file I/O changes made during a transaction. The WORK keyword is provided for compatibility with SQL syntax conventions; it is ignored by the compiler.

Syntax

COMMIT [WORK] [THEN *statements*] [ELSE *statements*]

A transaction includes all statements between a BEGIN TRANSACTION statement and the COMMIT statement or ROLLBACK statement that ends the transaction. Either a COMMIT or a ROLLBACK statement ends the current transaction.

The COMMIT statement can either succeed or fail.

When a subtransaction commits, it makes the results of its database operations accessible to its parent transaction. The subtransaction commits to the database only if all of its predecessors up to the top-level transaction are committed.

If a top-level transaction succeeds, all changes to files made during the active transaction are committed to disk.

If a subtransaction fails, all its changes are rolled back and do not affect the parent transaction. If the top-level transaction fails, none of the changes made during the active transaction are committed, and the database remains unaffected by the failed transaction. This ensures that the database is maintained in a consistent state.

If the COMMIT statement succeeds, the THEN statements are executed; any ELSE statements are ignored. If COMMIT fails, any ELSE statements are executed. After the THEN or the ELSE statements are executed, control is transferred to the statement following the next END TRANSACTION statement.

All locks obtained during a transaction remain in effect for the duration of the active transaction; they are not released by a RELEASE statement, WRITE statements, WRITEV statement, or MATWRITE statements that are part of the transaction. The parent transaction adopts the acquired or promoted locks. If a subtransaction rolls back, any locks that have been acquired or promoted within that transaction are demoted or released.

The COMMIT statement that ends the top-level transaction releases locks set during that transaction. Locks obtained outside the transaction are not affected by the COMMIT statement.

If no transaction is active, the COMMIT statement generates a runtime warning, and the ELSE statements are executed.

Example

This example begins a transaction that applies locks to rec1 and rec2. If no errors occur, the COMMIT statement ensures that the changes to rec1 and rec2 are written to the file. The locks on rec1 and rec2 are released, and control is transferred to the statement following the END TRANSACTION statement.

```
BEGIN TRANSACTION
 READU data1 FROM file1,rec1 ELSE ROLLBACK
 READU data2 FROM file2,rec2, ELSE ROLLBACK
 .
 .
 .
 WRITE new.data1 ON file1,rec1 ELSE ROLLBACK
 WRITE new.data2 ON file2,rec2 ELSE ROLLBACK
 COMMIT WORK
 END TRANSACTION
```

The update record lock on rec1 is not released on completion of the first WRITE statements but on completion of the COMMIT statement.

COMMON statement

Use the COMMON statement to provide a storage area for variables. Variables in the common area are accessible to main programs and external subroutines. Corresponding variables can have different names in the main program and in external subroutines, but they must be defined in the same order. The COMMON statement must precede any reference to the variables it names.

Syntax

COM[MON] [/*name*/] *variable* [,*variable* …]

A common area can be either named or unnamed. An unnamed common area is lost when the program completes its execution and control returns to the UniVerse command level. A named common area remains available for as long as the user remains in the UniVerse environment.

The common area name can be of any length, but only the first 31 characters are significant.

Arrays can be dimensioned and named with a COMMON statement. They can be redimensioned later with a DIMENSION statement, but the COMMON statement must appear before the DIMENSION statement. When an array is dimensioned in a subroutine, it takes on the dimensions of the array in the main program regardless of the dimensions stated in the COMMON statement. For a description of dimensioning array variables in a subroutine, see the CALL statement, on page 81.

When programs share a common area, use the $INCLUDE statement to define the common area in each program.

Example

Program:

```
COMMON NAME, ADDRESS (15, 6), PHONE
```

Subroutine:

```
COMMON A, B (15, 6), C
```

In this example the variable pairs NAME and A, ADDRESS and B, PHONE and C are stored in the same memory location.

COMPARE function

Use the COMPARE function to compare two strings and return a numeric value indicating the result.

Syntax

COMPARE (*string1*, *string2* [, *justification*])

string1, string2 specify the strings to be compared.

justification is either L for left-justified comparison or R for right-justified comparison. (Any other value causes a run-time warning, and 0 is returned.)

The comparison can be left-justified or right-justified. A right-justified comparison compares numeric substrings within the specified strings as numbers. The numeric strings must occur at the same character position in each string. For example, a right-justified comparison of the strings AB100 and AB99 indicates that AB100 is greater than AB99 since 100 is greater than 99. A right-justified comparison of the strings AC99 and AB100 indicates that AC99 is greater since C is greater than B.

If neither L nor R is specified, the default comparison is left-justified.

The following list shows the values returned:

Value	Description
–1	*string1* is less than *string2*.
0	*string1* equals *string2* or the justification expression is not valid.
1	*string1* is greater than *string2*.

If NLS is enabled, the COMPARE function uses the sorting algorithm and the Collate convention specified in the NLS.LC.COLLATE file in order to compare the strings. For more information about conventions, see the *NLS Guide*.

Examples

In the following example, the strings AB99 and AB100 are compared with the right-justified option and the result displayed. In this case the result displayed is –1.

```
PRINT COMPARE('AB99','AB100','R')
```

An example in NLS mode follows. It compares the strings anilno and anillo, returning the result as 1. It sets the locale to Spanish and compares the strings again. In this case, the result displayed is –1.

```
$INCLUDE UNIVERSE.INCLUDE UVNLSLOC.H
x=SETLOCALE(UVLC$ALL, 'OFF' )
PRINT COMPARE('anilno', 'anillo', 'L' )
x=SETLOCALE(UVLC$ALL, 'ES-SPANISH' )
PRINT COMPARE('anilno', 'anillo', 'L' )
```

This is the program output:

```
1
 -1
```

The CONTINUE statement is a loop-controlling statement. For syntax details, see the FOR statement, on page 178 and the LOOP statement, on page 248.

CONVERT function

Use the CONVERT function to return a copy of *variable* with every occurrence of specified characters in *variable* replaced with other specified characters. Every time a character to be converted appears in *variable*, it is replaced by the replacement character.

Syntax

CONVERT (*expression1*, *expression2*, *variable*)

expression1 specifies a list of characters to be converted. *expression2* specifies the corresponding replacement characters. The first character of *expression2* replaces all instances of the first character of *expression1*, the second character of *expression2* replaces all instances of the second character of *expression1*, and so on.

If *expression2* contains more characters than *expression1*, the extra characters are ignored. If *expression1* contains more characters than *expression2*, the characters with no corresponding *expression2* characters are deleted from the result.

If *variable* is the null value, null is returned. If either *expression1* or *expression2* is the null value, the CONVERT function fails and the program terminates with a run-time error message.

The CONVERT function works similarly to the CONVERT statement.

Example

```
A="NOW IS THE TIME"
PRINT A
A=CONVERT('TI','XY',A)
PRINT A
```

```
A=CONVERT('XY','T',A)
PRINT A
```

This is the program output:

```
NOW IS THE TIME
NOW YS XHE XYME
NOW S THE TME
```

CONVERT statement

Use the CONVERT statement to replace every occurrence of specific characters in a string with other characters. Every time the character to be converted appears in the string, it is replaced by the replacement character.

Syntax

CONVERT *expression1* TO *expression2* IN *variable*

expression1 specifies a list of characters to be converted. *expression2* specifies a list of replacement characters. The first character of *expression2* replaces all instances of the first character of *expression1*, the second character of *expression2* replaces all instances of the second character of *expression1*, and so on.

If *expression2* contains more characters than *expression1*, the extra characters are ignored. If *expression1* contains more characters than *expression2*, the characters with no corresponding *expression2* characters are deleted from the variable.

If *variable* is the null value, null is returned. If either *expression1* or *expression2* evaluates to the null value, the CONVERT statement fails and the program terminates with a run-time error message.

Example

```
A="NOW IS THE TIME"
PRINT A
CONVERT 'TI' TO 'XY' IN A
PRINT A
CONVERT 'XY' TO 'T' IN A
PRINT A
```

This is the program output:

```
NOW IS THE TIME
NOW YS XHE XYME
NOW S THE TME
```

COS function

Use the COS function to return the trigonometric cosine of an angle. *expression* is an angle expressed as a numeric value in degrees. The COS function is the inverse of the ACOS function.

Values outside the range of 0 to 360 degrees are interpreted as modulo 360. Numbers greater than 1E17 produce a warning message and 0 is returned. If *expression* evaluates to the null value, null is returned.

Syntax

COS (*expression*)

Example

```
PRINT "COS(45) = " : COS(45)
END
```

This is the program output:

```
COS(45) = 0.7071
```

COSH function

Use the COSH function to return the hyperbolic cosine of *expression*. *expression* must be a numeric value.

If *expression* evaluates to the null value, null is returned.

Syntax

COSH (*expression*)

Example

```
PRINT "COSH(2) = ":COSH(2)
```

This is the program output:

```
COSH(2) = 1.0006
```

COUNT function

Use the COUNT function to return the number of times a substring is repeated in a string value.

Syntax

COUNT (*string, substring*)

string is an expression that evaluates to the string value to be searched. *substring* is an expression that evaluates to the substring to be counted. *substring* can be a character string, a constant, or a variable.

If *substring* does not appear in *string*, a 0 value is returned. If *substring* is an empty string, the number of characters in *string* is returned. If *string* is the null value, null is returned. If *substring* is the null value, the COUNT function fails and the program terminates with a run-time error message.

By default, each character in string is matched to *substring* only once. Therefore, when *substring* is longer than one character and a match is found, the search continues with the character following the matched substring. No part of the matched string is recounted toward another match. For example, the following statement counts two occurrences of substring TT and assigns the value 2 to variable C:

```
C = COUNT ('TTTT', 'TT')
```

PICK, IN2, and REALITY flavors

In PICK, IN2, and REALITY flavors, the COUNT function continues the search with the next character regardless of whether it is part of the matched string. For example, the following statement counts three occurrences of substring TT:

```
C = COUNT ('TTTT', 'TT')
```

Use the COUNT.OVLP option of the $OPTIONS statement to get this behavior in IDEAL and INFORMATION flavor accounts.

Example

```
A=COUNT('ABCAGHDALL','A')
PRINT "A= ",A
*
Z='S#FF##G#JJJJ#'
Q=COUNT(Z,'#')
PRINT "Q= ",Q
*
Y=COUNT('11111111','11')
PRINT "Y= ",Y
```

This is the program output:

```
A=         3
Q=         5
Y=         4
```

COUNTS function

Use the COUNTS function to count the number of times a substring is repeated in each element of a dynamic array. The result is a new dynamic array whose elements are the counts corresponding to the elements in *dynamic.array*.

Syntax

COUNTS (*dynamic.array, substring*)

CALL -**COUNTS** (*return.array, dynamic.array, substring*)

CALL !**COUNTS** (*return.array, dynamic.array, substring*)

dynamic.array specifies the dynamic array whose elements are to be searched.

substring is an expression that evaluates to the substring to be counted. *substring* can be a character string, a constant, or a variable.

Each character in an element is matched to *substring* only once. Therefore, when *substring* is longer than one character and a match is found, the search continues with the character following the matched substring. No part of the matched element is recounted toward another match.

If *substring* does not appear in an element, a 0 value is returned. If *substring* is an empty string, the number of characters in the element is returned. If *substring* is the null value, the COUNTS function fails and the program terminates with a runtime error message.

If any element in *dynamic.array* is the null value, null is returned.

If you use the subroutine syntax, the resulting dynamic array is returned as *return.array*.

PICK, IN2, and REALITY flavors

In PICK, IN2, and REALITY flavors, the COUNTS function continues the search with the next character regardless of whether it is part of the matched string. Use the COUNT.OVLP option of the $OPTIONS statement to get this behavior in IDEAL and INFORMATION flavor accounts.

Example

```
ARRAY="A":@VM:"AA":@SM:"AAAAA"
PRINT COUNTS(ARRAY, "A")
PRINT COUNTS(ARRAY, "AA")
```

This is the program output:

```
1V2S5
 0V1S2
```

CREATE statement

Use the CREATE statement after an OPENSEQ statement to create a record in a type 1 or type 19 UniVerse file or to create a UNIX or DOS file. CREATE creates the record or file if the OPENSEQ statement fails. An OPENSEQ statement for the specified *file.variable* must be executed before the CREATE statement to associate the path or record ID of the file to be created with the *file.variable*. If *file.variable* is the null value, the CREATE statement fails and the program terminates with a runtime error message.

Use the CREATE statement when OPENSEQ cannot find a record or file to open and the next operation is to be a NOBUF statement, READSEQ statement, or READBLK statement. You need not use the CREATE statement if the first file operation is a WRITESEQ statement, since WRITESEQ creates the record or file if it does not exist.

If the record or file is created, the THEN statements are executed, and the ELSE statements are ignored. If no THEN statements are specified, program execution continues with the next statement.

If the record or file is not created, the ELSE statements are executed; any THEN statements are ignored.

Syntax

CREATE *file.variable* {THEN *statements* [ELSE *statements*] | ELSE *statements*}

File buffering

Normally UniVerse uses buffering for sequential input and output operations. Use the NOBUF statement after an OPENSEQ statement to turn off buffering and cause all writes to the file to be performed immediately. For more information about file buffering, see the NOBUF statement.

Example

In the following example, RECORD4 does not yet exist. When OPENSEQ fails to open RECORD4 to the file variable FILE, the CREATE statement creates RECORD4 in the type 1 file FILE.E and opens it to the file variable FILE.

```
OPENSEQ 'FILE.E', 'RECORD4' TO FILE
 ELSE CREATE FILE ELSE ABORT
 WEOFSEQ FILE
```

```
WRITESEQ 'HELLO, UNIVERSE' TO FILE ELSE STOP
```

createCertificate function

The `createCertificate`() function generates a certificate. The certificate can either be a self-signed certificate as a root CA that can then be used later to sign other certificates, or it can be a CA signed certificate. The generated certificate conforms to X509V3 standard.

Syntax

`createCertificate``(action, req, signKey, keyPass, CAcert, days, extensions, certOut, signAlg)`

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
action	1 - Creating a self-signed root certificate (SSL_CERT_SELF_SIGN) 2 - Creating an intermediate CA certificate (SSL_CERT_CA_SIGN) 3 - Creating a server/client certificate (SSL_CERT_LEAF_SIGN)
req	A string containing the certificate signing request file name.
signKey	A string containing the private key file name.
keyPass	A string containing the pass phrase to protect the private key.
CAcert	A string containing the CA certificate.
days	The number of days the certificate is valid for. The default is 365 days.
extensions	A string containing extension specifications.
certOut	A string containing the generated certificate file.
signAlg	Allows users to specify a signing digest algorithm. The value can be any digest algorithm supported by OpenSSL. For example, MD5, SHA1, SHA224, SHA256, SHA384, or SHA512.

Return code status

The following table describes the status of each return code.

Return code	Status
0	Success.
1	Cannot read certificate request file.
2	Cannot read the key file.
3	Cannot read the CA certificate file.
4	Cannot generate the certificate.

As input, a certificate request file must be specified by *req*. Three actions can be chosen:

- Creating a self-signed root certificate
- Creating an intermediate CA certificate
- Creating a server/client certificate

For self-signed root certificates, a key file must be specified by *signKey*. For the other two actions, a CA certificate file must be specified by *CAcert*, along with the CA private key specified by *signKey*. The output certificate file is specified by *certOut*. The format for these files should all be in PEM format.

The difference between intermediate CA certificates and server/client certificates is that the intermediate CA certificate can be used to sign other certificates, while a server/client certificate cannot be used to sign other certificates.

The *days* parameter specifies the number of days the generated certificate is valid. The certificate is valid starting from the current date until the number of days specified expires. If an invalid *days* value is provided (0 or negative) the default value of 365 (one year) will be used.

Note: This function is provided mainly for the purpose of enabling application development and testing. As such, the certificate generated contains only a minimum amount of information and does not allow all permissible extensions specified by the X509 standard that are supported by many other vendors. You can use XAdmin. It is recommended that you implement a complete PKI solution partnered with a reputed PKI solution vendor.

createCertRequest function

The `createCertRequest`() function generates a PKCS #10 certificate request from a private key in PKCS #8 form and a set of user specified data. The request can be sent to a CA or used as a parameter to `createCertificate`() to obtain an X.509 public key certificate.

Syntax

`createCertRequest``(`*`key, inFormat, keyLoc, algorithm, digest, passPhrase, subjectData, outFile, outFormat`*`)`

The private key and its format, type, algorithm, and pass phrase are specified the same.

The certificate request will typically contain the information described in the following table.

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
key	A string containing the key or name of the file storing the key.
inFormat	The key format. 1 - PEM (SSL_FMT_PEM) 2 - DER (SSL_FMT_DER)
keyLoc	1 - Put the key into string privKey/pubKey (SSL_LOC_STRING). 2 - Put the key into a file (SSL_LOC_FILE).
algorithm	Flag 1- RSA key (SSL_KEY_RSA) 2- DSA key (SSL_KEY_DSA)

Parameter	Description
digest	1 - MD5 (SSL_DIGEST_MD5) 2 - SHA1 (SSL_DIGEST_SHA1) 3 - SHA224 (SSL_DIGEST_SHA224) 4 - SHA256 (SSL_DIGEST_SHA256) 5 - SHA384 (SSL_DIGEST_SHA384) 6 - SHA512 (SSL_DIGEST_SHA512)
passPhrase	A string storing the *passPhrase* to protect the private key.
subjectData	The requester's identification information.
outFile	A string containing the path name of the file where the certificate request is stored. By convention, this file should have a `.req` as its extension.
outFormat	The generated certificate format. 1 - PEM (SSL_FMT_PEM) 2 - DER (SSL_FMT_DER)

Return code status

The following table describes the status of each return code.

Return code	Status
0	Success.
1	Private key file cannot be opened.
2	Unrecognized key or certificate format.
3	Unrecognized key type.
4	Unrecognized encryption algorithm.
5	Unrecognized key (corrupted key or algorithm mismatch).
6	Invalid pass phrase.
7	Invalid subject data (illegal format or unrecognized attribute, and so forth).
8	Invalid digest algorithm.
9	Output file cannot be created.
99	Cert request cannot be generated.

The request can be sent to a CA or used as a parameter to `createCertificate`() to obtain an X.509 public key certificate.

The private key and its format, type, algorithm and pass phrase are specified the same.

The certificate signing request typically contains the information described in the following table.

Item	Description
Version	Defaults to 0.
Subject	The certificate holder's identification data. This includes country, state/province, locality (city), organization, unit, common name, email address, and so on.
Public key	The key's algorithm (RSA or DSA) and value.
Signature	The requester's signature, (signed by the private key).

The subject data must be provided by the requester through the dynamic array, *subjectData*. It contains @FM separated attributes in the form of “attri=value”.

The commonly used *subjectData* attributes are described in the following table.

Item	Description	Example
C	Country	C=US
ST	State	ST=Colorado
L	Locality	L=Denver
O	Organization	O=MyCompany
OU	Organization Unit	OU=Sales
CN	Common Name	CN=service@mycompany.com
Email	Email Address	Email=john.doe@mycompany.com

Be aware that since the purpose of a certificate is to associate the certificate’s bearer with his or her identity, in order for the outside party to verify the identity of the certificate’s holder, some recognizable characteristics should be built between the holder and verifier. For example, it is a general practice that a server’s certificate uses its DNS name (such as myServer.com) as its common name (CN).

Digest specifies what algorithm is going to be used to generate a Message Authentication Code (MAC) which will then be signed with the provided private key as a digital signature as part of the request. Currently only two algorithms, MD5 and SHA1, are supported. SHA1 is recommended.

Note: For a DSA request, SHA1 will always be used.

For more information on certificates, see the references for X.509 and PKCS #10, and PKCS # 12.

createRequest function

The `createRequest` function creates an HTTP request and returns a handle to the request.

Syntax

`createRequest``(`*`URL, http_method, request_handle`*`)`

URL is a string containing the URL for a resource on a web server. An accepted URL must follow the specified syntax defined in RFC 1738. The general format is: http://<host>:<port>/<path>?<searchpart>. The host can be either a name string or IP address. The port is the port number to connect to, which usually defaults to 80 and is often omitted, along with the preceding colon. The path tells the web server which file you want, and, if omitted, means “home page” for the system. The searchpart can be used to send additional information to a web server.

http_method is a string which indicates the method to be performed on the resource. See the table below for the available (case-sensitive) methods.

request_handle is a handle to the request object.

The following table describes the available methods for *http_method.*

Method	Description
GET	Retrieves whatever information, in the form of an entity, identified by the Request-URI. If the Request-URI refers to a data-producing process, it is the produced data which shall be returned as the entity in the response and not the source text of the process, unless that text happens to be the output of the process.
POST	[:<MIME-type>] For this method, it can also have an optional MIME-type to indicate the content type of the data the request intends to send. If no MIME-type is given, the default content type will be “application/x-www-form-urlencoded.” Currently, only “multipart/form-data” is internally supported, as described in function addRequestParameter() and submitRequest(), although other “multipart/*” data can also be sent if the user can assemble it on his/her own. (The multipart/form-data format itself is thoroughly described in RFC 2388).
HEAD	The HEAD method is identical to GET except that the server MUST NOT return a message-body in the response. The metainformation contained in the HTTP headers in response to a HEAD request SHOULD be identical to the information sent in response to a GET request. This method can be used for obtaining metainformation about the entity implied by the request without transferring the entity-body itself. This method is often used for testing hypertext links for validity, accessibility, and recent modification.
OPTIONS	The OPTIONS method represents a request for information about the communication options available on the request/response chain identified by the Request-URI. This method allows the client to determine the options and/or requirements associated with a resource, or the capabilities of a server, without implying a resource action or initiating a resource retrieval. HTTP 1.1 and later.
DELETE	The DELETE method requests that the origin server delete the resource identified by the Request-URI. HTTP 1.1 and later.
TRACE	The TRACE method is used to invoke a remote, application-layer loop- back of the request message. HTTP 1.1 and later.
PUT	The PUT method requests that the enclosed entity be stored under the supplied Request-URI. HTTP 1.1 and later but not supported.
CONNECT	/* HTTP/1.1 and later but not supported */

Return codes

The following table describes the status of each return code.

Return code	Status
0	Success.
1	Invalid URL (Syntactically).
2	Invalid method (For HTTP 1.0, only GET/POST/HEAD)

createSecureRequest function

The `createSecureRequest` function behaves exactly the same as the `createRequest`() function, except for the fourth parameter, a handle to a security context, which is used to associate the security context with the request. If the URL does not start with “https” then the parameter is ignored. If the URL starts with “https” but an invalid context handle or no handle is provided, the function will abort and return with an error status.

Syntax

createSecureRequest(*URL, http_method, request_handle, security_context*)

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
URL	A string containing the URL for a resource on a web server. An accepted URL must follow the specified syntax defined in RFC 2818. The general format is: https://<host>:<port>/<path>?<searchpart>. The host can be either a name string or IP address. The port is the port number to connect to, which usually defaults to 443 and is often omitted, along with the preceding colon. The path tells the web server which file you want, and, if omitted, means “home page” for the system. The searchpart can be used to send additional information to a web server.
http_method	A string which indicates the method to be performed on the resource. See the table below for the available (case-sensitive) methods.
request_handle	A handle to the request object.
securityContext	A handle to the security context.

The following table describes the available methods for *http_method*.

Method	Description
GET	Retrieves whatever information, in the form of an entity, identified by the Request-URI. If the Request-URI refers to a data-producing process, it is the produced data which shall be returned as the entity in the response and not the source text of the process, unless that text happens to be the output of the process.
POST	[:<MIME-type>] For this method, it can also have an optional MIME-type to indicate the content type of the data the request intends to send. If no MIME-type is given, the default content type will be “application/x-www-form-urlencoded”. Currently, only “multipart/form-data” is internally supported, as described in function addRequestParameter() and submitRequest(), although other “multipart/* data can also be sent if the user can assemble it on his/her own. (The multipart/form-data format itself is thoroughly described in RFC 2388).
HEAD	The HEAD method is identical to GET except that the server MUST NOT return a message-body in the response. The metainformation contained in the HTTP headers in response to a HEAD request SHOULD be identical to the information sent in response to a GET request. This method can be used for obtaining metainformation about the entity implied by the request without transferring the entity-body itself. This method is often used for testing hypertext links for validity, accessibility, and recent modification.
OPTIONS	The OPTIONS method represents a request for information about the communication options available on the request/response chain identified by the Request-URI. This method allows the client to determine the options and/or requirements associated with a resource, or the capabilities of a server, without implying a resource action or initiating a resource retrieval. HTTP 1.1 and later.
DELETE	The DELETE method requests that the origin server delete the resource identified by the Request-URI. HTTP 1.1 and later.

Method	Description
TRACE	The TRACE method is used to invoke a remote, application-layer loop- back of the request message. HTTP 1.1 and later.
PUT	The PUT method requests that the enclosed entity be stored under the supplied Request-URI. HTTP 1.1 and later but not supported.
CONNECT	/* HTTP/1.1 and later but not supported */

Return codes

The following table describes the status of each return code.

Return code	Status
0	Success.
1	Invalid URL (Syntactically).
2	Invalid method (For HTTP 1.0, only GET/POST/HEAD)

Note: If URL does include a searchpart, it must be in its encoded format (space is converted into +, and other non-alphanumeric characters are converted into %HH format. See addRequestParameter() for more details). However, host and path are allowed to have these “unsafe” characters. UniVerse BASIC will encode them before communicating with the web server.

createSecurityContext function

The `createSecurityContext`() function creates a security context and returns a handle to the context.

Syntax

`createSecurityContext`(*`context, “protocol version`*:[*`rule`*],...”)

A security context is a data structure that holds all aspects of security characteristics that the application intends to associate with a secured connection. Specifically, the following information can be held for each context:

- Protocol version
- Sender’s certificate to be sent to the peer
- Sender’s private key for signature and key exchange
- Issuer’s certificate or certificate chain to be used to verify incoming certificate
- Certificate verification depth, strength and other rules
- Certificate Revocation List
- Flag to perform client authentication (useful for server socket only)
- Context ID and time stamp

For any given connection, not all of the information is required.

A version (SSL version 2 or 3 or TLS version 1) can be associated with a security context. It specifies what protocol or protocols are allowed for an SSL connection using the security context record. The version string is a list of version specifications separated by commas. Each version specification contains a protocol version and an optional rule, separated by a colon.

Currently there are five supported protocol versions: SSLv2, SSLv3, TLSv1, TLSv1.1, and TLSv1.2, listed in order of security strength. For all practical purposes, SSLv2 should never be used. To comply with the latest regulations, only TLSv1.1 and TLSv.1.2 should be used.

If the version contains only one protocol version without a rule, it means the minimal protocol allowed. For example, `createSecurityContext(myctx, "TLSv1")` means that the allowed protocols are TLSv1, TLSv1.1, and TLSv1.2. If no version is provided (for example, a null string is specified), the default version will be SSLv3, TLSv1, TLSv1.1, and TLSv1.2.

Rule example 1

In this example, SSLv3, TLSv1.1 and TLSv1.2 are allowed:

```
createSecurityContext(myctx, "SSLv3:min,TLSv1:no,TLSv1.2:max")
```

Rule example 2

In this example, only TLSv1.1 is allowed:

```
createSecurityContext(myctx, "TLSv1.1:only")
```

Note that the actual protocols allowed during an SSL session are determined at runtime based on the versions specified by the `createSecurityContext()` function and the SSL_PROTOCOLS value defined in the `uvconfig` (UniVerse) or `udtconfig` (UniData) file. Only protocol strings that are specified by both the Basic API and the configuration files are considered.

For example, if the SSL_PROTOCOLS parameter contains “TLSv1.1,TLSv1.2”, the actual protocols allowed during negotiation are TLSv1.1 and TLSv1.2, if the SCR created in rule example1 (above) is used.

In rule example 2 (above), the actual protocol allowed is “TLSv1.1”.

For secure socket connections and socket APIs, `openSecureSocket()` or `initSecureServerSocket()` must be called to associate a security context with a connection by a client or a server, respectively.

For secure HTTP connection (https), you must supply a valid context handle with the `createSecureRequest()` function.

All aspects of a security context can be changed by the APIs described in the following table.

Parameters

Parameter	Description
context	The security context handle.
protocol version	A string with the following values: SSLv2 SSLv3 TLSv1 TLSv1.1 (UniVerse 11.2.5 or later and UniData 8.1 or later) TLSv1.2 (UniVerse 11.2.5 or later and UniData 8.1 or later)

Parameter	Description
rule	Defines the minimum and maximum protocol value. Available rule options are: Min - The minimum version, from the specified version to the highest version. Max - The maximum version, from the lowest version to the specified version. No - Do not use the specified version. Only - Use only the specified version.

Return code status

Return code	Status
0	Success.
1	Security context could not be created.
2	Invalid version.

CRT statement

Use the CRT statement to print data on the screen, regardless of whether a PRINTER ON statement has been executed. The syntax for *print.list* is the same as for a PRINT statement.

Syntax

CRT [*print.list*]

print.list can contain any BASIC expression. The elements of the list can be numeric or character strings, variables, constants, or literal strings; the null value, however, cannot be output. The list can consist of a single expression or a series of expressions separated by commas (,) or colons (:) for output formatting. If no *print.list* is designated, a blank line is output.

Expressions separated by commas are printed at preset tab positions. You can use multiple commas together to cause multiple tabulation between expressions.

Expressions separated by colons are concatenated. That is, the expression following the colon is printed immediately after the expression preceding the colon. To print a list without a LINEFEED and RETURN, end the *print.list* with a colon (:).

The CRT statement works similarly to the DISPLAY statement, on page 133.

If NLS is enabled, the CRT statement uses the terminal map in order to print. For more information about maps and devices, see the *NLS Guide*.

Example

```
CRT "This can be used to print something on the"
CRT "terminal while"
CRT "the PRINTER ON statement is in effect."
```

The program output on the terminal is:

```
This can be used to print something on the
terminal while
```

```
the PRINTER ON statement is in effect.
```

DATA statement

Use the DATA statement to place values in an input stack. These values can be used as responses to INPUT statements executed later in the program or in a subroutine (see the INPUT statement). The values can also serve as responses to UniVerse commands that request input.

Syntax

DATA *expression* [,*expression* …]

Expressions used in DATA statements can be numeric or character string data. The null value cannot be stored in the input stack. If *expression* evaluates to null, the DATA statement fails and the program terminates with a runtime error message.

Put a comma at the end of each line of a DATA statement to indicate that more data expressions follow on the next line.

The order in which expressions are specified in the DATA statement is the order in which the values are accessed by subsequent INPUT statements: first-in, first-out. When all DATA values have been exhausted, the INPUT statement prompts the user for a response at the terminal.

The DATA statement must be executed before an INPUT statement that is to use *expression* for input.

You can store up to 512 characters in a data stack.

You can list the current data in the stack from your program by accessing the @DATA.PENDING variable with the statement:

```
PRINT @DATA.PENDING
```

Example

In the following example, the INPUT NBR statement uses the first value placed in the input stack by the DATA statement, 33, as the value of NBR. The INPUT DESCR statement uses the second value, 50, as the value of DESCR. The INPUT PRICE statement uses the third value, 21, as the value of PRICE.

```
X=33;   Y=50; Z=21
DATA X,Y,Z
X=Y+Z
*
INPUT NBR
INPUT DESCR
INPUT PRICE
INPUT QTY
PRINT NBR,DESCR,PRICE,QTY
```

This is the program output:

```
?33
 ?50
 ?21
 ?2
 33       50      21      2
```

The value of NBR is the value of X when the DATA statement is executed, not the current value of X (namely, Y+Z). The INPUT QTY statement has no corresponding value in the input stack, so it prompts the user for input.

DATE function

Use the DATE function to return the numeric value of the internal system date. Although the DATE function takes no arguments, parentheses are required to identify it as a function.

Syntax

DATE ()

The internal format for the date is based on a reference date of December 31, 1967, which is day 0. All dates thereafter are positive numbers representing the number of days elapsed since day 0. All dates before day 0 are negative numbers representing the number of days before day 0. For example:

Date	Internal representation
December 10, 1967	-21
November 15, 1967	-46
February 15, 1968	46
January 1, 1985	6575

Example

```
PRINT DATE()
PRINT OCONV(DATE(),"D2/")
```

This is the program output:

```
9116
12/15/92
```

DBTOXML function

To create an XML document from the UniVerse database using UniVerse BASIC, use the DBTOXML function.

Syntax

DBTOXML(*xml_document, doc_location, u2xmap_file, u2xmap_location, condition, status*)

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
xml_document	The name of the XML document to create.

Parameter	Description
doc_flag	A flag defining the type of *xml_document*. Valid values are: ▪ XML.FROM.DOM - *xml_document* is a DOM handle. ▪ XML.FROM.FILE - *xml_document* is a file name. ▪ XML.FROM.STRING - *xml_document* is a string located within the UniVerse BASIC program..
u2xmap_file	The name of the U2XMAP file to use to produce the XML document.
u2xmap_location	A flag indicating if the mapping file is the U2XMAP file itself or a string located within the UniVerse BASIC program. Valid values are: ▪ XMAP.FROM.FILE - the mapping rules are contained in a U2XMAP file. ▪ XMAP.FROM.STRING - *u2xmapping_rules* is the name of the variable containing the mapping rules.
condition	The conditions to use when selecting data for the XML document.
Status	The return code.

Example

The following example illustrates the use of DBTOXML:

```
*DBTOXML("myXMlFile",XML.FROM.FILE,"myMapFile",XML.FROM.FILE,STATUS)
```

DCOUNT function

Use the DCOUNT function to return the number of delimited fields in a data string.

Syntax

DCOUNT (*string, delimiter*)

string is an expression that evaluates to the data string to be searched.

delimiter is an expression that evaluates to the delimiter separating the fields to be counted. *delimiter* can be a character string of 0, 1, or more characters.

DCOUNT differs from the COUNT function in that it returns the number of values separated by delimiters rather than the number of occurrences of a character string. Two consecutive delimiters in *string* are counted as one field. If *delimiter* evaluates to an empty string, a count of 1 plus the number of characters in the string is returned. If *string* evaluates to an empty string, 0 is returned.

If *string* evaluates to the null value, null is returned. If *delimiter* evaluates to the null value, the DCOUNT function fails and the program terminates with a run-time error message.

PICK, IN2, and REALITY flavors

In PICK, IN2, and REALITY flavors, the DCOUNT function continues the search with the next character regardless of whether it is part of the matched delimiter string. Use the COUNT.OVLP option of the $OPTIONS statement to get this behavior in IDEAL and INFORMATION flavor accounts.

Example

```
REC="88.9.B.7"
Q=DCOUNT(REC,'.')
PRINT "Q= ",Q
```

```
REC=34:@VM:55:@VM:88:@VM:"FF":@VM:99:@VM:"PP"
R=DCOUNT(REC,@VM)
PRINT "R= ",R
```

This is the program output:

```
Q=          4
R=          6
```

DEACTIVATEKEY statement

Use the DEACTIVATEKEY command to deactivate one or more encryption keys. This command is useful to deactivate keys to make your system more secure.

Syntax

DEACTIVATEKEY *<key.id>, <password>* [ON *<hostname>*]

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
key.id	The key ID to deactivate.
password	The password corresponding to *key.id*.
ON *hostname*	The name of the remote host on which you want to deactivate the encryption key.

Note: You can deactivate only keys with password protection with this command. Keys that do not have password protection are automatically activated and cannot be deactivated.

Use the STATUS function after an DEACTIVATEKEY statement is executed to determine the result of the operation, as follows:

Value	Description
0	Operation successful.
1	Key is already activated. This applies to a single key, not a wallet operation.
2	Operation failed. This applies to a single key, not a wallet operation.
3	Invalid key or wallet ID or password.
4	No access to wallet.
5	Invalid key ID or password in a wallet.
6	No access to one of the keys in the wallet.
9	Other error.

DEBUG statement

Use the DEBUG statement to invoke RAID, the interactive UniVerse BASIC debugger. The DEBUG statement takes no arguments. When this statement is encountered, program execution stops and the double colon (::) prompt appears, waiting for a RAID command. The following table summarizes the RAID commands.

Syntax

DEBUG

Command	Action
line	Displays the specified line of the source code.
/[*string*]	Searches the source code for *string*.
B	Set a RAID breakpoint.
C	Continue program execution.
D	Delete a RAID breakpoint.
G	Go to a specified line or address and continue program execution.
H	Display statistics for the program.
I	Display and execute the next object code instruction.
L	Print the next line to be executed.
M	Set watchpoints.
Q	Quit RAID.
R	Run the program.
S	Step through the UniVerse BASIC source code.
T	Display the call stack trace.
V	Enter verbose mode for the M command.
V*	Print the compiler version that generated the object code.
W	Display the current window.
X	Display the current object code instruction and address.
X*	Display local run machine registers and variables.
Z	Display the next 10 lines of source code.
$	Turn on instruction counting.
#	Turn on program timing.
+	Increment the current line or address.
-	Decrement the current line or address.
.	Display the last object code instruction executed.
variable/	Print the value of *variable*.
variable!*string*	Change the value of *variable* to *string*.

DEFFUN statement

Use the DEFFUN statement to define a user-written function. You must declare a user-defined function before you can use it in a program. The DEFFUN statement provides the compiler with information such as the function name and the number and type of arguments. You can define a user-written function only once in a program. A subsequent DEFFUN statement for an already defined user-written function causes a fatal error.

Syntax

```
DEFFUN function [([MAT] argument [, [MAT] argument …] )]
 [CALLING call.ID]
```

function is the name of the user-written function.

*argument*s supply up to 254 arguments in the DEFFUN statement. To pass an array, you must precede the array name with the keyword MAT. An extra argument is hidden so that the user-defined function can use it to return a value. An extra argument is retained by the user-written function so that a value is returned by a RETURN (*value*) statement (for more information see the RETURN (value) statement, on page 328). If the RETURN (*value*) statement specifies no value, an empty string is returned. The extra argument is reported by the `MAP` and `MAKE.MAPE.FILE` commands.

call.ID is an expression that evaluates to the name by which the function is called if it is not the same as the function name. It can be a quoted string (the call ID itself) or a variable that evaluates to the call ID. If you do not use the CALLING clause, the user-defined function is presumed to be defined in the VOC file and cataloged without any prefix.

Examples

The following example defines a user-written function called MYFUNC with the arguments or formal parameters A, B, and C:

```
FUNCTION MYFUNC(A, B, C)
Z = ...
RETURN (Z)
END
```

The next example declares the function `MYFUNC`. It uses the function with the statement T = MYFUNC (X, Y, Z). The actual parameters held in X, Y, and Z are referenced by the formal parameters A, B, and C, so the value assigned to T can be calculated.

```
DEFFUN MYFUNC(X, Y, Z)
T = MYFUNC(X, Y, Z)
END
```

DEL statement

Use the DEL statement to delete a field, value, or subvalue from a dynamic array. The DEL statement works similarly to the `DELETE` function.

Syntax

DEL *dynamic.array* < *field#* [,*value#* [,*subvalue#*]] >

dynamic.array is an expression that evaluates to a dynamic array. If *dynamic.array* evaluates to the null value, null is returned.

field# is an expression that evaluates to the field in *dynamic.array*. *value#* is an expression that evaluates to the value in the field. *subvalue#* is an expression that evaluates to the subvalue in the value. These expressions are called delimiter expressions. The numeric values of the delimiter expressions specify which field, value, or subvalue to delete. The entire position is deleted, including its delimiter characters.

value# and *subvalue#* are optional. If they are equal to 0, the entire field is deleted. If *subvalue#* is equal to 0 and *value#* and *field#* are greater than 0, the specified value in the specified field is deleted. If all three delimiter expressions are greater than 0, only the specified subvalue is deleted.

If any delimiter expression is the null value, the DEL statement fails and the program terminates with a run-time error message.

If a higher-level delimiter expression has a value of 0 when a lower-level delimiter expression is greater than 0, the 0 delimiter is treated as if it were equal to 1. The delimiter expressions are, from highest to lowest: field, value, and subvalue.

If the DEL statement references a subelement of a higher element whose value is the null value, the dynamic array is unchanged. Similarly, if all delimiter expressions are 0, the original string is returned.

Examples

In the following examples a field mark is shown by F, a value mark is shown by V, and a subvalue mark is shown by S.

The first example deletes field 1 and sets Q to `VAL1VSUBV1SSUBV2FFSUBV3SSUBV4`:

```
R="FLD1":@FM:"VAL1":@VM:"SUBV1":@SM:"SUBV2":@FM:@FM:"SUBV3":@SM:"SUBV4"
Q=R
DEL Q<1,0,0>
PRINT Q
```

The next example deletes the first subvalue in field 4 and sets the value of Q to `FLD1FVAL1VSUBV1SSUBV2FFSUBV4`:

```
Q=R
 DEL Q<4,1,1>
```

The next example deletes the second value in field 2 and sets the value of Q to `FLD1FVAL1FFSUBV3SSUBV4`:

```
Q=R
 DEL Q<2,2,0>
```

The next example deletes field 3 entirely and sets the value of Q to `FLD1FVAL1VSUBV1SSUBV2FSUBV3SSUBV4`:

```
Q=R
 DEL Q<3,0,0>
```

The next example deletes the second subvalue in field 4 and sets the value of Q to `FLD1FVAL1VSUBV1SSUBV2FFSUBV3`:

```
Q=R
 DEL Q<4,1,2>
```

DELETE function

Use the `DELETE` function to erase the data contents of a specified field, value, or subvalue and its corresponding delimiter from a dynamic array. The `DELETE` function returns the contents of the dynamic array with the specified data removed without changing the actual value of the dynamic array.

Syntax

DELETE (*dynamic.array, field#*[,*value#*[,*subvalue#*]])

dynamic.array is an expression that evaluates to the array in which the field, value, or subvalue to be deleted can be found. If *dynamic.array* evaluates to the null value, null is returned.

field# is an expression that evaluates to the field in the dynamic array; *value#* is an expression that evaluates to the value in the field; *subvalue#* is an expression that evaluates to the subvalue in the value. The numeric values of the delimiter expressions specify which field, value, or subvalue to delete. The entire position is deleted, including its delimiting characters.

value# and *subvalue#* are optional. If they are equal to 0, the entire field is deleted. If *subvalue#* is equal to 0 and *value#* and *field#* are greater than 0, the specified value in the specified field is deleted. If all three delimiter expressions are greater than 0, only the specified subvalue is deleted.

If any delimiter expression is the null value, the `DELETE` function fails and the program terminates with a run-time error message.

If a higher-level delimiter expression has a value of 0 when a lower-level delimiter is greater than 0, the 0 delimiter is treated as if it were equal to 1. The delimiter expressions are, from highest to lowest: field, value, and subvalue.

If the `DELETE` function references a subelement of a higher element whose value is the null value, the dynamic array is unchanged. Similarly, if all delimiter expressions are 0, the original string is returned.

Examples

In the following examples a field mark is shown by F, a value mark is shown by V, and a subvalue mark is shown by S.

The first example deletes field 1 and sets Q to `VAL1VSUBV1SSUBV2FFSUBV3SSUBV4`:

```
R="FLD1":@FM:"VAL1":@VM:"SUBV1":@SM:"SUBV2":@FM:@FM:"SUBV3":@SM:"SUBV4"
Q=DELETE (R,1)
PRINT Q
```

The next example deletes the first subvalue in field 4 and sets the value of Q to `FLD1FVAL1VSUBV1SSUBV2FFSUBV4`:

```
R="FLD1":@FM:"VAL1":@VM:"SUBV1":@SM:"SUBV2":@FM:@FM:"SUBV3":@SM:"SUBV4"
Q=DELETE (R,4,1,1)
PRINT Q
```

The next example deletes the second value in field 2 and sets the value of Q to `FLD1FVAL1FFSUBV3SSUBV4`:

```
R="FLD1":@FM:"VAL1":@VM:"SUBV1":@SM:"SUBV2":@FM:@FM:"SUBV3":@SM:"SUBV4"
Q=DELETE (R,2,2)
PRINT Q
```

The next example deletes field 3 entirely and sets the value of Q to `FLD1FVAL1VSUBV1SSUBV2FSUBV3SSUBV4`:

```
Q=DELETE (R,3,0,0)
PRINT Q
```

The next example deletes the second subvalue in field 4 and sets the value of Q to `FLD1FVAL1VSUBV1SSUBV2FFSUBV3`:

```
R="FLD1":@FM:"VAL1":@VM:"SUBV1":@SM:"SUBV2":@FM:@FM:"SUBV3":@SM:"SUBV4"
Q=DELETE (R,4,1,2)
PRINT Q
```

DELETE statements

Use the DELETE statements to delete a record from a UniVerse file. If you specify a file variable, the file must be open when the DELETE statement is encountered.

Syntax

```
DELETE [file.variable ,] record.ID [ON ERROR statements]
[LOCKED statements ]
[THEN statements ] [ELSE statements ]
```

```
DELETEU [file.variable ,] record.ID [ON ERROR statements ]
[LOCKED statements ]
 [THEN statements ] [ELSE statements ]
```

file.variable is a file variable from a previous OPEN statement.

record.ID is an expression that evaluates to the record ID of the record to be deleted.

If the file does not exist or is not open, the program terminates and a runtime error results. If you do not specify a file variable, the most recently opened default file is used (see the OPEN statement for more information on default files). If you specify both a file variable and a record ID, you must use a comma to separate them.

If the file is an SQL table, the effective user of the program must have SQL DELETE privilege to delete records in the file. For information about the effective user of a program, see the AUTHORIZATION statement, on page 71.

The record is deleted, and any THEN statements are executed. If the deletion fails, the ELSE statements are executed; any THEN statements are ignored.

If a record is locked, it is not deleted, and an error message is produced. The ELSE statements are not executed.

If either *file.variable* or *record.ID* evaluates to the null value, the DELETE statement fails and the program terminates with a run-time error message.

The DELETEU statement

Use the DELETEU statement to delete a record without releasing the update record lock set by a previous READU statement (see the READ statements, on page 303.

The file must have been previously opened with an OPEN statement. If a file variable was specified in the OPEN statement, it can be used in the DELETEU statement. You must place a comma between the file variable and the record ID expression. If no file variable is specified in the DELETEU statement, the statement applies to the default file. See the OPEN statement, on page 276 for a description of the default file.

The ON ERROR clause

The ON ERROR clause is optional in the DELETE statement. Its syntax is the same as that of the ELSE clause. The ON ERROR clause lets you specify an alternative for program termination when a fatal error is encountered during processing of the DELETE statement.

If a fatal error occurs, and the ON ERROR clause was not specified, or was ignored (as in the case of an active transaction), the following occurs:

- An error message appears.

- Any uncommitted transactions begun within the current execution environment roll back.
- The current program terminates.
- Processing continues with the next statement of the previous execution environment, or the program returns to the UniVerse prompt.

A fatal error can occur if any of the following occur:

- A file is not open.
- *file.variable* is the null value.
- A distributed file contains a part file that cannot be accessed.

If the ON ERROR clause is used, the value returned by the STATUS function, on page 380 is the error number.

The LOCKED clause

The LOCKED clause is optional, but recommended.

The LOCKED clause handles a condition caused by a conflicting lock (set by another user) that prevents the DELETE statement from processing. The LOCKED clause is executed if one of the following conflicting locks exists:

- Exclusive file lock
- Intent file lock
- Shared file lock
- Update record lock
- Shared record lock

If the DELETE statement does not include a LOCKED clause, and a conflicting lock exists, the program pauses until the lock is released.

If a LOCKED clause is used, the value returned by the STATUS function is the terminal number of the user who owns the conflicting lock.

Releasing the record lock

A record lock held by a DELETEU statement can be released explicitly with a RELEASE statement or implicitly with WRITE statements, WRITEV statement, MATWRITE statements, or DELETE statements. The record lock is released when you return to the UniVerse prompt.

Examples

```
OPEN "","MLIST" TO MALIST ELSE STOP
 PRINT "FILE BEFORE DELETE STATEMENT:"
 EXECUTE "COUNT MLIST"
 PRINT
 DELETE MALIST, "JONES"
 PRINT "FILE AFTER DELETE STATMENT:"
 EXECUTE "LIST MLIST"
```

This is the program output:

```
FILE BEFORE DELETE STATEMENT:

 3 records listed.

 FILE AFTER DELETE STATMENT:
```

```
  2 records listed.
```

In the following example, the data portion of the SUBSIDIARIES files is opened to the file variable SUBS. If the file cannot be opened an appropriate message is printed. The record MADRID is read and then deleted from the file. An update record lock had been set and is maintained by the DELETEU statement.

```
OPEN "","SUBSIDIARIES" TO SUBS
 READU REC FROM SUBS, 'MADRID'
 ELSE STOP 'Sorry, cannot open Subsidiaries file'
 DELETEU SUBS, "MADRID”
ELSE STOP 'Sorry, cannot delete Subsidiaries file'
```

DELETELIST statement

Use the DELETELIST statement to delete a select list saved in the &SAVEDLISTS& file.

Syntax

DELETELIST *listname*

listname can evaluate to the form:

record.ID

or:

record.IDaccount.name

record.ID is the name of a select list in the &SAVEDLISTS& file. If you specify *account.name*, the &SAVEDLISTS& file of the specified account is used instead of the local &SAVEDLISTS& file.

If *listname* evaluates to the null value, the DELETELIST statement fails and the program terminates with a run-time error message.

Use the DELETEU statement to maintain an update record lock while performing DELETE statements.

DESCRINFO function

The `DESCRINFO` function returns requested information (*key*) about a variable.

Set the *key* value to 1 to obtain information about the type of variable. Set the *key* value to 2 to obtain the reuse flag of the variable. Any other value is invalid, and will result in the program exiting.

Syntax

DESCRINFO(*key, variable*)

If the *key* value is 1, the return type indicates the following type of variable:

Return value	Type
0	unassigned variable
1	integer
2	numeric
3	string

Return value	Type
4	file
5	array
6	subroutine
7	sequential file
8	GCI descriptor
9	NULL value
10	ODBC descriptor

Example

The following example illustrates the DESCRINFO function.

```
A=1
B="DENVER"
C=10.7

VAL1 = DESCRINFO(1,A)
PRINT VAL1

VAL2 = DESCRINFO(1,B)
PRINT VAL2

VAL3 = DESCRINFO(1,C)
PRINT VAL3
```

This program returns the following results:

```
1
3
2
```

DIGEST function

The DIGEST() function generates a message digest of supplied data. A message digest is the result of a one-way hash function (digest algorithm) performed on the message. Message digest has the unique properties that a slight change in the input will result in a significant difference in the resulting digest. Therefore, the probability of two different messages resulting in the same digest (collision) is very unlikely. It is also virtually impossible to reverse to the original message from a digest. Message digest is widely used for digital signatures and other purposes.

The desired digest algorithm is specified in *algorithm*. Data and its location are specified by *data* and *dataLoc*, respectively. The arrived digest will be put into a dynamic array in *result*. Since digest is short and has a fixed length, it is always put into a string and no file option is provided. The result can be in either binary or hex format.

Note: DIGEST data is arbitrary binary data and may contain UniVerse delimiters. If you do not want the data to contain delimiters, use the ENCODE() function to perform BASE64 encoding.

Syntax

DIGEST(*algorithm, data, dataLoc, result*)

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
algorithm	A string containing the digest algorithm name (uppercase or lowercase). UniVerse 11.2.4+ supports the following algorithms: ▪ MD4 ▪ MD5 ▪ SHA ▪ SHA1 ▪ SHA224 ▪ SHA256 ▪ SHA384 ▪ SHA512 Versions prior to 11.2.4 support MD2, MDC2, and RMD160. These algorithms are no longer supported in later versions.
data	Data or the name of the file containing the data to be digested.
dataLoc	1 - Data in a string (SSL_LOC_STRING) 2 - Data in a file (SSL_LOC_FILE)
result	A string to store the digest result.

Return code status

The following table describes the status of each return code.

Return code	Status
0	Success.
1	Unsupported digest algorithm.
2	The data file cannot be read.
3	Message digest cannot be obtained.
4	Invalid parameters.

DIMENSION statement

Use the DIMENSION statement to define the dimensions of an array variable before referencing the array in the program. For a matrix (a two-dimensional array), use the DIMENSION statement to set the maximum number of rows and columns available for the elements of the array. For a vector (a one-dimensional array), use the DIMENSION statement to set the maximum value of the subscript (the maximum elements) in the array.

Syntax

```
DIM[ENSION] matrix (rows, columns) [, matrix (rows, columns) …]
DIM[ENSION] vector (subscript) [, vector (subscript) …]
```

matrix and *vector* can be any valid variable name. The maximum dimension can be any valid numeric expression. When specifying the two dimensions of a matrix, you must use a comma to separate the row and column expressions. These expressions are called indices.

You can use a single DIMENSION statement to define multiple arrays. If you define more than one array with a DIMENSION statement, you must use commas to separate the array definitions.

The DIMENSION statement declares only the name and size of the array. It does not assign values to the elements of the array. Assignment of values to the elements is done with the MAT statement, MATPARSE statement, MATREAD statements, MATREADU statement, and assignment statements.

The DIMENSION statement in an IDEAL or INFORMATION flavor account is executed at run time. The advantage of the way UniVerse handles this statement is that the amount of memory allocated is not determined until the DIM statement is executed. This means that arrays can be redimensioned at run time.

When redimensioning an array, you can change the maximum number of elements, rows, columns, or any combination thereof. You can even change the dimensionality of an array (that is, from a one-dimensional to a two-dimensional array or vice versa).

The values of the array elements are affected by redimensioning as follows:

- Common elements (those with the same indices) are preserved.
- New elements (those that were not indexed in the original array) are initialized as unassigned.
- Abandoned elements (those that can no longer be referenced in the altered array) are lost, and the memory space is returned to the operating system.

The DIMENSION statement fails if there is not enough memory available for the array. When this happens, the INMAT function is set to a value of 1.

An array variable that is passed to a subroutine in its entirety as an argument in a CALL statement cannot be redimensioned in the subroutine. Each array in a subroutine must be dimensioned once. The dimensions declared in the subroutine DIMENSION statement are ignored, however, when an array is passed to the subroutine as an argument (for more information, see the CALL statement, on page 81).

PICK, IN2, and REALITY flavors

In PICK, IN2, and REALITY flavor accounts, arrays are created at compile time, not run time. Arrays are not redimensionable, and they do not have a zero element. To get the same characteristics in an INFORMATION or IDEAL flavor account, use the STATIC.DIM option of the $OPTIONS statement.

Examples

```
DIM ARRAY(2,2)
 ARRAY(1,1)="KK"
 ARRAY(1,2)="GG"
 ARRAY(2,1)="MM"
 ARRAY(2,2)="NN"
```

In the next example warning messages are printed for the unassigned elements in the matrix. The elements are assigned empty strings as their values.

```
DIM ARRAY(2,3)
 *
 PRINT
 FOR X=1 TO 2
 FOR Y=1 TO 3
 PRINT "ARRAY(":X:",":Y:")", ARRAY(X,Y)
 NEXT Y
```

```
NEXT X
DIM S(3,2)
S(1,1)=1
S(1,2)=2
S(2,1)=3
S(2,2)=4
S(3,1)=5
S(3,2)=6
```

In the next example the common elements are preserved. Those elements that cannot be referenced in the new matrix (S(3,1), S(3,2)) are lost.

```
DIM S(2,2)
 *
 PRINT
 FOR X=1 TO 2
 FOR Y=1 TO 2
 PRINT "S(":X:",":Y:")", S(X,Y)
 NEXT Y
 NEXT X
```

This is the program output:

```
ARRAY(1,1)              KK
 ARRAY(1,2)              GG
 ARRAY(1,3)             Program 'DYNAMIC.DIMENSION':
 Line 12, Variable previously undefined, empty string used.

 ARRAY(2,1)               MM
 ARRAY(2,2)                NN
 ARRAY(2,3)               Program 'DYNAMIC.DIMENSION':
 Line 12, Variable previously undefined, empty string used.

 S(1,1)          1
 S(1,2)          2
 S(2,1)          3
 S(2,2)          4
```

DISABLEDEC statement

Use the `DISABLEDEC` command to turn off decryption on a field or fields you specify.

> **Note:** You cannot disable encryption on a file with WHOLERECORD encryption.

Syntax

DISABLEDEC <*filename*> [, <*multilevel-filename*>], {ALL |<*field_list*>} [ON ERROR <*statements*>]

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
filename	The name of the file on which you want to disable decryption.

Parameter	Description
ALL	If you specify ALL, UniVerse will disable decryption on all encrypted fields of this file.
field_list	A comma-separated list of fields for which you want to disable decryption. Do not enter spaces between the field names.
ON ERROR *statements*	If you specify ON ERROR statements and an error occurs, UniVerse executes the statements following the ON ERROR clause. Otherwise, UniVerse executes the next statement.

Use the `STATUS` function after an DISABLEDEC statement is executed to determine the result of the operation, as follows:

Value	Description
0	Success.
1	Already disabled.
2	General failure.
3	Not an encrypted file.
4	Cannot disable WHOLERECORD encrypted file
5	Not an encrypted field.
6	No disablement information found.
7	Not a valid field in the file.

DISPLAY statement

Use the DISPLAY statement to print data on the screen, regardless of whether a PRINTER ON statement has been executed. The syntax for *print.list* is the same as for PRINT statement.

Syntax

DISPLAY [*print.list*]

The elements of the list can be numeric or character strings, variables, constants, or literal strings; the null value, however, cannot be output. The list can consist of a single expression or a series of expressions separated by commas (,) or colons (:) for output formatting. If no *print.list* is designated, a blank line is output.

Expressions separated by commas are printed at preset tab positions. You can use multiple commas together to cause multiple tabulation between expressions.

Expressions separated by colons are concatenated. That is, the expression following the colon is printed immediately after the expression preceding the colon. To print a list without a LINEFEED and RETURN, end the print list with a colon (:).

The DISPLAY statement works similarly to the CRT statement, on page 117.

Example

```
DISPLAY "This can be used to print something on the"
DISPLAY "terminal while"
DISPLAY "the PRINTER ON statement is in effect."
```

The program output on the terminal is:

```
This can be used to print something on the
terminal while
the PRINTER ON statement is in effect.
```

DIV function

Use the DIV function to calculate the value of the quotient after *dividend* is divided by *divisor*.

The dividend and divisor expressions can evaluate to any numeric value. The only exception is that *divisor* cannot be 0. If either *dividend* or *divisor* evaluates to the null value, null is returned.

Syntax

DIV (*dividend, divisor*)

Example

```
X=100;  Y=25
Z = DIV (X,Y)
PRINT Z
```

This is the program output:

```
4
```

DIVS function

Use the DIVS function to create a dynamic array containing the result of the element-by-element division of two dynamic arrays.

Syntax

DIVS (*array1, array2*)

CALL **-DIVS** (*return.array, array1, array2*)

CALL **!DIVS** (*return.array, array1, array2*)

Each element of *array1* is divided by the corresponding element of *array2* with the result being returned in the corresponding element of a new dynamic array. If elements of *array1* have no corresponding elements in *array2*, *array2* is padded with ones and the *array1* elements are returned. If an element of *array2* has no corresponding element in *array1*, 0 is returned. If an element of *array2* is 0, a run-time error message is printed and a 0 is returned. If either element of a corresponding pair is the null value, null is returned.

If you use the subroutine syntax, the resulting dynamic array is returned as *return.array*.

Example

```
A=10:@VM:15:@VM:9:@SM:4
B=2:@VM:5:@VM:9:@VM:2
PRINT DIVS(A,B)
```

This is the program output:

```
5V3V1S4V0
```

DOWNCASE function

Use the DOWNCASE function to change all uppercase letters in *expression* to lowercase. If *expression* evaluates to the null value, null is returned.

Syntax

DOWNCASE (*expression*)

DOWNCASE is equivalent to OCONV("MCL").

If NLS is enabled, the DOWNCASE function uses the conventions specified by the Ctype category for the Lowercase field of the NLS.LC.CTYPE file to change the letters in *expression*. For more information about the NLS.LC.CTYPE file, see the *NLS Guide*.

Example

```
A="DOWN CASE DOES THIS:     "
PRINT A:DOWNCASE(A)
B="Down Case Does This:    "
PRINT B:DOWNCASE(B)
```

This is the program output:

```
DOWN CASE DOES THIS:    down case does this:
Down Case Does This:   down case does this:
```

DQUOTE function

Use the DQUOTE function to enclose an expression in double quotation marks. If *expression* evaluates to the null value, null is returned (without quotation marks).

Syntax

DQUOTE (*expression*)

Example

```
PRINT DQUOTE(12 + 5) : " IS THE ANSWER."
END
```

This is the program output:

```
"17" IS THE ANSWER.
```

DTX function

Use the DTX function to convert a decimal integer to its hexadecimal equivalent.

size indicates the minimum size which the hexadecimal character string should have. This field is supplemented with zeros if appropriate.

If *number* evaluates to the null value, null is returned. If *size* is the null value, the DTX function fails and the program terminates with a runtime error message.

Syntax

DTX (*number* [,*size*])

Example

```
X = 25
 Y = DTX (X)
 PRINT Y
 Y = DTX (X,4)
 PRINT Y
 END
```

This is the program output:

```
19
 0019
```

EBCDIC function

Use the EBCDIC function to convert each character of expression from its ASCII representation value to its EBCDIC representation value. The EBCDIC and ASCII function perform complementary operations. Data that is not represented in ASCII code produces undefined results.

If *expression* evaluates to the null value, the EBCDIC function fails and the program terminates with a runtime error message.

Syntax

EBCDIC (*expression*)

Example

```
X = 'ABC 123'
 Y = EBCDIC(X)
 PRINT "ASCII", "EBCDIC", " X "
 PRINT "------", "-----", "---"
 FOR I = 1 TO LEN (X)
 PRINT SEQ(X[I,1]) , SEQ(Y[I,1]),X[I,1]
 NEXT I
```

This is the program output:

```
ASCII     EBCDIC     X
------    -----     ---
65        193       A
66        194       B
67        195       C
32        64
49        241       1
50        242       2
51        243       3
```

ECHO statement

Use the ECHO statement to control the display of input characters on the screen.

Syntax

```
ECHO {ON | OFF | expression}
```

If ECHO ON is specified, subsequent input characters are displayed, or echoed, on the screen. If ECHO OFF is specified, subsequent input characters are assigned to the INPUT statement variables but are not displayed on the screen.

The ability to turn off character display is useful when the keyboard is to be used for cursor movement or for entering password information. If *expression* evaluates to true, ECHO is turned ON. If *expression* evaluates to false, ECHO is turned OFF. If *expression* evaluates to the null value, it is treated as false, and ECHO is turned OFF.

Example

```
PROMPT ""
 ECHO OFF
 PRINT "ENTER YOUR PASSWORD"
 INPUT PWORD
 ECHO ON
```

This is the program output:

```
ENTER YOUR PASSWORD
```

ENABLEDEC statement

Use the ENABLEDEC command to activate decryption on a file or fields you specify.

Syntax

```
ENABLEDEC <filename> [, <multilevel-filename>], { ALL |<field_list>}
[ON ERROR <statements>]
```

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
filename	The name of the file on which you want to enable decryption.
ALL	If you specify ALL, UniVerse enables decryption on all encrypted fields of this file.
field_list	A comma-separated list of fields for which you want to enable decryption. Do not enter spaces between the field names.
ON ERROR statements	If you specify ON ERROR statements and an error occurs, UniVerse executes the statements following the ON ERROR clause. Otherwise, UniVerse executes the next statement.

Use the STATUS function after an ENABLEDEC statement is executed to determine the result of the operation, as follows:

Value	Description
0	Success.
1	Already enabled/disabled.
2	DISABLEDEC error.
3	Not an encrypted file.
4	Cannot disable WHOLERECORD encrypted file.
5	Not an encrypted field.
6	No disablement information found.
7	Not a valid field in the file.

ENCODE function

The ENCODE() function performs data encoding on input data.

The function can perform either encoding or decoding, as specified by *action*. The data can either be in the dynamic array, *data*, or in a file whose name is specified in *data*, determined by *dataLoc*.

Syntax

ENCODE(*algorithm, action, data, dataLoc, result, resultLoc*)

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
algorithm	A string containing the encode method name. The three valid values are: ▪ SSL_BASE64 - Base64 encoding of data on one line. ▪ SSL_BASE64_ONELINE - Base64 encoding of data on one line. ▪ URLENCODE - Performs URL encoding or decoding on the data passed to the function according to standard RFC 3986.
action	1 - Encode (SSL_ENCODE) 2 - Decode (SSL_DECODE)
data	Data or the name of the file containing the data to be encoded or decoded.
dataLoc	1 - Data in a string (SSL_LOC_STRING) 2 - Data in a file (SSL_LOC_FILE)
result	Encoded or decoded data or the name of the file storing the processed data.
resultLoc	1 - Result in a string (SSL_LOC_STRING) 2 - Result in a file (SSL_LOC_FILE)

Return code status

The following table describes the status of each return code.

Return code	Status
0	Success.
1	Unsupported algorithm.

Return code	Status
2	Invalid parameters (invalid data or result location type, and so forth.).
3	The data cannot be read.
4	The data cannot be encoded or decoded.

Base 64 encoding is designed to represent arbitrary sequences of octets that do not need to be humanly readable. A 64-character subset of US-ASCII is used, enabling 6-bits to be represented per printable character. The subset has the important property that it is represented identically in all versions of ISO646, including US-ASCII, and all characters in the subset are also represented identically in all versions of EBCDIC. The encoding process represents 24-bit groups of input bits as output strings of 4 encoded characters.

There are two BASE64 encoding modes, default and one-line. In default mode, the encoded output stream must be represented in lines of no more than 76 characters each. All line breaks must be ignored by the decoding process. All other characters not found in the 64-character subset should trigger a warning by the decoding process. In one-line mode, the data is a continuous stream of the allowed ASCII characters without any line breaks.

URL encoding performs encoding or decoding on the data passed to the function according to the RFC 3986 standard. This algorithm changes all characters that need to be encoded to the “percent-escaped” form, such as changing “=” to “%3D” when encoding the data, then back to ASCII characters when decoding.

ENCRYPT function

The `ENCRYPT`() function performs symmetric encryption operations. Various block and stream symmetric ciphers can be called through this function.

Syntax

`ENCRYPT` (*`algorithm, action, data, dataLoc,key, keyLoc, keyAction, salt, IV, result, resultLoc, signAlg`*)

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
algorithm	A string containing the cipher name.

Parameter	Description
action	1 - Encrypt (SSL_ENCRYPT) 2 - Base64 encode after encryption (SSL_ENCRYPT_ENCODE) 3 - Decrypt (SSL_DECRYPT) 4 - Base64 decode before decryption (SSL_DECODE_DECRYPT) 5 - One-line Base64 encode after encryption (SSL_ENCRYPT_ENCODE_A) 6 - One-line Base64 decode before decryption (SSL_DECODE_DECRYPT_A) 11 - NOPAD encryption (SSL_ENCRYPT_NOPAD) 12 - NOPAD Base64 encode after encryption (SSL_ENCRYPT_ ENCODE_NOPAD) 13 - NOPAD Decryption (SSL_DECRYPT_NOPAD) 14- NOPAD Base64 decode before decryption (SSL_DECODE_DECRYPT_NOPAD) 15 - NOPAD one-line Base64 encode after encryption (SSL_ENCRYPT_ENCODE_A_NOPAD) 16 - NOPAD one-line Base64 decode before decryption (SSL_DECODE_DECRYPT_A_NOPAD)
data	Data or the name of the file containing the data to be processed.
dataLoc	1 - Data in a string (SSL_LOC_STRING) 2 - Data in a file (SSL_LOC_FILE)
key	The actual key (password) or file name containing the key.
keyLoc	1 - Key in a string (SSL_LOC_STRING) 2 - Key in file (SSL_LOC_FILE)

Parameter	Description
keyAction	1 - Use actual key (SSL_KEY_ACTUAL) 2 - Derive key from pass phrase (SSL_KEY_DERIVE) 3 - Use actual key compatible with OpenSSL (SSL_KEY_ACTUAL_OPENSSL) or (SSL_KEY_ACTUAL_COMPAT) 4 - Derive key from pass phrase using MD5 algorithm (SSL_KEY_DRIVE_MD5) 5 - Derive key from pass phrase using SHA1 algorithm (SSL_KEY_DERIVE_SHA1) 6 - Derive key from pass phrase using MD2 algorithm (SSL_KEY_DERIVE_MD2) 7 - Unavailable 8 - Derive key from pass phrase using RM0160 algorithm (SSL_KEY_DERIVE_RM0160) 9 - Derive key from pass phrasing using SHA algorithm (SSL_KEY_DERIVE_SHA) 10 - Derive key from pass phrasing using SHA224 algorithm (SSL_KEY_DERIVE_SHA224) 11 - Derive key from pass phrasing using SHA256 algorithm (SSL_KEY_DERIVE_SHA256) 12 - Derive key from pass phrasing using SHA384 algorithm (SSL_KEY_DERIVE_SHA384) 13 - Derive key from pass phrasing using SHA512 algorithm (SSL_KEY_DERIVE_SHA512) **Note:** keyAction 1 and 2 can be used to exchange encrypted data between UniVerse and UniData systems. However, if you want to exchange encrypted data between UniData or UniVerse and third party products such as OpenSSL-based programs, Java programs, or Microsoft.Net programs, you should use keyActions 3-13.
Salt	A string containing the Salt value. You can specify nosalt in this parameter to perform encryption in nosalt mode. In this mode, the `ENCRYPT`() function will not prepend magic data and salt to encrypted data, or will not check for it in decryption. **Note:** If you use the literal string "nosalt" as the salt value, it mimics the -nosalt option for OpenSSL. It is not meant to exchange encrypted data with other third-party products, such as Java, .NET, or PHP.
IV	A string containing IV.
result	The result buffer or the name of the file storing the result.
resultLoc	1 - Result in a string (SSL_LOC_STRING) 2 - Result in a file (SSL_LOC_FILE)
signAlg	Allows users to specify a signing digest algorithm. The value can be any digest algorithm supported by OpenSSL. For example, MD5, SHA1, SHA224, SHA256, SHA384, or SHA512.

Return code status

The following table describes the status of each return code.

Return code	Status
0	Success.
1	Invalid cipher.
2	Invalid parameters (location/action value is out of range, and so forth).
3	The data cannot be read.
4	The key cannot be derived.
5	Base 64 encoding/decoding error.
6	Encryption/decryption error.

If you specify the KeyAction value as 3 (SSL_KEY_ACTUAL_OPENSSL), the key string and IV string must be in hexadecimal format with correct length for the algorithm you specify. You can exchange encrypted data with third-party products.

If you specify the KeyAction value as 2 (SSL_KEY_ACTUAL), a specific salt and algorithm will be used to derive the actual key and IV. The result cannot be exchanged with third-party products.

Ciphers are specified by algorithm and are not case sensitive. Base64 encoding and decoding can be specified with the *action* parameter. If encoding is specified, the encrypted data is Base64 encoded before being entered into *result*. If decoding is specified, the data is Base64 decoded before being encrypted. The data and its location are specified by *data* and *dataLoc*, respectively. *Key* can be explicitly specified or read from a file, or, alternatively, derived on the fly, specified by *keyAction*, in which case the *key* string is used as a pass phrase to derive the actual key. The encrypted or decrypted data is put into the dynamic array *result*, or a file, as specified by *resultLoc*.

Salt is used to provide more security against certain kinds of cryptanalysis attacks, such as dictionary attacks. If an empty *salt* is supplied, an internally generated salt will be used in deriving the key. *Salt* is ignored when *action* is set to decrypt. *IV* (Initialization Vector) is used to provide additional security to some block ciphers. It does not need to be secret but should be fresh, meaning different for each encrypted data. If an actual key is supplied, *IV* is generally needed. However if the encryption key is to be derived from a pass phrase, *IV* is generated automatically. Both *salt* and *IV* must be provided in hexadecimal format.

You have two ways to supply key and IV to the `ENCRYPT()` function. You can supply the actual key and IV, or you can supply a seed (also called a password) and optionally a salt, then let U2 derive the actual key and IV. When you do the latter, you have multiple options to tell U2 how to derive the key and IV, some of which will allow you to exchange encrypted data between UniVerse and third-party products.

Note: Some ciphers are more secure than others. Due to the amount of terminology regarding cryptography in general and SSL in particular, interested readers can refer to the following publications. *Applied Cryptography*, by Bruce Schneier
Internet Cryptography, by Richard E. Smith
SSL and TLS: Designing and Building Secure Systems, by Eric Rescorla

The following ciphers are supported. All cipher names are not case sensitive.

Note: Due to export restrictions, all ciphers may not be available for a specific distribution.

56-bit key DES algorithms

Algorithm	Description
des-cbc	DES in CBC mode
des	Alias for des-cbc
des-cfb	DES in CFB mode

Algorithm	Description
des-ofb	DES in OFB mode
des-ecb	DES in ECB mode

112-bit key DES algorithms

Algorithm	Description
des-ede-cbc	Two key triple DES EDE in CBC mode
des-ede	Alias for des-ede-cbc
des-ede-cfb	Two key triple DES EDE in CFB mode
des-ede-ofb	Two key triple DES EDE in OFB mode

128-bit key AES algorithms

Algorithm	Description
aes-128-cbc	Alias for aes-128
aes-128-ecb	Alias for aes-128

168-bit key DES algorithms

Algorithm	Description
des-ede3-cbc	Three key triple DES EDE in CBC mode
des-ede3	Alias for des-ede3-cbc
des3	Alias for des-ede3-cbc
des-ede3-cfb	Three key triple DES EDE in CFB mode
des-ede3-ofb	Three key triple DES EDE in OFB mode

192-bit AES algorithms

Algorithm	Description
aes-192-cbc	Alias for aes-192
aes-192-ecb	Alias for aes-192

256-bit AES algorithms

Algorithm	Description
aes-256-cbc	Alias for aes-256
aes-256-ecb	Alias for aes-256

RC2 algorithms

Algorithm	Description
rc2-cbc	128-bit RC2 in CBC mode
rc2	Alias for rc2-cbc
rc2-cfb	128-bit RC2 in CBC mode
rc2-ecb	128-bit RC2 in ECB mode
rc2-ofb	128-bit RC2 in OFB mode

Algorithm	Description
rc2-64-cbc	64-bit RC2 in CBC mode
rc2-40-cbc	40-bit RC2 in CBC mode

RC4 algorithms

Algorithm	Description
rc4	128-bit RC4
rc4-40	40-bit RC4

Blowfish algorithms (variable key size, typically 128 bits)

Algorithm	Description
bf	BF
bf-cbc	BF in CBC mode
bf-cfb	BF in CFB mode
bf-ecb	BF in ECB mode
bf-ofb	BF in OFB mode

CAST algorithms (variable key size, typically 128 bits)

Algorithm	Description
cast	CAST
cast-cbc	CAST in CBC mode
cast5-cbc	CAST5 in CBC mode
cast5-cfb	CAST5 in CFB mode
cast5-ecb	CAST5 in ECB mode
cast5-ofb	CAST5 in OFB mode

END statement

Use the END statement to terminate a BASIC program or a section of an IF statement, READ statements, or OPEN statement.

Syntax

END

An END statement is the last statement in a UniVerse BASIC program; it indicates the logical end of the program. When an END statement that is not associated with an IF, READ, or OPEN statement is encountered, execution of the program terminates. You can use comments after the END statement.

You can also use the END statement with conditional statements in the body of a program. In this case END indicates the end of a multistatement conditional clause.

INFORMATION and REALITY flavors

In INFORMATION and REALITY flavors a warning message is printed if there is no final END statement. The END.WARN option of the $OPTIONS statement prints the warning message in IDEAL, IN2, PICK, and PIOPEN flavors under the same conditions.

Example

```
A="YES"
 IF A="YES" THEN
 PRINT "THESE TWO LINES WILL PRINT ONLY"
 PRINT "WHEN THE VALUE OF 'A'    IS 'YES'."
 END
 *
 PRINT
 PRINT "THIS IS THE END OF THE PROGRAM"
 END ; * END IS THE LAST STATEMENT EXECUTED
```

This is the program output:

```
THESE TWO LINES WILL PRINT ONLY
WHEN THE VALUE OF 'A'    IS 'YES'.

THIS IS THE END OF THE PROGRAM
```

END CASE statement

Use the END CASE statement to end a set of CASE statements.

END TRANSACTION statement

Use the END TRANSACTION statement to specify where processing is to continue after a transaction ends.

ENTER statement

Use the ENTER statement to transfer program control from the calling program to an external subroutine without returning to the calling program. The subroutine must have been compiled and cataloged.

Syntax

```
ENTER subroutine

 variable = 'subroutine'
ENTER @variable
```

The ENTER statement is similar to the CALL statement, except that with the ENTER statement, program flow does not return from the entered program to the calling program (see the CALL statement, on page 81). The ENTER statement also does not accept arguments.

In the PIOPEN flavor, the ENTER statement is a synonym for the CALL statement. It takes arguments and returns control to the calling program.

External subroutines can be entered directly or indirectly. To enter a subroutine indirectly, assign the name of the cataloged subroutine to a variable or to an element of an array. Use the name of this variable or array element, prefixed with an at sign (@), as the operand of the ENTER statement.

If *subroutine* evaluates to the null value, the ENTER statement fails and the program terminates with a runtime error message.

Example

The following program transfers control to the cataloged program PROGRAM2:

```
ENTER PROGRAM2
```

EOF(ARG.) function

Use the `EOF(ARG.)` function to check if the command line argument pointer is past the last command line argument. ARG. is part of the syntax of the `EOF(ARG.)` function and must be specified. EOF(ARG.) returns 1 (true) if the pointer is past the last command line argument, otherwise it returns 0 (false).

The *arg#* argument of the GET(ARG.) statement and the SEEK(ARG.) statement affect the value of the `EOF(ARG.)` function.

Syntax

`EOF(ARG.)`

EQS function

Use the `EQS` function to test if elements of one dynamic array are equal to the elements of another dynamic array.

Syntax

`EQS` (*`array1, array2`*)

`CALL` **`-EQS`** (*`return.array, array1, array2`*)

`CALL` **`!EQS`** (*`return.array, array1, array2`*)

Each element of *array1* is compared with the corresponding element of *array2*. If the two elements are equal, a 1 is returned in the corresponding element of a dynamic array. If the two elements are not equal, a 0 is returned. If an element of one dynamic array has no corresponding element in the other dynamic array, a 0 is returned. If either element of a corresponding pair is the null value, null is returned for that element.

If you use the subroutine syntax, the resulting dynamic array returns as *return.array*.

Example

```
A=1:@VM:45:@SM:3:@VM:"one"
 B=0:@VM:45:@VM:1
 PRINT EQS(A,B)
```

This is the program output:

0V1S0V0

EQUATE statement

In an EQUATE statement, *symbol* represents the value of *expression* or *string*. You can use the two interchangeably in the program. When the program is compiled, each occurrence of *symbol* is replaced by the value of *expression* or *string*. The value is compiled as object code and does not have to be reassigned each time the program is executed.

You can define multiple symbols in a single EQUATE statement. *symbol* cannot be a number.

You can define *symbol* only once. Any subsequent EQUATE state generates a compiler error because the compiler interprets the symbol before the statement is parsed.

If you use TO as a connector, the object can be any UniVerse BASIC expression. If you use LIT or LITERALLY as a connector, the object must be a literal string.

RAID does not recognize EQUATE symbols. You must use the object value in RAID sessions.

There is no limit on the number of EQUATE statements allowed by the UniVerse BASIC compiler, except that of memory.

If *symbol* is the same as the name of a BASIC function, the function is disabled in the program. If a statement exists with the same name as a disabled function, the statement is also disabled.

Syntax

EQU[ATE] *symbol* TO *expression* [,*symbol* TO *expression* …]

EQU[ATE] *symbol* LIT[ERALLY] *string* [,*symbol* LIT *string* …]

Examples

In the following example, A is made equivalent to the string JANE:

```
JANE="HI"
 EQUATE A TO "JANE"
 PRINT A
```

Next, B is made equivalent to the variable JANE:

```
JANE="HI"
 EQUATE A TO "JANE"
 EQUATE B LIT "JANE"
 PRINT "A IS EQUAL TO ":A
 PRINT "B IS EQUAL TO ":B
```

This is the program output:

```
A IS EQUAL TO JANE
 B IS EQUAL TO HI
```

In the next example COST is made equivalent to the value of the expression PRICE*QUANTITY:

```
EQUATE COST LIT "PRICE * QUANTITY"
 PRICE=3;QUANTITY=7
 PRINT "THE TOTAL COST IS $": COST
```

This is the program output:

```
THE TOTAL COST IS $21
```

The next example shows an EQUATE statement with multiple symbols:

```
EQUATE C TO "5",
 D TO "7",
 E LIT "IF C=5 THEN PRINT 'YES'"
 PRINT "C+D=": C+D
 E
```

This is the program output:

```
C+D=12
 YES
```

EREPLACE function

Use the EREPLACE function to replace *substring* in *expression* with another substring. If you do not specify *occurrence*, each occurrence of *substring* is replaced.

Syntax

EREPLACE (*expression, substring, replacement* [,*occurrence* [,*begin*]])

occurrence specifies the number of occurrences of *substring* to replace. To replace all occurrences, specify *occurrence* as a number less than 1.

begin specifies the first occurrence to replace. If *begin* is omitted or less than 1, it defaults to 1.

If *substring* is an empty string, *replacement* is prefixed to *expression*. If *replacement* is an empty string, all occurrences of *substring* are removed.

If *expression* evaluates to the null value, null is returned. If *substring*, *replacement*, *occurrence*, or *begin* evaluates to the null value, the EREPLACE function fails and the program terminates with a run-time error message.

The EREPLACE function behaves like the CHANGE function except when *substring* evaluates to an empty string.

Example

```
A = "AAABBBCCCDDDBBB"
 PRINT EREPLACE (A,"BBB","ZZZ")
 PRINT EREPLACE (A,"","ZZZ")
 PRINT EREPLACE (A,"BBB","")
```

This is the program output:

```
AAAZZZCCCDDDZZZ
 ZZZAAABBBCCCDDDBBB
 AAACCCDDD
```

ERRMSG statement

Use the ERRMSG statement to print a formatted error message from the ERRMSG file.

message.ID is an expression evaluating to the record ID of a message to be printed on the screen. Additional expressions are evaluated as arguments that can be included in the error message.

If *message.ID* evaluates to the null value, the default error message is printed:

```
Message ID is NULL: undefined error
```

Syntax

ERRMSG *message.ID* [,*message.ID* …]

A standard Pick ERRMSG file is supplied with UniVerse. Users can construct a local ERRMSG file using the following syntax in the records. Each field must start with one of these codes shown in the following table:

Code	Action
A[(*n*)]	Display next argument left-justified; *n* specifies field length.
D	Display system date.
E [*string*]	Display record ID of message in brackets; *string* displayed after ID.
H [*string*]	Display *string*.
L [(*n*)]	Output a newline; *n* specifies number of newlines.
R [(*n*)]	Display next argument right-justified; *n* specifies field length.
S [(*n*)]	Output *n* blank spaces from beginning of line.
T	Display system time.

Example

```
>ED ERRMSG
17 lines long.
 ----: P0001: HBEGINNING OF ERROR MESSAGE
 0002: L
 0003: HFILE NAMED "
 0004: A


0005: H" NOT FOUND.
 0006: L
 0007: H END OF MESSAGE
 Bottom at line 7
 ----: QOPEN 'SUN.SPORT' TO test
 THEN PRINT "File Opened" ELSE ERRMSG "1", "SUN.SPORT"
```

This is the program output:

```
BEGINNING OF ERROR MESSAGE
 FILE NAMED "SUN.SPORT" NOT FOUND.
 END OF MESSAGE
```

EXCHANGE function

Use the EXCHANGE function to replace one character with another or to delete all occurrences of the specified character.

Syntax

```
EXCHANGE (string, xx, yy)
```

string is an expression evaluating to the string whose characters are to be replaced or deleted. If *string* evaluates to the null value, null is returned.

xx is an expression evaluating to the character to be replaced, in hexadecimal notation.

yy is an expression evaluating to the replacement character, also in hexadecimal notation.

If *yy* is FF, all occurrences of *xx* are deleted. If *xx* or *yy* consist of fewer than two characters, no conversion is done. If *xx* or *yy* is the null value, the EXCHANGE function fails and the program terminates with a run-time error message.

Note: 0x80 is treated as @NULL.STR, not as @NULL.

If NLS is enabled, EXCHANGE uses only the first two bytes of *xx* and *yy* in order to evaluate the characters. Note how the EXCHANGE function evaluates the following characters:

Bytes...	Evaluated as...
00 through FF	00 through FF
00 through FA	Unicode characters 0000 through FA
FB through FE	System delimiters

For more information about character values, see the *NLS Guide*.

Example

In the following example, 41 is the hexadecimal value for the character A and 2E is the hexadecimal value for the period character (.):

```
PRINT EXCHANGE('ABABC','41','2E')
```

This is the program output:

```
.B.BC
```

EXECUTE statement

Use the EXECUTE statement to execute UniVerse commands from within the BASIC program and then return execution to the statement following the EXECUTE statement.

Syntax

```
EXECUTE commands [CAPTURING variable] [PASSLIST [dynamic.array]]
 [RTNLIST [variable]] [{SETTING | RETURNING} variable]
EXECUTE commands [,IN < expression] [,OUT > variable]
 [,SELECT[(list)]< dynamic.array] [,SELECT[(list)] > variable]
 ,PASSLIST [(dynamic.array)]] [,STATUS  > variable]
EXECUTE commands [ ,//IN. < expression] [,//OUT. > variable]
 [,//SELECT.[(list)] < dynamic.array] [,//SELECT.[(list)]
> variable]
 [,//PASSLIST.[(dynamic.array)]] [,//STATUS. > variable]
```

EXECUTE creates a new environment for the executed command. This new environment is initialized with the values of the current prompt, current printer state, Break key counter, the values of inline prompts, KEYEDITs, KEYTRAPs, and KEYEXITs. If any of these values change in the new environment, the changes are not passed back to the calling environment. In the new environment, stacked @variables are either initialized to 0 or set to reflect the new environment. Nonstacked @variables are shared between the EXECUTE and calling environments.

commands can be sentences, paragraphs, verbs, procs, menus, or BASIC programs. You can specify multiple commands in the EXECUTE statement in the same way they are specified in a UniVerse paragraph. Each command or line must be separated by a field mark (ASCII CHAR 254).

The EXECUTE statement has two main syntaxes. The first syntax requires options to be separated by spaces. The second and third syntaxes require options to be separated by commas. In the third syntax, the "//" preceding the keywords and the periods following them are optional; the compiler ignores these marks. Except for the slashes and periods, the second and third syntaxes are the same.

In the first syntax the CAPTURING clause assigns the output of the executed commands to *variable*. The PASSLIST clause passes the current active select list or expression to the commands for use as select list 0. The RTNLIST option assigns select list 0, created by the commands, to *variable*. If you do not specify *variable*, the RTNLIST clause is ignored. Using the SETTING or RETURNING clause causes the @SYSTEM.RETURN.CODE of the last executed command to be placed in *variable*.

In the second syntax the executed commands use the value of *expression* in the IN clause as input. When the IN clause is used, the DATA queue is passed back to the calling program, otherwise data is shared between environments. The OUT clause assigns the output of the commands to *variable*. The SELECT clauses let you supply the select list stored in *expression* as a select list to the commands, or to assign a select list created by the commands to *variable*. If *list* is not specified, select list 0 is used. The PASSLIST clause passes the currently active select list to the commands. If you do not specify *list*, select list 0 in the current program's environment is passed as select list 0 in the executed command's environment. The STATUS clause puts the @SYSTEM.RETURN.CODE of the last executed command in *variable*.

The EXECUTE statement fails and the program terminates with a run-time error message if:

- *dynamic.array* or *expression* in the IN clause evaluates to the null value.
- The command expression evaluates to the null value.

In transactions you can use only the following UniVerse commands and SQL statements with EXECUTE:

- CHECK.SUM
- INSERT
- SEARCH
- SSELECT
- COUNT
- LIST
- SELECT (RetrieVe)
- STAT
- DELETE (SQL)
- LIST.ITEM
- SELECT (SQL)
- SUM
- DISPLAY
- LIST.LABEL

- SORT
- UPDATE
- ESEARCH
- RUN
- SORT.ITEM

INFORMATION flavor

In INFORMATION flavor accounts, the EXECUTE statement without any options is the same as the PERFORM statement. In this case executed commands keep the same environment as the BASIC program that called them. Use the EXEC.EQ.PERF option of the $OPTIONS statement to cause EXECUTE to behave like PERFORM in other flavors.

$OPTIONS PIOPEN.EXECUTE option

Use the PIOPEN.EXECUTE option to make the EXECUTE statement work similarly to the way it works on PI/open systems. The PIOPEN.EXECUTE option lets you use all syntaxes of the EXECUTE statement without creating a new environment for the executed command.

Executed commands keep the same environment as the BASIC program that called them. Unnamed common variables, @variables, and in-line prompts retain their values, and the DATA stack remain active. Select lists also remain active unless they are passed back to the calling program by the RTNLIST clause. If retained values change, the new values are passed back to the calling program.

Output from the CAPTURING clause does not include the trailing field mark, which the standard CAPTURING clause does.

Example

The following example performs a nested SELECT, demonstrating the use of the CAPTURING, RTNLIST, and PASSLIST keywords:

```
CMD = "SELECT VOC WITH TYPE = V"
 EXECUTE CMD RTNLIST VERBLIST1
 CMD = "SELECT VOC WITH NAME LIKE ...LIST..."
 EXECUTE CMD PASSLIST VERBLIST1 RTNLIST VERBLIST2
 CMD = "LIST VOC NAME"
 EXECUTE CMD CAPTURING RERUN PASSLIST VERBLIST2
 PRINT RERUN
```

The program first selects all VOC entries that define verbs, passing the select list to the variable VERBLIST1. Next, it selects from VERBLIST1 all verbs whose names contain the string LIST and passes the new select list to VERBLIST2. The list in VERBLIST2 is passed to the `LIST` command, whose output is captured in the variable RERUN, which is then printed.

EXIT statement

Use the EXIT statement to quit execution of a FOR...NEXT loop or a LOOP...REPEAT loop and branch to the statement following the NEXT or REPEAT statement of the loop. The EXIT statement quits exactly one loop. When loops are nested and the EXIT statement is executed in an inner loop, the outer loop remains in control.

Syntax

EXIT

Example

```
COUNT = 0
 LOOP
 WHILE COUNT < 100 DO
 INNER = 0
 LOOP
 WHILE INNER < 100 DO
 COUNT += 1
 INNER += 1
 IF INNER = 50 THEN EXIT
 REPEAT
 PRINT "COUNT = ":COUNT
 REPEAT
```

This is the program output:

```
COUNT = 50
 COUNT = 100
```

EXP function

Use the EXP function to return the value of "e" raised to the power designated by *expression*. The value of "e" is approximately 2.71828. *expression* must evaluate to a numeric value.

Syntax

EXP (*expression*)

If *expression* is too large or small, a warning message is printed and 0 is returned. If *expression* evaluates to the null value, null is returned.

The formula used by the EXP function to perform the calculations is

value of EXP function = 2.71828**(*expression*)

Example

```
X=5
 PRINT EXP(X-1)
```

This is the program output:

```
54.5982
```

EXTRACT function

Use the EXTRACT function to access the data contents of a specified field, value, or subvalue from a dynamic array. You can use either syntax shown to extract data. The first syntax uses the EXTRACT keyword, the second uses angle brackets.

Syntax

```
EXTRACT (dynamic.array, field#[,value# [,subvalue#]] )
variable < field# [ ,value# [,subvalue#]] >
```

dynamic.array is an expression that evaluates to the array in which the field, value, or subvalue to be extracted is to be found. If *dynamic.array* evaluates to the null value, null is returned.

field# specifies the field in the dynamic array; *value#* specifies the value in the field; *subvalue#* specifies the subvalue in the value. These arguments are called delimiter expressions. The numeric values of the delimiter expressions determine whether a field, a value, or a subvalue is to be extracted. *value#* and *subvalue#* are optional.

Angle brackets used as an EXTRACT function appear on the right side of an assignment statement. Angle brackets on the left side of the assignment statement indicate that a REPLACE function is to be performed (for examples, see the REPLACE function, on page 325).

The second syntax uses angle brackets to extract data from dynamic arrays. *variable* specifies the dynamic array containing the data to be extracted. *field#*, *value#*, and *subvalue#* are delimiter expressions.

Here are the five outcomes that can result from the different uses of delimiter expressions:

Case	Result
Case 1:	If *field#*, *value#*, and *subvalue#* are omitted or evaluate to 0, an empty string is returned.
Case 2:	If *value#* and *subvalue#* are omitted or evaluate to 0, the entire field is extracted.
Case 3:	If *subvalue#* is omitted or specified as 0 and *value#* and *field#* evaluate to nonzero, the entire specified value in the specified field is extracted.
Case 4:	If *field#*, *value#*, and *subvalue#* are all specified and are all nonzero, the specified subvalue is extracted.
Case 5:	If *field#*, *value#*, or *subvalue#* evaluates to the null value, the EXTRACT function fails and the program terminates with a run-time error message.

If a higher-level delimiter expression has a value of 0 when a lower-level delimiter is greater than 0, a 1 is assumed. The delimiter expressions are from highest to lowest: field, value, and subvalue.

If the EXTRACT function references a subelement of an element whose value is the null value, null is returned.

Example

In the following example a field mark is shown by F, a value mark is shown by V, and a subvalue mark is shown by S:

```
VAR=1:@FM:4:@VM:9:@SM:3:@SM:5:@FM:1:@VM:0:@SM:7:@SM:3
Z=EXTRACT(VAR,1,0,0)
PRINT Z
*
Z=VAR<1,1,1>
PRINT Z
*
Z=EXTRACT(VAR,2,1,1)
PRINT Z
*
Z=VAR<3,2,3>
PRINT Z
*
```

```
Z=EXTRACT(VAR,10,0,0)
PRINT Z
*
Z=EXTRACT(VAR,2,2,0)
PRINT Z
*
```

This is the program output:

```
1
 1
 4
 3

 9S3S5
```

FADD function

Use the FADD function to perform floating-point addition on two numeric values. If either number evaluates to the null value, null is returned. If either *number1* or *number2* evaluates to the null value, null is returned. *return.array* equates to *number1* plus *number2.*

This function is provided for compatibility with existing software. You can also use the + operator to perform floating-point addition.

Syntax

```
CALL !FADD (return.array, number1, number2)
```

Example

```
PRINT FADD(.234,.567)
```

This is the program output:

```
0.801
```

FDIV function

Use the FDIV function to perform floating-point division on two numeric values. *number1* is divided by *number2*. *return.array* equates to *number1* divided by *number2*. If *number2* is 0, a runtime error message is produced and a 0 is returned for the function. If either number evaluates to the null value, null is returned.

This function is provided for compatibility with existing software. You can also use the / operator to perform floating-point division.

Syntax

```
FDIV (number1, number2)
CALL !FDIV (return.array, number1, number2)
```

Example

```
PRINT FDIV(.234,.567)
```

This is the program output:

```
0.4127
```

FFIX function

Use the FFIX function to convert a floating-point number to a numeric string with fixed precision. If *number* evaluates to the null value, null is returned.

This function is provided for compatibility with existing software.

Syntax

```
FFIX (number)
```

FFLT function

Use the FFLT function to round a number to a string with a precision of 13. The number also converts to scientific notation when required for precision. If *number* evaluates to the null value, null is returned.

Syntax

```
FFLT (number)
```

FIELD function

Use the FIELD function to return one or more substrings located between specified delimiters in *string*.

Syntax

```
FIELD (string, delimiter, occurrence [,num.substr] )
```

delimiter evaluates to any character, including field mark, value mark, and subvalue marks. It delimits the start and end of the substring. If *delimiter* evaluates to more than one character, only the first character is used. Delimiters are not returned with the substring.

occurrence specifies which occurrence of the delimiter is to be used as a terminator. If *occurrence* is less than 1, 1 is assumed.

num.substr specifies the number of delimited substrings to return. If the value of *num.substr* is an empty string or less than 1, 1 is assumed. When more than one substring is returned, delimiters are returned along with the successive substrings.

If either *delimiter* or *occurrence* is not in the string, an empty string is returned, unless *occurrence* specifies 1. If *occurrence* is 1 and *delimiter* is not found, the entire string is returned. If *delimiter* is an empty string, the entire string is returned.

If *string* evaluates to the null value, null is returned. If *string* contains CHAR(128) (that is, @NULL.STR), it is treated like any other character in a string. If *delimiter*, *occurrence*, or *num.substr* evaluate to the null value, the FIELD function fails and the program terminates with a run-time error message.

The FIELD function works identically to the GROUP function.

Examples

```
D=FIELD("###DHHH#KK","#",4)
 PRINT "D= ",D
```

The variable D is set to DHHH because the data between the third and fourth occurrence of the delimiter # is DHHH.

```
REC="ACADABA"
 E=FIELD(REC,"A",2)
 PRINT "E= ",E
```

The variable E is set to "C".

```
VAR="?"
 Z=FIELD("A.1234$$$$&&",VAR,3)
 PRINT "Z= ",Z
```

Z is set to an empty string since "?" does not appear in the string.

```
Q=FIELD("+1+2+3ABAC","+",2,2)
 PRINT "Q= ",Q
```

Q is set to "1+2" since two successive fields were specified to be returned after the second occurrence of "+".

This is the program output:

```
D=        DHHH
E=        C
Z=
Q=        1+2
```

FIELDS function

Use the `FIELDS` function to return a dynamic array of substrings located between specified delimiters in each element of *dynamic.array*.

Syntax

FIELDS (*dynamic.array, delimiter, occurrence* [,*num.substr*])

CALL **-FIELDS** (*return.array, dynamic.array, delimiter, occurrence, num.substr*)

CALL **!FIELDS** (*return.array, dynamic.array, delimiter, occurrence, num.substr*)

delimiter evaluates to any character, excluding value and subvalue characters. It marks the start and end of the substring. If *delimiter* evaluates to more than one character, the first character is used.

occurrence specifies which occurrence of the delimiter is to be used as a terminator. If *occurrence* is less than 1, 1 is assumed.

num.substr specifies the number of delimited substrings to return. If the value of *num.substr* is an empty string or less than 1, 1 is assumed. In this case delimiters are returned along with the successive substrings. If *delimiter* or *occurrence* does not exist in the string, an empty string is returned, unless

occurrence specifies 1. If *occurrence* is 1 and the specified delimiter is not found, the entire element is returned. If *occurrence* is 1 and *delimiter* is an empty string, an empty string is returned.

If *dynamic.array* is the null value, null is returned. If any element in *dynamic.array* is the null value, null is returned for that element. If *delimiter*, *occurrence*, or *num.substr* evaluates to the null value, the `FIELDS` function fails and the program terminates with a runtime error message.

If you use the subroutine syntax, the resulting dynamic array is returned as *return.array*.

Example

```
A="000-P-0":@VM:"-H--O-":@SM:"N-I-T":@VM:"BC":@SM:"-L-"
PRINT FIELDS(A,"-",2)
```

This is the program output:

```
PVHSIVSL
```

FIELDSTORE function

Use the `FIELDSTORE` function to modify character strings by inserting, deleting, or replacing fields separated by specified delimiters.

Syntax

FIELDSTORE (*string, delimiter, start, n, new.string*)

string is an expression that evaluates to the character string to be modified.

delimiter evaluates to any single ASCII character, including field, value, and subvalue marks.

start evaluates to a number specifying the starting field position. Modification begins at the field specified by *start*. If *start* is greater than the number of fields in *string*, the required number of empty fields is generated before the `FIELDSTORE` function is executed.

n specifies the number of fields of *new.string* to insert in *string*. *n* determines how the `FIELDSTORE` operation is executed. If *n* is positive, *n* fields in *string* are replaced with the first *n* fields of *new.string*. If *n* is negative, *n* fields in *string* are replaced with all the fields in *new.string*. If *n* is 0, all the fields in *new.string* are inserted in *string* before the field specified by *start*.

If *string* evaluates to the null value, null is returned. If *delimiter*, *start*, *n*, or *new.string* is null, the FIELDSTORE function fails and the program terminates with a runtime error message.

Example

```
Q='1#2#3#4#5'
 *
 TEST1=FIELDSTORE(Q,"#",2,2,"A#B")
 PRINT "TEST1= ",TEST1
 *
 TEST2=FIELDSTORE(Q,"#",2,-2,"A#B")
 PRINT "TEST2= ",TEST2
 *
 TEST3=FIELDSTORE(Q,"#",2,0,"A#B")
 PRINT "TEST3= ",TEST3
 *
 TEST4=FIELDSTORE(Q,"#",1,4,"A#B#C#D")
 PRINT "TEST4= ",TEST4
 *
```

```
TEST5=FIELDSTORE(Q,"#",7,3,"A#B#C#D")
PRINT "TEST5= ",TEST5
```

This is the program output:

```
TEST1=    1#A#B#4#5
 TEST2=    1#A#B#4#5
 TEST3=    1#A#B#2#3#4#5
 TEST4=    A#B#C#D#5
 TEST5=    1#2#3#4#5##A#B#C
```

FILEINFO function

Use the FILEINFO function to return information about the specified file's configuration, such as the specified file's parameters, its modulus and load, its operating system file name, and its VOC name. The information returned depends on the file type and the value of the key.

Syntax

FILEINFO (*file.variable* , *key*)

file.variable is the file variable of an open file.

key is a number that indicates the particular information required. These key numbers are described in the Keys and Values Supplied to the FILEINFO Function table.

If the first argument is not a file variable, all keys except 0 return an empty string. A warning message is also displayed. A fatal error results if an invalid key is supplied.

Equate names for keys

An insert file of equate names is provided to let you use mnemonics rather than key numbers. The insert file, called FILEINFO.INS.IBAS, is located in the INCLUDE directory in the UV account directory. It is referenced in PIOPEN flavor accounts through a VOC file pointer called SYSCOM. Use the $INCLUDE statement to insert this file if you want to use equate names, as shown in the example. The following table lists the symbolic name, value, and description:

Symbolic Name	Value	Description
FINFO$IS.FILEVAR	0	1 if *file.variable* is a valid file variable; 0 otherwise.
FINFO$VOCNAME	1	VOC name of the file.
FINFO$PATHNAME	2	Path name of the file.
FINFO$TYPE	3	File type as follows: 1 Static hashed 3 Dynamic hashed 4 Type 1 5 Sequential 7 Distributed and Multivolume
FINFO$HASHALG	4	Hashing algorithm: 2 for GENERAL, 3 for SEQ.NUM.
FINFO$MODULUS	5	Current modulus.
FINFO$MINMODULUS	6	Minimum modulus.
FINFO$GROUPSIZE	7	Group size, in 1-KB units.

Symbolic Name	Value	Description
FINFO$LARGERECORDSIZE	8	Large record size.
FINFO$MERGELOAD	9	Merge load parameter.
FINFO$SPLITLOAD	10	Split load parameter.
FINFO$CURRENTLOAD	11	Current loading of the file (%).
FINFO$NODENAME	12	Empty string, if the file resides on the local system, otherwise the name of the node where the file resides.
FINFO$IS.AKFILE	13	1 if secondary indexes exist on the file; 0 otherwise.
FINFO$CURRENTLINE	14	Current line number.
FINFO$PARTNUM	15	For a distributed file, returns list of currently open part numbers.
FINFO$STATUS	16	For a distributed file, returns list of status codes showing whether the last I/O operation succeeded or failed for each part. A value of –1 indicates the corresponding part file is not open.
FINFO$RECOVERYTYPE	17	1 if the file is marked as recoverable, 0 if it is not. Returns an empty string if recoverability is not supported on the file type (such as type 1 and type 19 files).
FINFO$RECOVERYID	18	Always returns an empty string.
FINFO$IS.FIXED.MODULUS	19	Always returns 0.
FINFO$NLSMAP	20	If NLS is enabled, the file map name, otherwise an empty string. If the map name is the default specified in the `uvconfig` file, the returned string is the map name followed by the name of the configurable parameter in parentheses.
FINFO$ENCRYPTION	22	Returns a dynamic array containing the following information: ▪ For a file encrypted with the WHOLERECORD option: -1@VM<key_id>@VM<algorithm> ▪ For a file encrypted at the field level: <location>@VM<key_id>@VM <algorithm>@VM<field_name>[@FM <location>...@VM<field_name>] ▪ Returns an empty string if the file is not encrypted.
FINFO$REPSTATUS	24	Return values can be: 0 – The file is not published, subscribed, or subwriteable. 1 – The file is being published. 2 – The file is being subscribed. 3 – The file is subwriteable. **Note:** If U2 Data Replication is not running, this function returns 0 for any file used with this function.

Value returned by the STATUS function

If the function executes successfully, the value returned by the `STATUS` function is 0. If the function fails to execute, `STATUS` returns a nonzero value. The following table lists the key, file type, and returned value for *key*:

Key	Dynamic	Directory	Distributed	Sequential
0	1 = file open 0 = file closed	1 = file open 0 = file closed	Dynamic array of codes: 1 = file open 0 = file closed	1 = file open 0 = file closed
1	VOC name	VOC name	VOC name	VOC name
2	File's path name	Path name of file	Dynamic array of complete path names in VOC record order (path name as used in VOC for unavailable files)	File's path name
3	3	4	7	5
4	2 = GENERAL 3 = SEQ.NUM	Empty string	Dynamic array of codes: 2 = GENERAL 3 = SEQ.NUM	Empty string
5	Current modulus	1	Dynamic array of the current modulus of each part file	
6	Minimum modulus	Empty string	Dynamic array of the minimum modulus of each part file	Empty string
7	Group size in disk records	Empty string	Dynamic array of the group size of each part file	Empty string
8	Large record size	Empty string	Dynamic array of the large record size of each part file	Empty string
9	Merge load value	Empty string	Dynamic array of the merge load % of each part file	Empty string

Key	Dynamic	Directory	Distributed	Sequential
10	Split load value	Empty string	Dynamic array of the split load value of each part file **Note:** The values returned for distributed files are dynamic arrays with the appropriate value for each part file. The individual values depend on the file type of the part file. For example, if the part file is a hashed file, some values, such as minimum modulus, have an empty value in the dynamic array for that part file.	Empty string
11	Current load value	Empty string	Dynamic array of the current load value of each part file 1	Empty string
12	Local file: empty string Remote file: node name	Empty string	Dynamic array of values where *value* is: Local file = empty string Remote file = node name	Empty string
13	1 = indexes 2 = no indexes	0	1 = common indexes present 2 = none present	Empty string
15	Empty string	Empty string	Dynamic array of codes in VOC record order. Code is: empty string if part file not open; part number if file is open.	Empty string
16	Empty string	Empty string	Dynamic array of codes in VOC record order for each part file: 0 = I/O operation OK –1 = part file unavailable >0 = error code	Empty string

Note: The first time that an I/O operation fails for a part file in a distributed file, the FILEINFO function returns an error code for that part file. For any subsequent I/O operations on the distributed file with the same unavailable part file, the FILEINFO function returns –1.

NLS mode

The FILEINFO function determines the map name of a file by using the value of FINFO$NLSMAP. NLS uses the insert file called FILEINFO.H. For more information about maps, see the *NLS Guide*.

Examples

In the following example, the file containing the key equate names is inserted with the $INCLUDE statement. The file FILMS is opened and its file type displayed.

```
$INCLUDE SYSCOM FILEINFO.INS.IBAS
 OPEN '','FILMS' TO FILMS
                                    ELSE STOP 'CANT OPEN FILE'
 PRINT FILEINFO(FILMS,FINFO$TYPE)
```

In the next example, the file FILMS is opened and its file type displayed by specifying the numeric key:

```
OPEN '','FILMS' TO FILMS
                                    ELSE STOP 'CANT OPEN FILE'
 PRINT FILEINFO(FILMS,3)
```

FILELOCK statement

Use the FILELOCK statement to acquire a lock on an entire file. This prevents other users from updating the file until the program releases it. A FILELOCK statement that does not specify *lock.type* is equivalent to obtaining an update record lock on every record of the file.

Syntax

```
FILELOCK [file.variable] [, lock.type]
[ON ERROR statements] [LOCKED statements]
```

file.variable specifies an open file. If *file.variable* is not specified, the default file is assumed (for more information on default files, see the OPEN statement, on page 276). If the file is neither accessible nor open, the program terminates with a runtime error message. If *file.variable* evaluates to the null value, the FILELOCK statement fails and the program terminates with a runtime error message.

lock.type is an expression that evaluates to one of the following keywords:

- SHARED (to request an FS lock)
- INTENT (to request an IX lock)
- EXCLUSIVE (to request an FX lock)The ON ERROR clause

The ON ERROR clause is optional in the FILELOCK statement. The ON ERROR clause lets you specify an alternative for program termination when a fatal error is encountered during processing of the FILELOCK statement.

If a fatal error occurs, and the ON ERROR clause was not specified, or was ignored (as in the case of an active transaction), the following occurs:

- An error message appears.
- Any uncommitted transactions begun within the current execution environment roll back.
- The current program terminates.
- Processing continues with the next statement of the previous execution environment, or the program returns to the UniVerse prompt.

A fatal error can occur if any of the following occur:

- A file is not open.
- *file.variable* is the null value.
- A distributed file contains a part file that cannot be accessed.

If the ON ERROR clause is used, the value returned by the STATUS function is the error number. If a FILELOCK statement is used when any portion of a file is locked, the program waits until the file is released.

The LOCKED clause

The LOCKED clause is optional, but recommended. The LOCKED clause handles a condition caused by a conflicting lock (set by another user) that prevents the FILELOCK statement from processing. The LOCKED clause is executed if one of the following conflicting locks exists:

This requested lock...	Conflicts with...
Shared file lock	Exclusive file lock Intent file lock Update record lock
Intent file lock	Exclusive file lock Intent file lock Shared file lock Update record lock
Exclusive file lock	Exclusive file lock Intent file lock Shared file lock Update record lock Shared record lock

If the FILELOCK statement does not include a LOCKED clause and a conflicting lock exists, the program pauses until the lock is released.

If a LOCKED clause is used, the value returned by the STATUS function is the terminal number of the user who owns the conflicting lock.

Releasing locks

A shared, intent, or exclusive file lock can be released by a FILEUNLOCK statement, RELEASE statement, or STOP statement.

Locks acquired or promoted within a transaction are not released when previous statements are processed.

Examples

```
OPEN '','SUN.MEMBER' TO DATA ELSE STOP "CAN'T OPEN FILE"
 FILELOCK DATA LOCKED STOP 'FILE IS ALREADY LOCKED'
 FILEUNLOCK DATA
 OPEN '','SUN.MEMBER' ELSE STOP "CAN'T OPEN FILE"
 FILELOCK LOCKED STOP 'FILE IS ALREADY LOCKED'
 PRINT "The file is locked."
 FILEUNLOCK
```

This is the program output:

```
The file is locked.
```

The following example acquires an intent file lock:

```
FILELOCK fvar, "INTENT" LOCKED
 owner = STATUS()
 PRINT "File already locked by":owner
 STOP
 END
```

FILEUNLOCK statement

Use the FILEUNLOCK statement to release a file lock set by the FILELOCK statement.

Syntax

FILEUNLOCK [*file.variable*] [ON ERROR *statements*]

file.variable specifies a file previously locked with a FILELOCK statement. If *file.variable* is not specified, the default file with the FILELOCK statement is assumed (for more information on default files, see the OPEN statement, on page 276). If *file.variable* evaluates to the null value, the FILEUNLOCK statement fails and the program terminates with a run-time error message.

The FILEUNLOCK statement releases only file locks set with the FILELOCK statement. Update record locks must be released with one of the other unlocking statements (for example, WRITE, WRITEV, and so on).

The ON ERROR clause

The ON ERROR clause is optional in the FILEUNLOCK statement. The ON ERROR clause lets you specify an alternative for program termination when a fatal error is encountered during processing of the FILEUNLOCK statement.

If a fatal error occurs, and the ON ERROR clause was not specified, or was ignored (as in the case of an active transaction), the following occurs:

- An error message appears.
- Any uncommitted transactions begun within the current execution environment roll back.
- The current program terminates.
- Processing continues with the next statement of the previous execution environment, or the program returns to the UniVerse prompt.

A fatal error can occur if any of the following occur:

- A file is not open.

- *file.variable* is the null value.
- A distributed file contains a part file that cannot be accessed.

If the ON ERROR clause is used, the value returned by the STATUS function is the error number. The ON ERROR clause is not supported if the FILEUNLOCK statement is within a transaction.

Example

In the following example, the first FILEUNLOCK statement unlocks the default file. The second FILEUNLOCK statement unlocks the file variable FILE.

```
OPEN '','SUN.MEMBER' ELSE STOP "CAN'T OPEN SUN.MEMBER"
 FILELOCK
 .
 .
 .
 FILEUNLOCK
 OPEN 'EX.BASIC' TO FILE ELSE STOP
 FILELOCK FILE
 .
 .
 .
 FILEUNLOCK FILE
```

FIND statement

Use the FIND statement to locate an element in *dynamic.array*. The field, value, and subvalue positions of *element* are put in the variables *fmc*, *vmc*, and *smc* respectively.

Syntax

```
FINDelement IN dynamic.array [,occurrence] SETTING fmc [,vmc [,smc]]
 {THEN statements [ELSE statements] | ELSE statements}
```

element evaluates to a character string. FIND succeeds only if the string matches an element in its entirety. If *element* is found in *dynamic.array*, any THEN statements are executed. If *element* is not found, or if *dynamic.array* evaluates to the null value, *fmc*, *vmc*, and *smc* are unchanged, and the ELSE statements are executed.

If *occurrence* is unspecified, it defaults to 1. If *occurrence* is the null value, the FIND statement fails and the program terminates with a runtime error message.

Example

```
A="THIS":@FM:"IS":@FM:"A":@FM:"DYNAMIC":@FM:"ARRAY"
FIND "IS" IN A SETTING FM,VM,SM ELSE ABORT
PRINT "FM=",FM
PRINT "VM=",VM
PRINT "SM=",SM
```

This is the program output:

```
FM=       2
VM=       1
```

```
SM=        1
```

FINDSTR statement

Use the FINDSTR statement to locate *substring* in *dynamic.array*. The field, value, and subvalue positions of *substring* are placed in the variables *fmc*, *vmc*, and *smc* respectively.

Syntax

```
FINDSTRsubstring IN dynamic.array [,occurrence]
SETTING fmc [,vmc [,smc]]
 {THEN statements [ELSE statements] | ELSE statements}
```

FINDSTR succeeds if it finds *substring* as part of any element in *dynamic array*. If *substring* is found in *dynamic.array*, any THEN statements are executed. If *substring* is not found, or if *dynamic.array* evaluates to the null value, *fmc*, *vmc*, and *smc* are unchanged, and the ELSE statements are executed.

If *occurrence* is unspecified, it defaults to 1. If *occurrence* is the null value, FINDSTR fails and the program terminates with a runtime error message.

Example

```
A="THIS":@FM:"IS":@FM:"A":@FM:"DYNAMIC":@FM:"ARRAY"
FINDSTR "IS" IN A SETTING FM,VM,SM ELSE ABORT
PRINT "FM=",FM
PRINT "VM=",VM
PRINT "SM=",SM
```

This is the program output:

```
FM=1
 VM=1
 SM=1
```

FIX function

Use the `FIX` function to convert a numeric value to a floating-point number with a specified precision. `FIX` lets you control the accuracy of computation by eliminating excess or unreliable data from numeric results. For example, a bank application that computes the interest accrual for customer accounts does not need to deal with credits expressed in fractions of cents. An engineering application needs to throw away digits that are beyond the accepted reliability of computations.

Syntax

```
FIX (number [,precision [,mode]] )
```

number is an expression that evaluates to the numeric value to be converted.

precision is an expression that evaluates to the number of digits of precision in the floating-point number. If you do not specify *precision*, the precision specified by the PRECISION statement is used. The default precision is 4.

mode is a flag that specifies how excess digits are handled. If *mode* is either 0 or not specified, excess digits are rounded off. If *mode* is anything other than 0, excess digits are truncated.

If *number* evaluates to the null value, null is returned.

Examples

The following example calculates a value to the default precision of 4:

```
REAL.VALUE = 37.73629273
 PRINT FIX (REAL.VALUE)
```

This is the program output:

```
37.7363
```

The next example calculates the same value to two digits of precision. The first result is rounded off, the second is truncated:

```
PRINT FIX (REAL.VALUE, 2)
 PRINT FIX (REAL.VALUE, 2, 1)
```

This is the program output:

```
37.74
 37.73
```

FLUSH statement

The FLUSH statement causes all the buffers for a sequential I/O file to be written immediately. Normally, sequential I/O uses UNIX "stdio" buffering for input/output operations, and writes are not performed immediately.

Syntax

FLUSH *file.variable* {THEN *statements* [ELSE *statements*] | ELSE *statements*}

file.variable specifies a file previously opened for sequential processing. If *file.variable* evaluates to the null value, the FLUSH statement fails and the program terminates with a run-time error message.

After the buffer is written to the file, the THEN statements are executed, and the ELSE statements are ignored. If THEN statements are not present, program execution continues with the next statement.

If the file cannot be written to or does not exist, the ELSE statements are executed; any THEN statements are ignored.

See the OPENSEQ statement, on page 283 and WRITESEQ statement, on page 454 for more information on sequential file processing.

Example

```
OPENSEQ 'FILE.E', 'RECORD1' TO FILE THEN
 PRINT "'FILE.E' OPENED FOR SEQUENTIAL PROCESSING"
 END ELSE STOP
 WEOFSEQ FILE
 *
 WRITESEQ 'NEW LINE' ON FILE THEN
```

```
FLUSH FILE THEN
PRINT "BUFFER FLUSHED"
END ELSE PRINT "NOT FLUSHED"
ELSE ABORT
*
CLOSESEQ FILE
END
```

FMT function

Use the FMT function or a format expression to format data for output. Any BASIC expression can be formatted for output by following it with a format expression.

Syntax

FMT (*expression, format*)*expressionformat*

expression evaluates to the numeric or string value to be formatted.

format is an expression that evaluates to a string of formatting codes. The syntax of the format expression is:

[*width*] [*fill*] *justification* [*edit*] [*mask*]

The format expression specifies the width of the output field, the placement of background or fill characters, line justification, editing specifications, and format masking.

If *expression* evaluates to the null value, null is returned. If *format* evaluates to null, the FMT function and the format operation fail.

width is an integer that specifies the size of the output field in which the value is to be justified. If you specify *mask*, you need not specify *width*. If you do not specify *mask*, *width* is required.

fill specifies the character to be used to pad entries when filling out the output field. *fill* is specified as a single character. The default fill character is a space. If you want to use a numeric character or the letter L, R, T, or Q as a fill character, you must enclose it in single quotation marks.

justification is required in one of the following forms.

Decimal notation:

Value	Description
L	Left justification - Break on field length.
R	Right justification - Break on field length.
T	Text justification - Left justify and break on space.
U	Left justification – Break on field length.
C	Center justification –Break on field length

Exponential notation:

Value	Description
Q	Right justification - Break on field length.
QR	Right justification - Break on field length.
QL	Left justification

edit can be any of the following:

Value	Description
n[*m*]	Used with L, R, or T justification, *n* is the number of digits to display to the right of the decimal point, and *m* descales the value by *m* minus the current precision. Each can be a number from 0 through 9. You must specify *n* in order to specify *m*. If you do not specify *m*, *m* = 0 is assumed. If you do not specify *n*, *n* = *m* = 0 is assumed. Remember to account for the precision when you specify *m*. The default precision is 4. If you specify 0 for *n*, the value is rounded to the nearest integer. If the formatted value has fewer decimal places than *n*, output is padded with zeros to the *n*th decimal place. If the formatted value has more decimal places than *n*, the value is rounded to the *n*th decimal place. If you specify 0 for *m*, the value is descaled by the current precision (0 - current precision).
*n*E*m*	Used with Q, QR, or QL justification, *n* is the number of fractional digits, and *m* specifies the exponent. Each can be a number from 0 through 9.
n.m	Used with Q, QR, or QL justification, *n* is the number of digits preceding the decimal point, and *m* the number of fractional digits. Each can be a number from 0 through 9.
$	Prefixes a dollar sign to the value.
F	Prefixes a franc sign to the value.
,	Inserts commas after every thousand.
Z	Suppresses leading zeros. Returns an empty string if the value is 0. When used with the Q format, only the trailing fractional zeros are suppressed, and a 0 exponent is suppressed.
E	Surrounds negative numbers with angle brackets (< >).
C	Appends cr to negative numbers.
D	Appends db to positive numbers.
B	Appends db to negative numbers.
N	Suppresses a minus sign on negative numbers.
M	Appends a minus sign to negative numbers.
T	Truncates instead of rounding.
Y	In NLS mode, prefixes the yen/yuan character to the value, that is, the Unicode value 00A5. Returns a status code of 2 if you use Y with the MR or ML code. If NLS is disabled or if the Monetary category is not used, Y prefixes the byte value 0xA5.

Note: The E, M, C, D and N options define numeric representations for monetary use, using prefixes or suffixes. In NLS mode, these options override the Numeric and Monetary categories.

mask lets literals be intermixed with numerics in the formatted output field. The mask can include any combination of literals and the following three special format mask characters:

Character	Description
#*n*	Data is displayed in a field of *n* fill characters. A blank is the default fill character. It is used if the format string does not specify a fill character after the width parameter.
%*n*	Data is displayed in a field of *n* zeros.
**n*	Data is displayed in a field of *n* asterisks.

If you want to use numeric characters or any of the special characters as literals, you must escape the character with a backslash (\ ).

A #, %, or * character followed by digits causes the background fill character to be repeated *n* times. Other characters followed by digits cause those characters to appear in the output data *n* times.

mask can be enclosed in parentheses () for clarity. If *mask* contains parentheses, you must include the whole mask in another set of parentheses. For example:

((###) ###-####)

You must specify either *width* or *mask* in the `FMT` function. You can specify both in the same function. When you specify *width*, the string is formatted according to the following rules:

If *string* is smaller than width *n*, it is padded with fill characters.

If *string* is larger than width *n*, a text mark (CHAR(251)) is inserted every *n*th character and each field is padded with the fill character to *width*.

The `STATUS` function reflects the result of *edit* as follows:

Value	Description
0	The edit code is successful.
1	The string expression is invalid.
2	The edit code is invalid.

See the STATUS function, on page 380 for more information.

REALITY flavor

In REALITY flavor accounts, you can use conversion codes in format expressions.

Examples

Format expressions	Formatted value
Z=FMT("236986","R##-##-##")	Z= 23-69-86
X="555666898" X=FMT(X,"20*R2$,")	X= *****$555,666,898.00
Y="DAVID" Y=FMT(Y,"10.L")	Y= DAVID.....
V="24500" V=FMT(V,"10R2$Z")	V= $24500.00
R=FMT(77777,"R#10")	R= 77777
B="0.12345678E1" B=FMT(B,"9*Q")	B= *1.2346E0
PRINT 233779 "R"	233779
PRINT 233779 "R0"	233779
PRINT 233779 "R00"	2337790000
PRINT 233779 "R2"	233779.00
PRINT 233779 "R20"	2337790000.00
PRINT 233779 "R24"	233779.00
PRINT 233779 "R26"	2337.79
PRINT 2337.79 "R"	2337.79

Format expressions	Formatted value
PRINT 2337.79 "R0"	2338
PRINT 2337.79 "R00"	23377900
PRINT 2337.79 "R2"	2337.79
PRINT 2337.79 "R20"	23377900.00
PRINT 2337.79 "R24"	2337.79
PRINT 2337.79 "R26"	23.38

FMTDP function

In NLS mode, use the FMTDP function to format data for output in display positions rather than character lengths.

Syntax

FMTDP (*expression, format* [, *mapname*])

expression evaluates to the numeric or string value to be formatted. Any unmappable characters in *expression* are assumed to have a display length of 1.

format is an expression that evaluates to a string of formatting codes. The syntax of the format expression is:

[*width*] [*fill*] *justification* [*edit*] [*mask*]

The format expression specifies the width of the output field, the placement of background or fill characters, line justification, editing specifications, and format masking. For complete syntax details, see the FMT function, on page 169.

If *format* has a display length greater than 1, and there is only one display position left to fill, FMTDP enters the extra fill character. The returned string can occupy more display positions than you intended.

mapname is the name of an installed map. If *mapname* is not installed, the display positions of the characters in *expression* are used. If any unmappable characters exist in *expression*, the display length is 1, that is, the unmapped character displays as a single unmappable character. If *mapname* is omitted, the map associated with the channel activated by the PRINTER ON statement is used; otherwise, the map associated with the terminal channel (or print channel 0) is used.

You can also specify *mapname* as CRT, AUX, LPTR, and OS. These use the maps associated with the terminal, auxiliary printer, print channel 0, or the operating system, respectively. If you specify *mapname* as NONE, the string is not mapped.

If you execute FMTDP when NLS is disabled, the behavior is the same as for FMT. For more information about display length, see the *UniVerse NLS Guide*.

FMTS function

Use the FMTS function to format elements of *dynamic.array* for output. Each element of the array is acted upon independently and is returned as an element in a new dynamic array.

Syntax

FMTS (*dynamic.array, format*)

CALL **-FMTS** (*return.array, dynamic.array, format*)

```
CALL !FMTS (return.array, dynamic.array, format)
```

format is an expression that evaluates to a string of formatting codes. The syntax of the format expression is:

[*width*] [*background*] *justification* [*edit*] [*mask*]

The format expression specifies the width of the output field, the placement of background or fill characters, line justification, editing specifications, and format masking. For complete syntax details, see the FMT function, on page 169.

If *dynamic.array* evaluates to the null value, null is returned. If *format* evaluates to null, the FMTS function fails and the program terminates with a runtime error message.

If you use the subroutine syntax, the resulting dynamic array is returned as *return.array*.

FMTSDP function

In NLS mode, use the FMTSDP function to format elements of *dynamic.array* for output in display positions rather than character lengths. Each element of the array is acted upon independently and is returned as an element in a new dynamic array. Any unmappable characters in *dynamic.array* are assumed to have a display length of 1.

Syntax

```
FMTSDP (dynamic.array, format [, mapname] )
```

format is an expression that evaluates to a string of formatting codes. The syntax of the format expression is:

[*width*] [*background*] *justification* [*edit*] [*mask*]

The format expression specifies the width of the output field, the placement of background or fill characters, line justification, editing specifications, and format masking. For complete syntax details, see the FMT function, on page 169.

If *format* has a display length greater than 1, and there is only one display position left to fill, FMTSDP enters the extra fill character. The returned string can occupy more display positions than you intend.

mapname is the name of an installed map. If *mapname* is not installed, the display positions of the characters in *dynamic.array* are used. If any unmappable characters exist in *dynamic.array*, the display length is 1, that is, the unmapped character displays as a single unmappable character. If *mapname* is omitted, the map associated with the channel activated by the PRINTER ON statement is used; otherwise, the map associated with the terminal channel (or print channel 0) is used.

You can also specify *mapname* as CRT, AUX, LPTR, and OS. These use the maps associated with the terminal, auxiliary printer, print channel 0, or the operating system, respectively. If you specify *mapname* as NONE, the string is not mapped.

If *dynamic.array* evaluates to the null value, null is returned. If *format* evaluates to null, the FMTSDP function fails and the program terminates with a run-time error message.

Note: If you execute FMTSDP when NLS is disabled, the behavior is the same as for FMTS function.

For more information about display length, see the *UniVerse NLS Guide*.

FMUL function

Use the FMUL function to perform floating-point multiplication on two numeric values. If either number evaluates to the null value, null is returned. *return.array* equates to *number1* multiplied by *number2*.

This function is provided for compatibility with existing software. You can also use the * operator to perform floating-point multiplication.

Syntax

FMUL (*number1, number2*)

CALL !**FMUL** (*return.array, number1, number2*)

Example

```
PRINT FMUL(.234,.567)
```

This is the program output:

```
0.1327
```

FOLD function

Use the FOLD function to divide a string into a number of substrings separated by field marks.

Syntax

FOLD (*string, length*)

CALL !**FOLD** (*subdivided.string, string, length*)

string is separated into substrings of length less than or equal to *length*. *string* is separated on blanks, if possible, otherwise it is separated into substrings of the specified length.

subdivided.string contains the result of the FOLD operation.

If *string* evaluates to the null value, null is returned. If *length* is less than 1, an empty string is returned. If *length* is the null value, the FOLD function fails and the program terminates with a runtime error message.

Examples

```
PRINT FOLD("THIS IS A FOLDED STRING.",5)
```

This is the program output:

```
THISFIS AFFOLDEFDFSTRINFG.
```

In the following example, the blanks are taken as substring delimiters, and as no substring exceeds the specified length of six characters, the output would be:

```
REDFMORANGEFMYELLOWFMGREENFMBLUEFMINDIGOFMVIOLET
```

The field mark *replaces* the space in the string:

```
A="RED ORANGE YELLOW GREEN BLUE INDIGO VIOLET"
 CALL !FOLD (RESULT,A,6)
```

```
PRINT RESULT
```

FOLDDP function

In NLS mode, use the FOLDDP function to divide a string into a number of substrings separated by field marks. The division is in display positions rather than character lengths.

Syntax

FOLDDP (*string, length* [, *mapname*])

string is separated into substrings of display length less than or equal to *length*. *string* is separated on blanks, if possible, otherwise it is separated into substrings of the specified length.

If *string* evaluates to the null value, null is returned. If *length* is less than 1, an empty string is returned. If *length* is the null value, the FOLDDP function fails and the program terminates with a run-time error message.

If you execute FOLDDP when NLS is disabled, the behavior is the same as for the FOLD function. For more information about display length, see the *UniVerse NLS Guide*.

FOOTING statement

Use the FOOTING statement to specify the text and format of the footing to print at the bottom of each page of output.

The ON clause specifies the logical print channel to use for output. *print.channel* is an expression that evaluates to a number from –1 through 255. If you do not use the ON clause, logical print channel 0 is used, which prints to the user's terminal if PRINTER OFF is set (see the PRINTER statement, on page 294). Logical print channel –1 prints the data on the screen, regardless of whether a PRINTER ON statement has been executed.

Syntax

FOOTING [ON *print.channel*] *footing*

footing is an expression that evaluates to the footing text and the control characters that specify the footing's format. You can use the following format control characters, enclosed in single quotation marks, in the footing expression:

Control character	Description
C[*n*]	Prints footing line centered in a field of *n* blanks. If *n* is not specified, centers the line on the page.
D	Prints current date formatted as *dd mmm yyyy*.
G	Inserts gaps to format footings.
I	Resets page number, time, and date for PIOPEN flavor only.
Q	Allows the use of the] ^ and \ characters.
R[*n*]	Inserts the record ID left-justified in a field of *n* blanks.
S	Left-justified, inserted page number.
T	Prints current time and date formatted as *dd mmm yyyy hh:mm:ss*. Time is in 12-hour format with "am" or "pm" appended.

Control character	Description
\	Prints current time and date formatted as *dd mmm yyyy hh:mm:ss*. Time is in 12-hour format with “am” or “pm” appended. Do not put the backslash inside single quotation marks.
L	Starts a new line.
]	Starts a new line. Do not put the right bracket inside single quotation marks.
P[*n*]	Prints current page number right-justified in a field of *n* blanks. The default value for *n* is 4.
^	Prints current page number right-justified in a field of *n* blanks. The default value for *n* is 4. Do not put the caret (^) inside single quotation marks.
N	Suppresses automatic paging.

Two single quotation marks (' ') print one single quotation mark in footing text.

When the program is executed, the format control characters produce the specified results. You can specify multiple options in a single set of quotation marks.

If either *print.channel* or *footing* evaluates to the null value, the FOOTING statement fails and the program terminates with a runtime error message.

Pagination begins with page 1 and increments automatically on generation of each new page or upon encountering the $PAGE statement.

Output to a terminal or printer is paged automatically. Use the N option in either a HEADING statement or a FOOTING statement to turn off automatic paging.

Using] ^ and \ in footings

The characters] ^ and \ are control characters in headings and footings. To use these characters as normal characters, you must use the Q option and enclose the control character in double or single quotation marks. You only need to specify Q once in any heading or footing, but it must appear before any occurrence of the characters] ^ and \.

Formatting the footing text

The control character G (for gap) can be used to add blanks to text in footings to bring the width of a line up to device width. If G is specified once in a line, blanks are added to that part of the line to bring the line up to the device width. If G is specified at more than one point in a line, the blank characters are distributed as *evenly* as possible to those points.

See the following examples, in which the vertical bars represent the left and right margins:

Specification	Result
"Hello there"	|Hello there |
"'G'Hello there"	| Hello there|
"'G'Hello there'G'"	| Hello there |
"Hello'G'there"	|Hello there|
"'G'Hello'G'there'G'"	| Hello there |

The minimum gap size is 0 blanks. If a line is wider than the device width even when all the gaps are 0, the line wraps, and all gaps remain 0.

If NLS is enabled, FOOTING calculates gaps using varying display positions rather than character lengths. For more information about display length, see the *UniVerse NLS Guide*.

Left-justified inserted page number

The control character S (for sequence number) is left-justified at the point where the S appears in the line. Only one character space is reserved for the number. If the number of digits exceeds 1, any text to the right is shifted right by the number of extra characters required.

For example, the statement:

```
FOOTING "This is page 'S' of 100000"
```

results in footings such as:

```
This is page 3 of 100000
 This is page 333 of 100000
 This is page 3333 of 100000
```

INFORMATION flavor

Page number field:

In an INFORMATION flavor account the default width of the page number field is the length of the page number. Use the *n* argument to P to set the field width of the page number. You can also include multiple P characters to specify the width of the page field, or you can include spaces in the text that immediately precedes a P option. For example, 'PPP' prints the page number right-justified in a field of three blanks.

Note: In all other flavors, 'PPP' prints three identical page numbers, each in the default field of four.

Date format:

In an INFORMATION flavor account the default date format is *mm-dd-yy*, and the default time format is 24-hour style.

In PICK, IN2, REALITY, and IDEAL flavor accounts, use the HEADER.DATE option of the $OPTIONS statement to cause HEADING statement, FOOTING statement, and $PAGE statement to behave as they do in INFORMATION flavor accounts.

PIOPEN flavor

Right-Justified Overwriting Page Number:

The control character P (for page) is right-justified at the point at which the P appears in the line. Only one character space is reserved for the number. If the number of digits exceeds 1, literal characters to the left of the initial position are overwritten. Normally you must enter a number of spaces to the left of the P to allow for the maximum page number to appear without overwriting other literal characters. For example, the statement:

```
FOOTING "This is page 'P' of 100000"
```

results in footings such as:

```
This is page 3 of 100000
 This is pag333 of 100000
 This is pa3333 of 100000
```

Resetting the page number and the date:

The control character I (for initialize) resets the page number to 1, and resets the date.

FOR statement

Use the FOR statement to create a FOR…NEXT program loop. A program loop is a series of statements that execute repeatedly until the specified number of repetitions have been performed or until specified conditions are met.

Syntax

```
FOR variable = start TO end [STEP increment]
 [loop.statements]
 [CONTINUE | EXIT]
{WHILE | UNTIL} expression]
 [loop.statements]
 [CONTINUE | EXIT]
 NEXT [variable]
```

variable is assigned the value of *start*, which is the initial value of the counter. *end* is the end value of the counter.

The *loop.statements* that follow the FOR clause execute until the NEXT statement is encountered. Then the counter is adjusted by the amount specified by the STEP clause.

At this point a check is performed on the value of the counter. If it is less than or equal to *end*, program execution branches back to the statement following the FOR clause and the process repeats. If it is greater than *end*, execution continues with the statement following the NEXT statement.

The WHILE condition specifies that as long as the WHILE expression evaluates to true, the loop continues to execute. When the WHILE expression evaluates to false, the loop ends, and program execution continues with the statement following the NEXT statement. If a WHILE or UNTIL expression evaluates to the null value, the condition is false.

The UNTIL condition specifies that the loop continues to execute only as long as the UNTIL expression evaluates to false. When the UNTIL expression evaluates to true, the loop ends and program execution continues with the statement following the NEXT statement.

expression can also contain a conditional statement. As *expression* you can use any statement that takes a THEN or an ELSE clause, but without the THEN or ELSE clause. When the conditional statement would execute the ELSE clause, *expression* evaluates to false; when the conditional statement would execute the THEN clause, *expression* evaluates to true. The LOCKED clause is not supported in this context.

You can use multiple WHILE and UNTIL clauses in a FOR…NEXT loop.

Use the CONTINUE statement within FOR…NEXT to transfer control to the next iteration of the loop, from any point in the loop.

Use the EXIT statement within FOR…NEXT to terminate the loop from any point within the loop.

If STEP is not specified, *increment* is assumed to be 1. If *increment* is negative, the end value of the counter is less than the initial value. Each time the loop is processed, the counter is decreased by the amount specified in the STEP clause. Execution continues to loop until the counter is less than *end*.

The body of the loop is skipped if *start* is greater than *end*, and *increment* is not negative. If *start*, *end*, or *increment* evaluates to the null value, the FOR statement fails and the program terminates with a runtime error message.

Nested loops

You can nest FOR…NEXT loops. That is, you can put a FOR…NEXT loop inside another FOR…NEXT loop. When loops are nested, each loop must have a unique variable name as its counter. The NEXT statement for the inside loop must appear before the NEXT statement for the outside loop.

If you omit the variables in the NEXT statement, the NEXT statement corresponds to the most recent FOR statement. If a NEXT statement is encountered without a previous FOR statement, an error occurs during compilation.

INFORMATION flavor

In an INFORMATION flavor account the FOR variable is checked to see if it exceeds *end* before *increment* is added to it. That means that the value of the FOR variable does not exceed *end* at the termination of the loop. In IDEAL, PICK, IN2, and REALITY flavors the increment is made before the bound checking. In this case it is possible for *variable* to exceed *end*. Use the FOR.INCR.BEF option of the $OPTIONS statement to get IDEAL flavor behavior in an INFORMATION flavor account.

Examples

In the following example, the loop is executed 100 times or until control is transferred by one of the statements in the loop:

```
FOR VAR=1 TO 100
 NEXT VAR
```

Here are more examples of FOR…NEXT loops:

Source code	Program output
FOR X=1 TO 10 PRINT "X= ",X NEXT X	X= 1 X= 2 X= 3 X= 4 X= 5 X= 6 X= 7 X= 8 X= 9 X= 10
FOR TEST=1 TO 10 STEP 2 PRINT "TEST= ":TEST NEXT TEST	TEST= 1 TEST= 3 TEST= 5 TEST= 7 TEST= 9
FOR SUB=50 TO 20 STEP -10 PRINT 'VALUE IS ',SUB NEXT	VALUE IS 50 VALUE IS 40 VALUE IS 30 VALUE IS 20

Source code	Program output
FOR A=1 TO 4 FOR B=1 TO A PRINT "A:B= ",A:B NEXT B NEXT A	A:B= 11 A:B= 21 A:B= 22 A:B= 31 A:B= 32 A:B= 33 A:B= 41 A:B= 42 A:B= 43 A:B= 44
PRINT 'LOOP 1 :' SUM=0 FOR A=1 TO 10 UNTIL SUM>20 SUM=SUM+A*A PRINT "SUM= ",SUM NEXT	LOOP 1 : SUM= 1 SUM= 5 SUM= 14 SUM= 30
PRINT 'LOOP 2 :' * Y=15 Z=0 FOR X=1 TO 20 WHILE Z<Y Z=Z+X PRINT "Z= ",Z NEXT X	LOOP 2 : Z= 1 Z=3 Z= 6 Z= 10 Z= 15

FORMLIST statement

The FORMLIST statement is the same as the SELECT statements.

Syntax

FORMLIST [*variable*] [TO *list.number*] [ON ERROR *statements*]

FSUB function

Use the `FSUB` function to perform floating-point subtraction on two numeric values. *number2* is subtracted from *number1*. If either number evaluates to the null value, null is returned. *result* equates to *number1* minus *number2*.

This function is provided for compatibility with existing software. You can also use the - operator to perform floating-point subtraction.

Syntax

FSUB (*number1, number2*)

CALL !**FSUB** (*result, number1, number2*)

Example

```
PRINT FSUB(.234,.567)
```

This is the program output:

```
-0.333
```

FUNCTION statement

Use the FUNCTION statement to identify a user-written function and to specify the number and names of the arguments to be passed to it. The FUNCTION statement must be the first noncomment line in the user-written function. A user-written function can contain only one FUNCTION statement.

Syntax

FUNCTION [*name*] [([MAT] *variable* [, [MAT] *variable* …])]

name is specified for documentation purposes; it need not be the same as the function name or the name used to reference the function in the calling program. *name* can be any valid variable name.

variable is an expression that passes values between the calling programs and the function. *variables* are the formal parameters of the user-written function. When actual parameters are specified as arguments to a user-written function, the actual parameters are referenced by the formal parameters so that calculations performed in the user-written function use the actual parameters.

Separate *variables* by commas. Up to 254 variables can be passed to a user-written function. To pass an array, you must precede the array name with the keyword MAT. When a user-written function is called, the calling function must specify the same number of variables as are specified in the FUNCTION statement.

An extra variable is hidden so that the user-written function can use it to return a value. An extra variable is retained by the user-written function so that a value is returned by the RETURN (value) statement. This extra variable is reported by the MAP and MAKE.MAP.FILE commands. If you use the RETURN statement in a user-written function and you do not specify a value to return, an empty string is returned by default.

The program that calls a user-written function must contain a DEFFUN statement that defines the user-written function before it uses it. The user-written function must be cataloged in either a local catalog or the system catalog, or it must be a record in the same object file as the calling program.

If the user-defined function recursively calls itself within the function, a DEFFUN statement must precede it in the user-written function.

Examples

The following user-defined function SHORT compares the length of two arguments and returns the shorter:

```
FUNCTION SHORT(A,B)
```

```
AL = LEN(A)
BL = LEN(B)
IF AL < BL THEN RESULT = A ELSE RESULT = B
RETURN(RESULT)
```

The following example defines a function called MYFUNC with the arguments or formal parameters A, B, and C. It is followed by an example of the DEFFUN statement declaring and using the MYFUNC function. The actual parameters held in X, Y, and Z are referenced by the formal parameters A, B, and C so that the value assigned to T can be calculated.

```
FUNCTION MYFUNC(A, B, C)
 Z = ...
 RETURN (Z)
 .
 .
 .
 END

 DEFFUN MYFUNC(X, Y, Z)
 T = MYFUNC(X, Y, Z)
 END
```

generateKey function

The generateKey() function generates a public key cryptography key pair and encrypts the private key. You should then put it into an external key file protected by the provided pass phrase. The protected private key can later be used by UniData and UniVerse SSL sessions (through setPrivateKey()) to secure communication. The public key will not be encrypted.

Syntax

generateKey(*privKey, pubKey, format, keyLoc, algorithm, keyLength, passPhrase, paramFile*)

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
privKey	A string storing the generated private key or name of the file storing the generated private key.
pubKey	A string storing the generated public key or name of the file storing the generated public key.
format	1 - PEM(SSL_FMT_PEM) 2 - DER (SSL_FMT_DER)
keyLoc	1 - Put the key into string privKey/pubKey. (SSL_LOC_STRING) 2 - Put the key into a file. (SSL_LOC_FILE)
algorithm	Flag 1- RSA key (SSL_KEY_RSA) 2- DSA key (SSL_KEY_DSA)
keyLength	Number of bits for the generated key. Between 512 and 16384.
passPhrase	A string storing the *passPhrase* to protect the private key.

Parameter	Description
paramFile	A parameter file needed by DSA key generation.

Return code status

The following table describes the status of each return code.

Return code	Status
0	Success.
1	Key pair cannot be generated.
2	Unrecognized key file format.
3	Unrecognized encryption algorithm.
4	Unrecognized key type or invalid key length (must be between 512 and 16384).
5	Empty pass phrase.
6	Invalid DSA parameter file.
7	Random number generator cannot be seeded properly.
8	Private key cannot be written.

The generated private key will be in PKCS #8 form and is encoded in either PEM or DER format specified by *format*. The generated public key is in standard form. If *keyLoc* is 1 (SSL_LOC_STRING), the resulting keys are put into dynamic arrays, *privKey* and *pubKey*, respectively. Otherwise they are put into OS level files specified by *privKey* and *pubKey*.

This function can generate two types of keys, RSA and DSA, specified by *algorithm*. The key length is determined by *keyLength* and must be in the range of 512 to 16384.

For DSA key generation, *paramFile* must be specified. If a parameter file is provided through *paramFile* and it contains valid parameters, the parameters are used to generate a new key pair. If the specified file does not exist or does not contain valid parameters, a new group of parameters will be generated and subsequently used to generate a DSA key pair. The generated parameters are then written to the specified parameter file. Since DSA parameter generation is time consuming, it is recommended that a parameter file be used to generate multiple DSA key pairs.

To make sure the private key is protected, a pass phrase must be provided. A one-way hash function will be used to derive a symmetric key from the pass phrase to encrypt the generated key. When installing the private key into a security context with the `setPrivateKey`() function, or generating a certificate request with the `generateCertRequest`() function, this pass phrase must be supplied to gain access to the private key.

GES function

Use the `GES` function to test if elements of one dynamic array are greater than or equal to corresponding elements of another dynamic array.

Syntax

```
GES (array1, array2)
CALL -GES (return.array, array1, array2)
CALL !GES (return.array, array1, array2)
```

Each element of *array1* is compared with the corresponding element of *array2*. If the element from *array1* is greater than or equal to the element from *array2*, a 1 is returned in the corresponding element of a new dynamic array. If the element from *array1* is less than the element from *array2*, a 0 is returned. If an element of one dynamic array has no corresponding element in the other dynamic array, the undefined element is evaluated as empty, and the comparison continues.

If either element of a corresponding pair is the null value, null is returned for that element.

If you use the subroutine syntax, the resulting dynamic array is returned as *return.array*.

GET statements

Use GET statements to read a block of data from an input stream associated with a device, such as a serial line or terminal. The device must be opened with the OPENDEV statement or OPENSEQ statement. Once the device has been opened, the GET statements read data from the device. The GET statements do not perform any pre- or postprocessing of the data stream, nor do they control local echo characteristics. These aspects of terminal control are handled either by the application or by the device driver. The behavior of certain devices can be managed through the TTYSET/TTYGET interface.

Syntax

```
GET[X] read.var[, length] [SETTING read.count] FROM device
[UNTIL eop.char.list ] [RETURNING last.char.read ]
  [WAITING seconds ] [THEN statements ] [ELSE statements]
```

Note: The WAITING clause is not supported on Windows NT.

Use the GETX statement to return the characters in ASCII hexadecimal format. For example, the sequence of 8-bit character “abcde” is returned as the character string “6162636465”. However, the value returned in the *last.char.read* variable is in standard ASCII character form.

read.var is the variable into which the characters read from *device* are stored. If no data is read, *read.var* is set to the empty string.

length is the expression evaluating to the number of characters read from the data stream; if *length* and *timeout* are not specified, the default length is 1. If *length* is not specified, but an *eop.char.list* value is included, no length limit is imposed on the input.

read.count is the variable that records the actual count of characters read and stored in *read.var*. This may differ from *length* when a timeout condition occurs or when a recognized end-of-packet character is detected.

device is a valid file variable resulting from a successful OPENDEV or OPENSEQ statement. This is the handle to the I/O device that supplies the data stream for the operation of the GET statements.

eop.char.list is an expression that evaluates to a recognized end-of-packet delimiters. The GET operation terminates if a valid end-of-packet character is encountered in the data stream before the requested number of characters is read.

last.char.read is a variable that stores the last character read by the GET operation. If no data is read, *read.var* is set to the empty string. If the input terminated due to the maximum number of characters being read or because of a timeout condition, an empty string is returned.

seconds specifies the number of seconds the program should wait before the GET operation times out.

Terminating conditions

GET statements read data from the device's input stream until the first terminating condition is encountered. The following table lists the possible terminating conditions:

Condition	Description
Requested read length has been satisfied	The read is fully satisfied. *read.var* contains the characters read, and *last.char.read* contains an empty string. Program control passes to the THEN clause if present. The default requested read length is one character unless an end-of-packet value has been selected (in which case, no length limit is used).
Recognized end-of-packet character has been processed	The read is terminated by a special application-defined character in the data stream. The data read to this point, excluding the end-of-packet character, is stored in *read.var*. The end-of-packet character is stored in *last.char.read*. Program control passes to the THEN clause if present. This terminating condition is only possible if the UNTIL clause has been specified. If there is no UNTIL clause, no end-of-packet characters are recognized.
Timeout limit has expired	The read could not be satisfied within the specified time limitation. If no characters have been read, *read.var* and *last.char.read* are set to the empty string, and *read.count* is set to 0. The system status code is set to 0 and may be checked with the STATUS function. Control passes to the ELSE clause if present. This condition is only possible if the WAITING clause is specified. In the absence of a WAITING clause, the application waits until one of the other terminating conditions is met.
Device input error	An unrecoverable error occurred on the device. Unrecoverable errors can include EOF conditions and operating system reported I/O errors. In this case, the data read to this point is stored in *read.var*, and the empty string is stored in *last.char.read*. If no characters have been read, *read.var* and *last.char.read* are set to the empty string, and *read.count* is set to 0. The system status code is set to a nonzero value and may checked with the `STATUS` function. Control passes to the ELSE clause if present.

Note: Under all termination conditions, *read.count* is set to the number of characters read from the input data stream.

THEN and ELSE clauses

For GET statements, the THEN and ELSE clauses are optional. They have different meanings and produce different results, depending on the conditions specified for terminal input.

The following rules apply only if the THEN or ELSE clauses are specified:

- If the UNTIL clause is used without a WAITING clause or an expected length, the GET statement behaves normally. The program waits indefinitely until a termination character is read, then executes the THEN clause. The ELSE clause is never executed.
- If the WAITING clause is used, the GET statement behaves normally, and the ELSE clause is executed only if the number of seconds for timeout has elapsed. If the input terminates for any other reason, it executes the THEN clause.
- If the WAITING clause is not used and there is a finite number of characters to expect from the input, then only the type-ahead buffer is examined for input. If the type-ahead buffer contains the

expected number of characters, it executes the THEN clause; otherwise it executes the ELSE clause. If the type-ahead feature is turned off, the ELSE clause is always executed.

- In a special case, the ELSE clause is executed if the line has not been attached before executing the GET statement.

In summary, unless the WAITING clause is used, specifying the THEN and ELSE clauses causes the GET statement to behave like an INPUTIF …FROM statement. The exception to this is the UNTIL clause without a maximum length specified, in which case the GET statement behaves normally and the ELSE clause is never used.

Example

The following code fragment shows how the GET statement reads a number of data buffers representing a transaction message from a device:

```
DIM SAVEBUFFER(10)
 SAVELIMIT = 10
 OPENDEV "TTY10" TO TTYLINE ELSE STOP "CANNOT OPEN TTY10"
 I = 1
 LOOP
 GET BUFFER,128 FROM TTYLINE UNTIL CHAR(10) WAITING 10
 ELSE
 IF STATUS()
 THEN PRINT "UNRECOVERABLE ERROR DETECTED ON DEVICE,
 "IM SAVEBUFFER(10)
 SAVELIMIT = 10
 OPENDEV "TTY10" TO TTYLINE ELSE STOP "CANNOT OPEN TTY10"
 I = 1
 LOOP
 GET BUFFER,128 FROM TTYLINE UNTIL CHAR(10)
 WAITING 10
 ELSE
 IF STATUS()
 THEN PRINT "UNRECOVERABLE ERROR DETECTED ON DEVICE,":
 ELSE PRINT "DEVICE TIMEOUT HAS OCCURRED, ":
 PRINT "TRANSACTION CANNOT BE COMPLETED."
 STOP
 END
 WHILE BUFFER # "QUIT" DO
 IF I > SAVELIMIT
 THEN
 SAVELIMIT += 10
 DIM SAVEBUFFER(SAVELIMIT)
 END
 SAVEBUFFER(I) = BUFFER
 I += 1
 REPEAT
```

getCipherSuite function

The `getCipherSuite`() function obtains information about supported cipher suites, their version, usage, strength, and type for the specified security *context*. The result is put into the dynamic array *ciphers*, with one line for each cipher suite, separated by a field mark (@FM).

Syntax

`getCipherSuite`(*`context,ciphers`*)

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
context	The security context handle.
ciphers	A dynamic array containing the cipher strings delimited by @FM.

Return code status

The following table describes the status of each return code.

Return code	Status
0	Success.
1	Invalid security context handle.
2	Unable to obtain information.

The format of the string for one cipher suite is as follows:

Suite, version, key-exchange, authentication, encryption, digest, export

Refer to the cipher tables in setCipherSuite function, on page 347 for definitions of all suites. The following is an example of a typical suite.

```
EXP-DES-CBC-SHA SSLv3 Kx=RSA(512) Au=RSA Enc=DES(40) Mac=SHA1
export
```

The suite name is EXP-DES-CBC-SHA. It is specified by SSLv3. The Key-exchange algorithm is RSA with 512-bit key. The authentication is also done by RSA algorithm. The data encryption uses DES (Data Encryption Standard, an NIST standard) with CBC mode. MAC (Message Authentication Code, a hash method to calculate message digest) will be done with SHA-1 (Secure Hash Algorithm 1, also an NIST standard) algorithm. The suite is exportable.

Only those methods that are active for the protocol will be retrieved.

getIpv

Use the `getIpv` function to display the current IPv setting on the whole system or a particular network's connection.

Syntax

`getIpv` `([networkexpr])`

With no arguments, `getIpv` returns the current IPv setting. For *networkexpr*, enter either "socket" or "uvnet" to view only that particular network's connection displays.

> **Note:** If you opened a server socket with "", the server socket will listen on 0.0.0.0 using IPv6 and is able to accept connection from IPv4 and IPv6 clients. If the server socket is bound to a particular address, the client connection must match the exact server network address (DNS domain or otherwise) and use the same IPv setting as well.

GETX statement

Use the GETX statement to read a block of data from an input stream and return the characters in ASCII hexadecimal format.

For details, see the GET statements, on page 184.

GET(ARG.) statement

Use the GET(ARG.) statement to retrieve the next command line argument. The command line is delimited by blanks, and the first argument is assumed to be the first word after the program name. When a cataloged program is invoked, the argument list starts with the second word in the command line.

Syntax

GET(ARG. [,*arg#*]) *variable* [THEN *statements*] [ELSE *statements*]

Blanks in quoted strings are not treated as delimiters and the string is treated as a single argument. For example, "54 76" returns 54 76.

arg# specifies the command line argument to retrieve. It must evaluate to a number. If *arg#* is not specified, the next command line argument is retrieved. The retrieved argument is assigned to *variable*.

THEN and ELSE statements are both optional. The THEN clause is executed if the argument is found. The ELSE clause is executed if the argument is not found. If the argument is not found and no ELSE clause is present, *variable* is set to an empty string.

If no *arg#* is specified or if *arg#* evaluates to 0, the argument to the right of the last argument retrieved is assigned to *variable*. The GET statement fails if *arg#* evaluates to a number greater than the number of command line arguments or if the last argument has been assigned and a GET with no *arg#* is used. To move to the beginning of the argument list, set *arg#* to 1.

If *arg#* evaluates to the null value, the GET statement fails and the program terminates with a run-time error message.

Example

In the following example, the command is:

```
RUN BP PROG ARG1 ARG2 ARG3
```

and the program is:

```
A=5;B=2
 GET(ARG.)FIRST
 GET(ARG.,B)SECOND
 GET(ARG.)THIRD
 GET(ARG.,1)FOURTH
 GET(ARG.,A-B)FIFTH
 PRINT FIRST
PRINT SECOND
 PRINT THIRD
 PRINT FOURTH
 PRINT FIFTH
```

This is the program output:

```
ARG1
 ARG2
 ARG3
 ARG1
 ARG3
```

If the command line is changed to RUN PROG, the system looks in the file PROG for the program with the name of the first argument. If PROG is a cataloged program, the command line would have to be changed to PROG ARG1 ARG2 ARG3 to get the same results.

getHTTPDefault function

The `getHTTPDefault` function returns the default values of the HTTP settings. See the section under `setHTTPDefault` for additional information.

Syntax

`getHTTPDefault``(`*`option, value`*`)`

option The following options are currently defined:

```
PROXY_NAME
PROXY_PORT
VERSION
BUFSIZE
AUTHENTICATE
HEADERS
```

value is a string containing the appropriate option value.

Return codes

The following table describes the status of each return code.

Return code	Status
0	Success.
1	Invalid option.

GETLIST statement

Use the GETLIST statement to activate a saved select list so that a READNEXT statement can use it.

Syntax

```
GETLIST listname [TO list.number] [SETTING variable]
{THEN statements [ELSE statements] | ELSE statements}
```

listname is an expression that evaluates to the form:

record.ID

or:

record.IDaccount.name

record.ID is the record ID of a select list in the &SAVEDLISTS& file. If *account.name* is specified, the &SAVEDLISTS& file of that account is used instead of the one in the local account.

If *listname* evaluates to the null value, the GETLIST statement fails and the program terminates with a run-time error message.

The TO clause puts the list in a select list numbered 0 through 10. If *list.number* is not specified, the list is saved as select list 0.

The SETTING clause assigns the count of the elements in the list to *variable*. The system variable @SELECTED is also assigned this count whether or not the SETTING clause is used. If the list is retrieved successfully, even if the list is empty, the THEN statements execute; if not, the ELSE statements execute.

PICK, REALITY, and IN2 flavors

PICK, REALITY, and IN2 flavor accounts store select lists in list variables instead of numbered select lists. In those accounts, and in programs that use the VAR.SELECT option of the $OPTIONS statement, the syntax of the GETLIST statement is:

GETLIST *listname* [TO *list.variable*] [SETTING *variable*] {THEN *statements* [ELSE *statements*] | ELSE *statements*}

GETLOCALE function

In NLS mode use the `GETLOCALE` function to return the names of specified categories of the current locale. The `GETLOCALE` function also returns the details of any saved locale that differs from the current one.

Syntax

`GETLOCALE` `(`*`category`*`)`

category is one of the following tokens that are defined in the UniVerse include file UVNLSLOC.H:

Category	Description
UVLC$ALL	The names of all the current locale categories as a dynamic array. The elements of the array are separated by field marks. The categories are in the order Time, Numeric, Monetary, Ctype, and Collate.
UVLC$SAVED	A dynamic array of all the saved locale categories.
UVLC$TIME	The setting of the Time category.
UVLC$NUMERIC	The setting of the Numeric category.
UVLC$MONETARY	The setting of the Monetary category.
UVLC$CTYPE	The setting of the Ctype category.
UVLC$COLLATE	The setting of the Collate category.

If the `GETLOCALE` function fails, it returns one of the following error tokens:

Error token	Description
LCE$NO.LOCALES	UniVerse locales are not enabled.
LCE $BAD.CATEGORY	Category is invalid.

For more information about locales, see the *UniVerse NLS Guide*.

GETREM function

Use the GETREM function after the execution of a REMOVE statement, a REMOVE function, or a REVREMOVE statement, to return the numeric value for the character position of the pointer associated with *dynamic.array*.

Syntax

GETREM (*dynamic.array*)

dynamic.array evaluates to the name of a variable containing a dynamic array.

The returned value is an integer. The integer returned is one-based, not zero-based. If no REMOVE statements have been executed on *dynamic.array*, 1 is returned. At the end of *dynamic.array*, GETREM returns the length of dynamic array plus 1. The offset returned by GETREM indicates the first character of the next dynamic array element to be removed.

Example

```
DYN = "THIS":@FM:"HERE":@FM:"STRING"
 REMOVE VAR FROM DYN SETTING X
 PRINT GETREM(DYN)
```

This is the program output:

```
5
```

getSocketErrorMessage function

Use the getSocketErrorMessage() function to translate an error code into a text error message.

This function works with all socket functions. The return status of those functions can be passed into this function to get the corresponding error message.

Syntax

getSocketErrorMessage(*errCode, errMsg*)

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
errCode	The status return code sent by the socket functions.
errMsg	A string containing corresponding error text.

Return codes

The following table describes the return code of each mode.

Return code	Description
0	Success.

Return code	Description
1	Invalid error code.

getSocketInformation function

Use the `getSocketInformation`() function to obtain information about a socket connection.

Syntax

`getSocketInformation`(*`socket_handle, self_ or_ peer, socket_info`*)

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
socket_handle	A handle to the open socket.
self_ or_ peer	Get information on the self end or the peer end of the socket. Specify 0 to return information from the peer end and non-zero for information from the self end.
socket_info	Dynamic array containing information about the socket connection. For information about the elements of this dynamic array, see the following table.

Elements

The following table describes each element of the *socket_info* dynamic array.

Element	Description
1	Open or closed
2	Name or IP
3	Port number
4	Secure or nonsecure
5	Blocking mode

Return codes

The following table describes the status of each return code.

Return codes	Status
0	Success.
Non-zero	See Socket function error return codes, on page 584.

getSocketMap function

The `getSocketMap`() function gets the NLS map associated with the input socket handle with the input socket handle *aSocket*.

Syntax

getSocketMap(*aSocket, mapname*)

aSocket is the socket handle from openSocket() or acceptConnection(), or 0. If *aSocket* is not 0, getSocketMap gets the NLS map associated with the input socket handle. If *aSocket* is 0, it gets the current default NLS map.

getSocketOptions function

The getSocketOptions() function gets the current value for a socket option associated with a socket of any type.

Syntax

getSocketOptions(*socket_handle, options*)

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
socket_handle	The socket handle from openSocket(), acceptSocket(), or initServerSocket().
options	A dynamic array containing information about the socket options and their current settings. When querying for options, the dynamic array is configured as: *optName1*<FM> *optName2*<FM> *optName*... When the options are returned, the dynamic array is configured as: *optName1*<VM>*optValue1a*[<VM>*optValue1b*]<FM> *optName2*<VM>*optValue2a*[<VM>*optValue2b*]<FM> *optName3*... Where *optName* is specified by the caller and must be an option name string listed below. For all options other than LINGER, the first *optValue* specifies whether the option is ON or OFF and must be one of two possible values: “1” for ON or “2” for OFF. The second *optValue* is optional and can hold additional data for a specific option. For the LINGER option, the first value will be zero for OFF and non-zero for ON. The second *optValue* is the timeout value, which is the number of time units to wait before closing the socket. The timeout value's unit type (seconds, milliseconds, and so forth) is dependent on the implementation of the SELECT() function on your operating system.

Available options

The following table describes the available options (case-sensitive) for getSocketOptions().

Option	Description
DEBUG	Enable/disable recording of debug information.

Option	Description
REUSEADDR	Enable/disable the reuse of a location address (default).
KEEPALIVE	Enable/disable keeping connections alive.
DONTROUTE	Enable/disable routing bypass for outgoing messages.
LINGER	Linger on close if data is present.
BROADCAST	Enable/disable permission to transmit broadcast messages.
OOBINLINE	Enable/disable reception of out-of-band data in band.
SNDBUF	Get buffer size for output (default 4KB).
RCVBUF	Get buffer size for input (default 4KB).
TYPE	Get the type of the socket. Refer to the `socket.h` file for more information.
ERROR	Get and clear error on the socket.

Return codes

The following table describes the status of each return code.

Return code	Description
0	Success.
Non-zero	See Socket function error return codes, on page 584.

GOSUB statement

Use the GOSUB statement to transfer program control to an internal subroutine referenced by *statement.label*. A colon (:) is optional in GOSUB statements, even though it is required after nonnumeric statement labels at the beginning of program lines.

Syntax

GOSUB *statement.label* [:]

GO SUB *statement.label* [:]

Use the RETURN statement at the end of the internal subroutine referenced by the GOSUB statement, to transfer program control to the statement following the GOSUB statement.

Use the RETURN TO statement at the end of an internal subroutine to transfer control to a location in the program other than the line following the GOSUB statement. In this case, use *statement.label* to refer to the target location.

Be careful with the RETURN TO statement, because all other GOSUBs or CALLs active when the GOSUB is executed remain active, and errors can result.

A program can call a subroutine any number of times. A subroutine can also be called from within another subroutine; this process is called nesting subroutines. You can nest up to 256 GOSUB calls.

Subroutines can appear anywhere in the program but should be readily distinguishable from the main program. To prevent inadvertent entry into the subroutine, precede it with a STOP statement, END statement, or GOTO statement that directs program control around the subroutine.

Example

```
VAR='ABKL1234'
```

```
 FOR X=1 TO LEN(VAR)
 Y=VAR[X,1]
 GOSUB 100
 NEXT X
 STOP
 100*
 IF Y MATCHES '1N' THEN RETURN TO 200
 PRINT 'ALPHA CHARACTER IN POSITION ',X
 RETURN
 200*
PRINT 'NUMERIC CHARACTER IN POSITION ',X
 STOP
```

This is the program output:

```
ALPHA CHARACTER IN POSITION     1
 ALPHA CHARACTER IN POSITION    2
 ALPHA CHARACTER IN POSITION    3
 ALPHA CHARACTER IN POSITION    4
 NUMERIC CHARACTER IN POSITION     5
```

GOTO statement

Use the GOTO statement to transfer program control to the statement specified by *statement.label*. A colon (:) is optional in GOTO statements.

If the statement referenced is an executable statement, that statement and those that follow are executed. If it is a nonexecutable statement, execution proceeds at the first executable statement encountered after the referenced statement.

Syntax

GO[TO] *statement.label* [:]

GO TO *statement.label* [:]

Example

```
X=80
 GOTO 10
 STOP
 *
 10*
 IF X>20 THEN GO 20 ELSE STOP
 *
 20*
 PRINT 'AT LABEL 20'
 GO TO CALCULATE:
 STOP
 *
 CALCULATE:
 PRINT 'AT LABEL CALCULATE'
```

This is the program output:

```
AT LABEL 20
 AT LABEL CALCULATE
```

GROUP function

Use the GROUP function to return one or more substrings located between specified delimiters in *string*.

Syntax

GROUP (*string, delimiter, occurrence* [,*num.substr*])

delimiter evaluates to any character, including field mark, value mark, and subvalue marks. It delimits the start and end of the substring. If *delimiter* evaluates to more than one character, only the first character is used. Delimiters are not returned with the substring.

occurrence specifies which occurrence of the delimiter is to be used as a terminator. If *occurrence* is less than 1, 1 is assumed.

num.substr specifies the number of delimited substrings to return. If the value of *num.substr* is an empty string or less than 1, 1 is assumed. When more than one substring is returned, delimiters are returned along with the successive substrings.

If either *delimiter* or *occurrence* is not in the string, an empty string is returned, unless *occurrence* specifies 1. If *occurrence* is 1 and *delimiter* is not found, the entire string is returned. If *delimiter* is an empty string, the entire string is returned.

If *string* evaluates to the null value, null is returned. If *string* contains CHAR(128) (that is, @NULL.STR), it is treated like any other character in a string. If *delimiter*, *occurrence*, or *num.substr* evaluates to the null value, the GROUP function fails and the program terminates with a run-time error message.

The GROUP function works identically to the FIELD function.

Examples

```
D=GROUP("###DHHH#KK","#",4)
 PRINT "D= ",D
```

The variable D is set to DHHH because the data between the third and fourth occurrence of the delimiter # is DHHH.

```
REC="ACADABA"
 E=GROUP(REC,"A",2)
 PRINT "E= ",E
```

The variable E is set to "C".

```
VAR="?"
 Z=GROUP("A.1234$$$$&&",VAR,3)
 PRINT "Z= ",Z
```

Z is set to an empty string since "?" does not appear in the string.

```
Q=GROUP("+1+2+3ABAC","+",2,2)
 PRINT "Q= ",Q
```

Q is set to "1+2" since two successive fields were specified to be returned after the second occurrence of "+".

This is the program output:

```
D=                      DHHH
```

```
E=                              C
Z=
Q=                              1+2
```

GROUPSTORE statement

Use the GROUPSTORE statement to modify character strings by inserting, deleting, or replacing fields separated by specified delimiters.

Syntax

GROUPSTORE *new.string* IN *string* USING *start, n* [,*delimiter*]

new.string is an expression that evaluates to the character string to be inserted in *string*.

string is an expression that evaluates to the character string to be modified.

delimiter evaluates to any single ASCII character, including field, value, and subvalue marks. If you do not specify *delimiter*, the field mark is used.

start evaluates to a number specifying the starting field position. Modification begins at the field specified by *start*. If *start* is greater than the number of fields in *string*, the required number of empty fields is generated before the GROUPSTORE statement is executed.

n specifies the number of fields of *new.string* to insert in *string*. *n* determines how the GROUPSTORE operation is executed. If *n* is positive, *n* fields in *string* are replaced with the first *n* fields of *new.string*. If *n* is negative, *n* fields in *string* are replaced with all the fields in *new.string*. If *n* is 0, all the fields in *new.string* are inserted in *string* before the field specified by *start*.

If *string* evaluates to the null value, null is returned. If *new.string*, *start*, *n*, or *delimiter* is null, the GROUPSTORE statement fails and the program terminates with a run-time error message.

Example

```
Q='1#2#3#4#5'
 GROUPSTORE "A#B" IN Q USING 2,2,"#"
 PRINT "TEST1= ",Q
 *
 Q='1#2#3#4#5'
 GROUPSTORE "A#B" IN Q USING 2,-2,"#"
 PRINT "TEST2= ",Q
 *
 Q='1#2#3#4#5'
 GROUPSTORE "A#B" IN Q USING 2,0,"#"
 PRINT "TEST3= ",Q
 *
Q='1#2#3#4#5'
 GROUPSTORE "A#B#C#D" IN Q USING 1,4,"#"
 PRINT "TEST4= ",Q
 *
 Q='1#2#3#4#5'
 GROUPSTORE "A#B#C#D" IN Q USING 7,3,"#"
 PRINT "TEST5= ",Q
```

This is the program output:

```
TEST1=      1#A#B#4#5
 TEST2=         1#A#B#4#5
 TEST3=         1#A#B#2#3#4#5
```

```
TEST4=          A#B#C#D#5
TEST5=          1#2#3#4#5##A#B#C
```

GTS function

Use the GTS function to test if elements of one dynamic array are greater than elements of another dynamic array.

Syntax

GTS (*array1, array2*)

CALL -**GTS** (*return.array, array1, array2*)

CALL !**GTS** (*return.array, array1, array2*)

Each element of *array1* is compared with the corresponding element of *array2*. If the element from *array1* is greater than the element from *array2*, a 1 is returned in the corresponding element of a new dynamic array. If the element from *array1* is less than or equal to the element from *array2*, a 0 is returned. If an element of one dynamic array has no corresponding element in the other dynamic array, the undefined element is evaluated as an empty string, and the comparison continues.

If either of a corresponding pair of elements is the null value, null is returned for that element.

If you use the subroutine syntax, the resulting dynamic array is returned as *return.array*.

HEADING statement

Use the HEADING statement to specify the text and format of the heading to print at the top of each page of output.

Syntax

HEADING [ON *print.channel*] *heading*

HEADINGE [ON *print.channel*] *heading*

HEADINGN [ON *print.channel*] *heading*

The ON clause specifies the logical print channel to use for output. *print.channel* is an expression that evaluates to a number from –1 through 255. If you do not use the ON clause, logical print channel 0 is used, which prints to the user's terminal if PRINTER OFF is set (see the PRINTER statement, on page 294). Logical print channel –1 prints the data on the screen, regardless of whether a PRINTER ON statement has been executed.

heading is an expression that evaluates to the heading text and the control characters that specify the heading's format. You can use the following format control characters, enclosed in single quotation marks, in the heading expression:

Control character	Description
C[*n*]	Prints heading line centered in a field of *n* blanks. If *n* is not specified, centers the line on the page.
D	Prints current date formatted as *dd mmm yyyy*.
T	Prints current time and date formatted as *dd mmm yyyy hh:mm:ss*. Time is in 12-hour format with "am" or "pm" appended.

Control character	Description
\	Prints current time and date formatted as *dd mmm yyyy hh:mm:ss*. Time is in 12-hour format with “am” or “pm” appended. Do not put the backslash inside single quotation marks.
G	Inserts gaps to format headings.
I	Resets page number, time, and date for PIOPEN flavor only.
Q	Allows the use of the] ^ and \ characters.
R[*n*]	Inserts the record ID left-justified in a field of *n* blanks.
L	Starts a new line.
]	Starts a new line. Do not put the right bracket inside single quotation marks.
N	Suppresses automatic paging.
P[*n*]	Prints current page number right-justified in a field of *n* blanks. The default value for *n* is 4.
S	Left-justified, inserted page number.
^	Prints current page number right-justified in a field of *n* blanks. The default value for *n* is 4. Do not put the caret inside single quotation marks.

Two single quotation marks (' ') print one single quotation mark in heading text.

When the program is executed, the format control characters produce the specified results. You can specify multiple options in a single set of quotation marks.

If either *print.channel* or *heading* evaluates to the null value, the HEADING statement fails and the program terminates with a run-time error message.

Pagination begins with page 1 and increments automatically on generation of each new page or upon encountering the $PAGE statement.

Output to a terminal or printer is paged automatically. Use the N option in either a HEADING statement or a FOOTING statement to turn off automatic paging.

HEADINGE and HEADINGN statements

The HEADINGE statement is the same as the HEADING statement with the $OPTIONS statement HEADER.EJECT selected. HEADINGE causes a page eject with the HEADING statement. Page eject is the default for INFORMATION flavor accounts.

The HEADINGN statement is the same as the HEADING statement with the $OPTIONS -HEADER.EJECT selected. HEADINGN suppresses a page eject with the HEADING statement. The page eject is suppressed in IDEAL, PICK, REALITY, and IN2 flavor accounts.

Using] ^ and \ in headings

The characters] ^ and \ are control characters in headings and footings. To use these characters as normal characters, you must use the Q option and enclose the control character in double or single quotation marks. You only need to specify Q once in any heading or footing, but it must appear before any occurrence of the characters] ^ and \.

Formatting the heading text

The control character G (for gap) can be used to add blanks to text in headings to bring the width of a line up to device width. If G is specified once in a line, blanks are added to that part of the line to bring the line up to the device width. If G is specified at more than one point in a line, the space characters are distributed as *evenly* as possible to those points. See the following examples, in which the vertical bars represent the left and right margins:

Specification	Result
"Hello there"	|Hello there |
"'G'Hello there"	| Hello there|
"'G'Hello there'G'"	||
"Hello'G'there"	|Hello there|
"'G'Hello'G'there'G'"	| Hello there |

The minimum gap size is 0 blanks. If a line is wider than the device width even when all the gaps are 0, the line wraps, and all gaps remain 0.

If NLS is enabled, HEADING calculates gaps using varying display positions rather than character lengths. For more information about display length, see the *UniVerse NLS Guide*.

Left-justified inserted page number

The control character S (for sequence number) is left-justified at the point where the S appears in the line. Only one character space is reserved for the number. If the number of digits exceeds 1, any text to the right is shifted right by the number of extra characters required. For example, the statement:

HEADING "This is page 'S' of 100000"

results in headings such as:

```
This is page 3 of 100000
 This is page 333 of 100000
 This is page 3333 of 100000
```

INFORMATION flavor

Page Number Field:

In an INFORMATION flavor account the default width of the page number field is the length of the page number. Use the *n* argument to P to set the field width of the page number. You can also include multiple P characters to specify the width of the page field, or you can include blanks in the text that immediately precedes a P option. For example, 'PPP' prints the page number right-justified in a field of three blanks.

Note: In all other flavors, 'PPP' prints three identical page numbers, each in the default field of four.

Date format:

In an INFORMATION flavor account the default date format is *mm-dd-yy*, and the default time format is 24-hour style.

In PICK, IN2, REALITY, and IDEAL flavor accounts, use the HEADER.DATE option of the $OPTIONS statement to cause the HEADING statement, FOOTING statement, and $PAGE statement to behave as they do in INFORMATION flavor accounts.

PIOPEN flavor

Right-justified overwriting page number:

The control character P (for page) is right-justified at the point at which the P appears in the line. Only one character space is reserved for the number. If the number of digits exceeds 1, literal characters to the left of the initial position are overwritten. Normally you must enter a number of blanks to the left of the P to allow for the maximum page number to appear without overwriting other literal characters. For example, the statement:

HEADING "This is page 'P' of 100000"

results in headings such as:

```
This is page 3 of 100000
 This is pag333 of 100000
 This is pa3333 of 100000
```

Resetting the page number and the date:

The control character I (for initialize) resets the page number to 1, and resets the date.

Example

```
HEADING "'C'    LIST PRINTED:   'D'"
 FOR N=1 TO 10
    PRINT "THIS IS ANOTHER LINE"
 NEXT
```

This is the program output:

```
            LIST PRINTED:   04 Jun 1994
 THIS IS ANOTHER LINE
 THIS IS ANOTHER LINE
 THIS IS ANOTHER LINE
 THIS IS ANOTHER LINE
 THIS IS ANOTHER LINE
 THIS IS ANOTHER LINE
 THIS IS ANOTHER LINE
 THIS IS ANOTHER LINE
 THIS IS ANOTHER LINE
 THIS IS ANOTHER LINE
```

HMAC function

HMAC (keyed-Hash Message Authentication Code) is a specific construction for calculating a message authentication code (MAC) involving a cryptographic hash function in combination with a secret key.

Note: The HMAC function is in full compliance with RFC 2104.

Syntax

hmac= **HMAC**(*hmacAlg*, *hmacKey*, *hmacData*, [*outFormat*])

hmacAlg, *hmacKey*, and *hmacData* are string values. They can be supplied as quoted strings or as string variables, or a mix of both.

Parameters

The following table describes each parameter of the syntax.

Variable	Description
hmacAlg	Any OpenSSL supported digest functions, such as MD5, SHA1, SHA256, SHA384, or SHA512. If FIPS mode is turned on, only FIPS-compliant digest algorithms are allowed (namely SHA1, SHA256, SHA384 and SHA512).
hmacKey	User must take responsibility to keep this key secure.

Variable	Description
hmacData	Data for which a keyed hash is to be generated by the HMAC function.
outformat	Optional. Determines the output format. Currently, the following values are supported: 0 - The output is binary format (bit-stream) 1 - Lower case hexadecimal format, for example f22a….0def, which doubles the size of binary format 2 - Uppercase hexadecimal format, for example F22A…0DEF 3 - Lowercase hexadecimal delimited by colons, for example f2:2a:…0d:ef 4 - Uppercase hexadecimal delimited by colons, for example F1:2A:…0D:EF 5 - Lowercase hexadecimal delimited by spaces, for example f2 2a … 0d ef 6 - Uppercase hexadecimal delimited by spaces, for example F2 2A … 0D EF

Return codes

The function returns *hmac* as the result, whose length is determined by the digest algorithm. For example, for MD5, the length is 16 bytes. For SHA1, it is 20 bytes. For SHA256, it is 32 bytes. All lengths are for binary format.

If an error occurs, the function will return -1. STATUS() can be called to determine the error details.

The following table describes the status of each return code.

Return code	Status
0	Success.
1	Unsupported digest algorithm.
2	Not applicable.
3	HMAC cannot be obtained.
4	Invalid parameters.

HUSH statement

Use the HUSH statement to suppress the display of all output normally sent to a terminal during processing. HUSH also suppresses output to a COMO file or TANDEM display.

Syntax

```
HUSH { ON | OFF | expression} [SETTING status ]
```

SETTING *status* sets the value of a variable to the value of the HUSH state before the HUSH statement was executed. It can be used instead of the STATUS function to save the state so that it can be restored later. STATUS has a value of 1 if the previous state was HUSH ON or a value of 0 if the previous state was HUSH OFF.

You might use this statement when you are transmitting information over phone lines or when you are sending data to a hard-copy terminal. Both these situations result in slower transmission speeds. The unnecessary data display makes the task even slower.

HUSH acts as a toggle. If it is used without a qualifier, it changes the current state.

Do not use this statement to shut off output display unless you are sure the display is unnecessary. When you use HUSH ON, all output is suppressed including error messages and requests for information.

Value returned by the STATUS function

The previous state is returned by the `STATUS` function. If terminal output was suppressed prior to execution of the HUSH statement, the `STATUS` function returns a 1. If terminal output was enabled before execution of the HUSH statement, the `STATUS` function returns a 0.

Example

In the following example, terminal output is disabled with the HUSH statement and the previous state was saved in the variable USER.HUSH.STATE.

After executing some other statements, the program returns the user's process to the same HUSH state as it was in previous to the execution of the first HUSH statement:

```
HUSH ON
 USER.HUSH.STATE = STATUS()
 ...
 HUSH USER.HUSH.STATE
```

The example could have been written as follows:

```
HUSH ON SETTING USER.HUSH.STATE
 .
 .
 .
 HUSH USER.HUSH.STATE
```

ICHECK function

Use the `ICHECK` function to check if data you intend to write to an SQL table violates any SQL integrity constraints. `ICHECK` verifies that specified data and primary keys satisfy the defined SQL integrity constraints for an SQL table.

Syntax

`ICHECK` (*`dynamic.array`* [, *`file.variable`*] , *`key`* [, *`column#`*])

dynamic.array is an expression that evaluates to the data you want to check against any integrity constraints.

file.variable specifies an open file. If *file.variable* is not specified, the default file variable is assumed (for more information on default files, see the OPEN statement, on page 276).

key is an expression that evaluates to the primary key you want to check against any integrity constraints.

column# is an expression that evaluates to the number of the column in the table whose data is to be checked. If you do not specify *column#*, all columns in the file are checked. Column 0 specifies the primary key (record ID).

If *dynamic.array*, *file.variable*, *key*, or *column#* evaluates to the null value, the `ICHECK` function fails and the program terminates with a run-time error message.

You might use the ICHECK function to limit the amount of integrity checking that is done and thus improve performance. If you do this, however, you are assuming responsibility for data integrity. For example, you might want to use ICHECK with a program that changes only a few columns in a file. To do this, turn off the OPENCHK configurable parameter, open the file with the OPEN statement rather than the OPENCHECK statement, and use the ICHECK function before you write the updated record to verify, for each column you are updating, that you are not violating the table's integrity checks.

If the ON UPDATE clause of a referential constraint specifies an action, ICHECK always validates data being written to the referenced table; it does not check the referencing table. Therefore, ICHECK can succeed, but when the actual write is done, it can have a constraint failure while attempting to update the referencing table. If the referential constraint does not have an ON UPDATE clause, or if these clauses specify NO ACTION, the referencing table is checked to ensure that no row in it contains the old value of the referenced column.

ICHECK does not check triggers when it checks other SQL integrity constraints. Therefore, a write that fires a trigger can fail even if the ICHECK succeeds.

ICHECK returns a dynamic array of three elements separated by field marks:

*error.code*F*column#*F*constraint*

Element	Code	Description
error.code		A code that indicates the type of failure. Error codes can be any of the following:
	0	No failure
	1	SINGLEVALUED failure
	2	NOT NULL failure
	3	NOT EMPTY failure
	4	ROWUNIQUE failure (including single-column association KEY)
	5	UNIQUE (column constraint) failure
	6	UNIQUE (table constraint) failure
	7	Association KEY ROWUNIQUE failure when association has multiple KEY fields.
	8	CHECK constraint failure
	9	Primary key has too many parts
	10	Referential constraint failure
	11	Referential constraint failure that occurs when a numeric column references a nonnumeric column in the referenced table.
column#	The number of the column where the failure occurred. If any part of a primary key fails, 0 is returned. If the violation involves more than one column, -1 is returned.	
constraint	This element is returned only when *error.code* is 7 or 8. For code 7, the association name is returned. For code 8, the name of the CHECK constraint is returned if it has a name; otherwise, the CHECK constraint itself is returned.	

If the record violates more than one integrity constraint, ICHECK returns a dynamic array only for the first constraint that causes a failure.

The ICHECK function is advisory only. That is, if two programs try to write the same data to the same column defined as UNIQUE (see error 5), an ICHECK in the first program may pass. If the second

program writes data to the file before the first program writes its ICHECKed data, the first program's write fails even though the ICHECK did not fail.

ICONV function

Use the ICONV function to convert *string* to a specified internal storage format. *string* is an expression that evaluates to the string to be converted.

Syntax

ICONV (*string, conversion*)

conversion is an expression that evaluates to one or more valid conversion codes, separated by value marks (ASCII 253).

string is converted to the internal format specified by conversion. If multiple codes are used, they are applied from left to right. The first conversion code converts the value of *string*. The second conversion code converts the output of the first conversion, and so on.

If *string* evaluates to the null value, null is returned. If *conversion* evaluates to the null value, the ICONV function fails and the program terminates with a run-time error message.

The STATUS function reflects the result of the conversion:

Result	Description
0	The conversion is successful.
1	*string* is invalid. An empty string is returned, unless *string* is the null value, in which case null is returned.
2	*conversion* is invalid.
3	Successful conversion of possibly invalid data.

For information about converting strings to an external format, see the OCONV function, on page 272.

Examples

The following are examples of date conversions:

Source line	Converted value
DATE=ICONV("02-23-85","D")	6264
DATE=ICONV("30/9/67","DE")	-92
DATE=ICONV("6-10-85","D")	6371
DATE=ICONV("19850625","D")	6386
DATE=ICONV("85161","D")	6371

The following is an example of a time conversion:

Source line	Converted value
TIME=ICONV("9AM","MT")	32400

The following are examples of hex, octal, and binary conversions:

Source line	Converted value
HEX=ICONV("566D61726B","MX0C")	Vmark
OCT=ICONV("3001","MO")	1537
BIN=ICONV(1111,"MB")	15

The following are examples of masked decimal conversions:

Source line	Converted value
X=4956.00 X=ICONV(X,"MD2")	495600
X=563.888 X=ICONV(X,"MD0")	-564
X=ICONV(1988.28,"MD24")	19882800

ICONVS function

Use the ICONVS function to convert each element of *dynamic.array* to a specified internal storage format.

Syntax

ICONVS (*dynamic.array, conversion*)

CALL -**ICONVS** (*return.array, dynamic.array, conversion*)

CALL !**ICONVS** (*return.array, dynamic.array, conversion*)

conversion is an expression that evaluates to one or more valid conversion codes, separated by value marks (ASCII 253).

Each element of *dynamic.array* is converted to the internal format specified by *conversion* and is returned in a dynamic array. If multiple codes are used, they are applied from left to right. The first conversion code converts the value of each element of *dynamic.array*. The second conversion code converts the value of each element of the output of the first conversion, and so on.

If *dynamic.array* evaluates to the null value, null is returned. If an element of *dynamic.array* is the null value, null is returned for that element. If *conversion* evaluates to the null value, the ICONV function fails and the program terminates with a run-time error message.

If you use the subroutine syntax, the resulting dynamic array is returned as *return.array*.

The STATUS function reflects the result of the conversion:

Value	Description
0	The conversion is successful.
1	An element of *dynamic.array* is invalid. An empty string is returned, unless *dynamic.array* is the null value, in which case null is returned.
2	*conversion* is invalid.
3	Successful conversion of possibly invalid data.

For information about converting elements in a dynamic array to an external format, see the OCONVS function, on page 274.

IF statement

Use the IF statement to determine program flow based on the evaluation of *expression*. If the value of *expression* is true, the THEN statements are executed. If the value of *expression* is false, the THEN statements are ignored and the ELSE statements are executed. If *expression* is the null value, *expression* evaluates to false. If no ELSE statements are present, program execution continues with the next executable statement.

Syntax

```
IF expression {THEN statements [ELSE statements] | ELSE statements}

 IF expression
 {THEN statements
 [ELSE statements] |
 ELSE statements}

 IF expression {THEN
   statements
 END [ELSE
  statements
  END] | ELSE
   statements
 END}

 IF expression
  {THEN
 statements
 END
  [ELSE
 statements
 END] |
 ELSE
  statements
  END }
```

The IF statement must contain either a THEN clause or an ELSE clause. It need not include both.

Use the ISNULL function with the IF statement when you want to test whether the value of a variable is the null value. This is the only way to test for the null value since null cannot be equal to any value, including itself. The syntax is:

IF ISNULL (*expression*) …

You can write IF…THEN statements on a single line or separated onto several lines. Separating statements onto several lines can improve readability. Either way, the statements are executed identically.

You can nest IF…THEN statements. If the THEN or ELSE statements are written on more than one line, you must use an END statement as the last statement of the THEN or ELSE statements.

Conditional compilation

You can specify the conditions under which all or part of a BASIC program is to be compiled, using a modified version of the IF statement. The syntax of the conditional compilation statement is the same

as that of the IF statement except for the test expression, which must be one of the following: $TRUE, $T, $FALSE, or $F.

Example

```
X=10
 IF X>5 THEN PRINT 'X IS GREATER THAN 5';Y=3
 *
 IF Y>5 THEN STOP ELSE Z=9;     PRINT 'Y IS LESS THAN 5'
 *
 IF Z=9 THEN PRINT 'Z EQUALS 9'
 ELSE PRINT 'Z DOES NOT EQUAL 9' ; STOP
 *
 IF Z=9 THEN
 GOTO 10
 END ELSE
 STOP
 END
 *
 10*
 IF Y>4
 THEN
 PRINT 'Y GREATER THAN 4'
 END
 ELSE
 PRINT 'Y IS LESS THAN 4'
 END
```

This is the program output:

```
X IS GREATER THAN 5
 Y IS LESS THAN 5
 Z EQUALS 9
 Y IS LESS THAN 4
```

IFS function

Use the `IFS` function to return a dynamic array whose elements are chosen individually from one of two dynamic arrays based on the contents of a third dynamic array.

Syntax

IFS (*dynamic.array, true.array, false.array*)

CALL **-IFS** (*return.array, dynamic.array, true.array, false.array*)

CALL **!IFS** (*return.array, dynamic.array, true.array, false.array*)

Each element of *dynamic.array* is evaluated. If the element evaluates to true, the corresponding element from *true.array* is returned to the same element of a new dynamic array. If the element evaluates to false, the corresponding element from *false.array* is returned. If there is no corresponding element in the correct response array, an empty string is returned for that element. If an element is the null value, that element evaluates to false.

If you use the subroutine syntax, the resulting dynamic array is returned as *return.array*.

ILPROMPT function

Use the `ILPROMPT` function to evaluate a string containing UniVerse in-line prompts.

Syntax

ILPROMPT (*in.line.prompt*)

in.line.prompt is an expression that evaluates to a string containing in-line prompts. In-line prompts have the following syntax:

<< [*control***,**] ... *text* [**,** *option*] >>

control is an option that specifies the characteristics of the prompt. Separate multiple control options with commas. Possible control options are:

Option	Description
A	Always prompts when the sentence containing the control option is executed. If this option is not specified, the input value from a previous execution of this prompt is used.
C*n*	Uses the word in the *n*th position in the command line as the input value. (The verb is in position 1.)
F(*filename*) *record.ID* [,*fm* [,*vm* [,*sm*]]])	Finds input value in *record.ID* in *filename*. Optionally, extract a value (*vm*) or subvalue (*sm*) from the field (*fm*).
I*n*	Uses the word in the *n*th position in the command line as the input value, but prompts if word *n* was not entered.
P	Saves the input from an in-line prompt. BASIC uses the input for all in-line prompts with the same prompt text until the saved input is overwritten by a prompt with the same prompt text and with a control option of A, C, I, or S, or until control returns to the UniVerse prompt. The P option saves the input from an in-line prompt in the current paragraph, or in other paragraphs.
R	Repeats the prompt until Return is pressed.
R(*string*)	Repeats the prompt until Return is pressed, and inserts *string* between each entry.
S*n*	Takes the *n*th word from the command but uses the most recent command entered at the UniVerse level to execute the paragraph, rather than an argument in the paragraph. Use this option in nested paragraphs.
@(CLR)	Clears the screen.
@(BELL)	Rings the terminal bell.
@(TOF)	Positions the prompt at the top left of the screen.
@(*col*, *row*)	Prompts at this column and row number on the terminal.

text is the prompt text to display. If you want to include quotation marks (single or double) or backslashes as delimiters within the prompt text, you must enclose the entire text string in a set of delimiters different from the delimiters you are using within the text string. For example, to print the following prompt text:

```
'P'RINTER OR 'T'ERMINAL
```

you must specify the prompt text as

```
\'P'RINTER OR 'T'ERMINAL\
```

or

```
"'P'RINTER OR 'T'ERMINAL"
```

option can be any valid ICONV function conversion or matching pattern (see the MATCH operator, on page 253). A conversion must be in parentheses.

If *in.line.prompt* evaluates to the null value, the `ILPROMPT` function fails and the program terminates with a run-time error.

If the in-line prompt has a value, that value is substituted for the prompt. If the in-line prompt does not have a value, the prompt is displayed to request an input value when the sentence is executed. The value entered at the prompt is then substituted for the in-line prompt.

Once a value has been entered for a particular prompt, the prompt will continue to have that value until a CLEARPROMPTS statement is executed, unless the control option A is specified. CLEARPROMPTS clears all values entered for in-line prompts.

You can enclose prompts within prompts.

Example

```
A="This is your number. - <<number>>"
 PRINT ILPROMPT(A)
 PRINT ILPROMPT("Your number is <<number>>, and your letter is
    <<letter>>.")
```

This is the program output:

```
number=5
This is your number. - 5
 letter=K
Your number is 5, and your letter is K.
```

INCLUDE statement

Use the INCLUDE statement to direct the compiler to insert the source code in the record program and compile it along with the main program. The INCLUDE statement differs from the $CHAIN statement in that the compiler returns to the main program and continues compiling with the statement following the INCLUDE statement.

Syntax

INCLUDE [*filename*] *program*

INCLUDE *program* FROM *filename*

When *program* is specified without *filename*, *program* must be a record in the same file as the program currently containing the INCLUDE statement.

If *program* is a record in a different file, the name of the file in which it is located must be specified in the INCLUDE statement, followed by the name of the program. The file name must specify a type 1 or type 19 file defined in the VOC file.

You can nest INCLUDE statements.

The INCLUDE statement is a synonym for the $INCLUDE and #INCLUDE statements.

Example

```
PRINT "START"
INCLUDE END
PRINT "FINISH"
```

When this program is compiled, the INCLUDE statement inserts code from the program END (see the example on the END statement, on page 144). This is the program output:

```
START
THESE TWO LINES WILL PRINT ONLY
WHEN THE VALUE OF 'A' IS 'YES'.

THIS IS THE END OF THE PROGRAM
FINISH
```

INDEX function

Use the `INDEX` function to return the starting character position for the specified occurrence of *substring* in *string*.

Syntax

`INDEX` (*`string, substring, occurrence`*)

string is an expression that evaluates to any valid string. *string* is examined for the substring expression.

occurrence specifies which occurrence of *substring* is to be located.

When *substring* is found and if it meets the occurrence criterion, the starting character position of the substring is returned. If *substring* is an empty string, 1 is returned. If the specified occurrence of the substring is not found, or if *string* or *substring* evaluate to the null value, 0 is returned.

If *occurrence* evaluates to the null value, the `INDEX` function fails and the program terminates with a run-time error message.

PICK, IN2, and REALITY flavors

In PICK, IN2, and REALITY flavor accounts, the search continues with the next character regardless of whether it is part of the matched substring. Use the COUNT.OVLP option of the $OPTIONS statement to get this behavior in IDEAL and INFORMATION flavor accounts.

Example

```
Q='AAA11122ABB1619MM'
 P=INDEX(Q,1,4)
 PRINT "P= ",P
 *
 X='XX'
 Y=2
 Q='P1234XXOO1299XX00P'
 TEST=INDEX(Q,X,Y)
 PRINT "TEST= ",TEST
 *
 Q=INDEX("1234",'A',1)
 PRINT "Q= ",Q
```

```
 * The substring cannot be found.
 *
 POS=INDEX('222','2',4)
PRINT "POS= ",POS
 * The occurrence (4) of the substring does not exist.
```

This is the program output:

```
P=                      12
 TEST=          14
 Q=                          0
 POS=               0
```

INDEXS function

Use the INDEXS function to return a dynamic array of the starting column positions for a specified occurrence of a substring in each element of *dynamic.array*.

Syntax

INDEXS (*dynamic.array, substring, occurrence*)

CALL **-INDEXS** (*return.array, dynamic.array, substring, occurrence*)

CALL **!INDEXS** (*return.array, dynamic.array, substring, occurrence*)

Each element is examined for *substring*.

occurrence specifies which occurrence of *substring* is to be located.

When *substring* is found, and if it meets the occurrence criterion, the starting column position of the substring is returned. If *substring* is an empty string, 1 is returned. If the specified occurrence of *substring* cannot be found, 0 is returned.

If *dynamic.array* evaluates to the null value, 0 is returned. If any element of *dynamic.array* is null, 0 is returned for that element. If *occurrence* is the null value, the INDEXS function fails and the program terminates with a run-time error message.

If you use the subroutine syntax, the resulting dynamic array is returned as *return.array*.

INDICES function

Use the INDICES function to return information about the secondary key indexes in a file.

Syntax

INDICES (*file.variable* [,*indexname*])

file.variable specifies an open file.

indexname is the name of a secondary index in the specified file.

If only *file.variable* is specified, a dynamic array is returned that contains the index names of all secondary indexes for the file. The index names are separated by field marks. If *file.variable* has no indexes, an empty string is returned.

If *indexname* is specified, information is returned in a dynamic array for *indexname*. Field 1 of the dynamic array contains the following information:

Value	Value can be...	Description
Value 1	D I A S C SQL	Data descriptor index. I-descriptor index. A-descriptor index. S-descriptor index. A- or S-descriptor index with correlative in field 8. SQL index.
Value 2	1 3 empty	Index needs rebuilding. Index is currently being built concurrently. Index does not need rebuilding.
Value 3	1 empty	Empty strings are not indexed. Empty strings are indexed.
Value 4	1 empty	Automatic updating enabled. Automatic updating disabled.
Value 5	pathname empty	Full path name of the index file. File is a distributed file.
Value 6	1 empty	Updates are pending. No updates pending.
Value 7	L R	Left-justified. Right-justified.
Value 8	N U	Nonunique. Unique.
Value 9	part numbers	Subvalued list of distributed file part numbers.
Value 10	1 Index needs building empty No build needed	Subvalued list corresponding to subvalues in Value 9.
Value 11	1 Empty strings not indexed empty Empty strings indexed	Subvalued list corresponding to subvalues in Value 9.
Value 12	1 Updating enabled empty Updating disabled	Subvalued list corresponding to subvalues in Value 9.
Value 13	index pathnames	Subvalued list of path names for indexes on distributed file part files, corresponding to subvalues in Value 9.
Value 14	1 Updates pending empty No updates pending	Subvalued list corresponding to subvalues in Value 9.
Value 15	L Left-justified R Right-justified	Subvalued list corresponding to subvalues in Value 9.
Value 16	N Nonunique U Unique	Subvalued list corresponding to subvalues in Value 9.

Value	Value can be...	Description
Value 17	collate name	Name of the Collate convention of the index.

If Value 1 of Field 1 is D, A, or S, Field 2 contains the field location (that is, the field number), and Field 6 contains either S (single-valued field) or M (multivalued field).

If Value 1 of Field 1 is I or SQL, the other fields of the dynamic array contain the following information, derived from the I-descriptor in the file dictionary:

Field	Value can be...
Field 2	I-type expression
Field 3	Output conversion code
Field 4	Column heading
Field 5	Width, justification
Field 6	S – single-valued field M – multivalued field
Field 7	Association name
Fields 8-15	Empty
Fields 16-*19*	Compiled I-descriptor data
Field 20	Compiled I-descriptor code

If Value 1 of Field 1 is C, the other fields of the dynamic array contain the following information, derived from the A- or S-descriptor in the file dictionary:

Field	Value can be...
Field 2	Field number (location of field)
Field 3	Column heading
Field 4	Association code
Fields 5-6	Empty
Field 7	Output conversion code
Field 8	Correlative code
Field 9	L or R (justification)
Field 10	Width of display column

If either *file.variable* or *indexname* is the null value, the `INDICES` function fails and the program terminates with a run-time error message.

Any file updates executed in a transaction (that is, between a BEGIN TRANSACTION statement and a COMMIT statement) are not accessible to the `INDICES` function until after the COMMIT statement has been executed.

If NLS is enabled, the INDICES function reports the name of the current Collate convention (as specified in the NLS.LC.COLLATE file) in force when the index was created. See Value 17 in Field 1 for the name of the Collate convention of the index. For more information about the Collate convention, see the *UniVerse NLS Guide*.

initSecureServerSocket function

Use the `initSecureServerSocket`() function to create a secured connection-oriented stream server socket. It does exactly the same as the `initServerSocket`() function except that the connection will be secure.

Once the server socket is opened, any change in the associated security context will not affect the opened socket.

Syntax

`initSecureServerSocket(`*`name_or_IP, port, backlog, svr_socket, context`*`)`

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
name_or_IP	DNS name (x.com) or host interface address. Special addresses include: ▪ 127.0.0.1 (INADDR_LOOPBACK) ▪ 0.0.0.0 (INADDR_ANY) ▪ 255.255.255.255 (INADDR_BROADCAST) Generally, this parameter should be set to 0.0.0.0.
port	Port number. If the port number is specified as a value <= 0, CallHTTP defaults to a port number of 40001.
backlog	The maximum length of the queue of pending connections (for example, concurrent client-side connections).
svr_socket	The handle to the server side socket.
context	The handle to the security context.

Return codes

The following table describes the status of each return code.

Return code	Description
0	Success.
1-41	See Socket function error return codes, on page 584.
99	UniVerse failed to obtain a license for an interactive PHANTOM process.
101	Invalid security context handle.

initServerSocket function

Use the `initServerSocket`() function to create a connection-oriented (stream) socket. Associate this socket with an address (*name_or_IP*) and port number (*port*), and specify the maximum length the queue of pending connections may grow to.

Syntax

`initServerSocket``(`*`name_or_IP, port, backlog, svr_socket`*`)`

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
name_or_IP	DNS name (x.com) or host interface address. Special addresses include: ▪ 127.0.0.1 (INADDR_LOOPBACK) ▪ 0.0.0.0 (INADDR_ANY) ▪ 255.255.255.255 (INADDR_BROADCAST) Generally, this parameter should be set to 0.0.0.0.
port	Port number. If the port number is specified as a value <= 0, CallHTTP defaults to a port number of 40001.
backlog	The maximum length of the queue of pending connections (for example, concurrent client-side connections).
svr_socket	The handle to the server side socket.

Return codes

The following table describes the status of each return code.

Return code	Description
0	Success.
Non-zero	See Socket function error return codes, on page 584.

INMAT function

Use the `INMAT` function to return the number of array elements that have been loaded after the execution of MATREAD statements, MATREADL statement, MATREADU statement, or MATPARSE statement, or to return the modulo of a file after the execution of an OPEN statement. You can also use the `INMAT` function after a DIMENSION statement to determine whether the DIM statement failed due to lack of available memory. If a preceding DIM statement fails, `INMAT` returns a value of 1.

Syntax

INMAT ([*array*])

If the matrix assignment exceeds the number of elements specified in its dimensioning statement, the zero element is loaded by the MATREAD, MATREADL, MATREADU, or MATPARSE statement. If the array dimensioning is too small and the zero element has been loaded, the `INMAT` function returns a value of 0.

If *array* is specified, the `INMAT` function returns the current dimensions of the array. If *array* is the null value, the `INMAT` function fails and the program terminates with a run-time error message.

Example

```
DIM X(6)
 D='123456'
 MATPARSE X FROM D,''
 Y=INMAT()
 PRINT 'Y= ':Y
```

```
*
DIM X(5)
A='CBDGFH'
MATPARSE X FROM A,''
C=INMAT()
PRINT 'C= ':C
*
OPEN '','VOC' TO FILE ELSE STOP
T=INMAT()
PRINT 'T= ':T
```

This is the program output:

```
Y= 6
 C= 0
 T= 23
```

INPUT statement

Use the INPUT statement to halt program execution and prompt the user to enter a response. Data entered at the terminal or supplied by a DATA statement in response to an INPUT statement is assigned to *variable*. Input supplied by a DATA statement is echoed to the terminal. If the response is a RETURN with no preceding data, an empty string is assigned to *variable*.

Syntax

```
INPUT variable [,length] [:] [_]
INPUT @ (col, row) [, | :] variable [,length] [:] [format] [_]

 INPUTIF @ (col, row) [, | :] variable [,length] [:] [format] [_]
[THEN statements] [ELSE statements]
```

The INPUT statement has two syntaxes. The first syntax displays a prompt and assigns the input to *variable*. The second syntax specifies the location of the input field on the screen and lets you display the current value of *variable*. Both the current value and the displayed input can be formatted.

Use the INPUTIF statement to assign the contents of the type-ahead buffer to a variable. If the type-ahead buffer is empty, the ELSE statements are executed, otherwise any THEN statements are executed.

Use the @ expression to specify the position of the input field. The prompt is displayed one character to the left of the beginning of the field, and the current value of *variable* is displayed as the value in the input field. The user can edit the displayed value or enter a new value. If the first character typed in response to the prompt is an editing key, the user can edit the contents of the field. If the first character typed is anything else, the field's contents are deleted and the user can enter a new value. Editing keys are defined in the *terminfo* files; they can also be defined by the KEYEDIT statement. Calculations are based on display length rather than character length.

col and *row* are expressions that specify the column and row positions of the input prompt. The prompt is positioned one character to the left of the input field. Because the prompt character is positioned to the left of the col position, you must set the prompt to the empty string if you want to use column 0. Otherwise, the screen is erased before the prompt appears.

length specifies the maximum number of characters allowed as input. When the maximum number of characters is entered, input is terminated. If the @ expression is used, the newline is suppressed.

If *length* evaluates to less than 0 (for example, -1), the input buffer is tested for the presence of characters. If characters are present, *variable* is set to 1, otherwise it is set to 0. No input is performed.

If you use the underscore (_) with the length expression, the user must enter the RETURN manually at the terminal when input is complete. Only the specified number of characters is accepted.

Use a format expression to validate input against a format mask and to format the displayed input field. The syntax of the format expression is the same as that for the FMT function. If you specify a length expression together with a format expression, length checking is performed. If input does not conform to the format mask, an error message appears at the bottom of the screen, prompting the user for the correct input.

The colon (:) suppresses the newline after input is terminated. This allows multiple input prompts on a single line.

The default prompt character is a question mark. Use the PROMPT statement to reassign the prompt character.

The INPUT statement prints only the prompt character on the screen. To print a variable name or prompt text along with the prompt, precede the INPUT statement with a PRINT statement.

The INPUT statement lets the user type ahead when entering a response. Users familiar with a sequence of prompts can save time by entering data at their own speed, not waiting for all prompts to be displayed. Responses to a sequence of INPUT prompts are accepted in the order in which they are entered.

If *col*, *row*, *length*, or *format* evaluate to the null value, the INPUT statement fails and the program terminates with a run-time error message. If *variable* is the null value and the user types the TRAP key, null is retained as the value of *variable*.

If NLS is enabled, INPUT @ displays the initial value of an external multibyte character set through the mask as best as possible. If the user enters a new value, *mask* disappears, and an input field of the approximate length (not including any inserted characters) is entered. For details about *format* and *mask*, see the FMTDP function.

Only backspace and kill are supported for editing functions when using a format mask with input. When the user finishes the input, the new value is redisplayed through the mask in the same way as the original value. For more information about NLS in BASIC programs, see the *UniVerse NLS Guide*.

PICK flavor

In a PICK flavor account, the syntax of the INPUT and INPUT @ statements includes THEN and ELSE clauses:

INPUT *variable* [,*length*] [:] [_] [THEN *statements*] [ELSE *statements*]

INPUT @ (*col*, *row*) [, | :] *variable* [,*length*] [:] [*format*] [_] [THEN *statements*] [ELSE *statements*]

To use THEN and ELSE clauses with INPUT statements in other flavors, use the INPUT.ELSE option of the $OPTIONS statement, on page 23.

PICK, IN2, and REALITY flavors

In PICK, IN2, and REALITY flavors, values supplied by a DATA statement are not echoed. To suppress echoing input from DATA statements in IDEAL and INFORMATION flavors, use the SUPP.DATA.ECHO option of the $OPTIONS statement.

Examples

In the following examples of program output, bold type indicates words the user types. In the first example the value entered is assigned to the variable NAME:

Source lines	Program output
INPUT NAME PRINT NAME	? **Dave** Dave

In the next example the value entered is assigned to the variable CODE. Only the first seven characters are recognized. A RETURN and a LINEFEED automatically occur.

Source lines	Program output
INPUT CODE, 7 PRINT CODE	? **1234567** 1234567

In the next example the user can enter more than two characters. The program waits for a RETURN to end input, but only the first two characters are assigned to the variable YES.

Source Lines	Program Output
INPUT YES, 2_ PRINT YES	? **1234** 12

In the next example the colon inhibits the automatic LINEFEED after the RETURN:

Source lines	Program output
INPUT YES, 2_: PRINT "=",YES	? **HI THERE** =HI

In the next example the input buffer is tested for the presence of characters. If characters are present, VAR is set to 1, otherwise it is set to 0. No input is actually done.

Source lines	Program output
INPUT VAR, -1 PRINT VAR	0

In the next example the PRINT statement puts INPUT NAME before the input prompt:

Source lines	Program output
PRINT "INPUT NAME": INPUT NAME PRINT NAME	INPUT NAME? **Dave** Dave

In the next example the contents of X are displayed at column 5, row 5 in a field of 10 characters. The user edits the field, replacing its original contents (CURRENT) with new contents (NEW). The new input is displayed. If the PRINT statement after the INPUT statement were not used, X would be printed immediately following the input field on the same line, since INPUT with the @ expression does not execute a LINEFEED after a RETURN.

Source lines	Program output
PRINT @(-1) X = "CURRENT" INPUT @(5,5) X,10 PRINT PRINT X	?**NEW**_____ NEW

INPUTCLEAR statement

Use the INPUTCLEAR statement to clear the type-ahead buffer. You can use this statement before input prompts so input is not affected by unwanted characters.

Syntax

```
INPUTCLEAR
```

Example

```
PRINT "DO YOU WANT TO CONTINUE (Y/N)?"
 INPUTCLEAR
 INPUT ANSWER, 1
```

INPUTDISP statement

Use the INPUTDISP statement with an @ expression to position the cursor at a specified location and define a format for the variable to print. The current contents of *variable* are displayed as the value in the defined field. Calculations are based on display length rather than character length.

Syntax

INPUTDISP [@(*col, row*) [, | :]] *variable* [*format*]

col specifies the column position, and *row* specifies the row position.

format is an expression that defines how the variable is to be displayed in the output field. The syntax of the format expression is the same as that for the FMT function.

Example

```
PRINT @(-1)
 X = "CURRENT LINE"
 INPUTDISP @(5,5),X"10T"
```

The program output on a cleared screen is:

```
                    CURRENT
               LINE
```

INPUTDP statement

In NLS mode, use the INPUTDP statement to let the user enter data. The INPUTDP statement is similar to the INPUT statement, INPUTIF statement, and INPUTDISP statement, but it calculates display positions rather than character lengths.

Syntax

```
INPUTDP variable [, length] [:] [_] [THEN statements] [ELSE statements]
```

variable contains the input from a user prompt.

length specifies the maximum number of characters in display length allowed as input. INPUTDP calculates the display length of the input field based on the current terminal map. When the specified number of characters is entered, an automatic newline is executed.

The colon (:) executes the RETURN, suppressing the newline. This allows multiple input prompts on a single line.

If you use the underscore (_), the user must enter the RETURN manually when input is complete, and the newline is not executed.

For more information about display length, see the *UniVerse NLS Guide*.

INPUTERR statement

Use the INPUTERR statement to print a formatted error message on the bottom line of the terminal. *error.message* is an expression that evaluates to the error message text. The message is cleared by the next INPUT statement or is overwritten by the next INPUTERR statement or PRINTERR statement. INPUTERR clears the type-ahead buffer.

Syntax

```
INPUTERR [error.message]
```

error.message can be any BASIC expression. The elements of the expression can be numeric or character strings, variables, constants, or literal strings. The null value cannot be output. The expression can be a single expression or a series of expressions separated by commas (,) or colons (:) for output formatting. If no error message is designated, a blank line is printed. If *error.message* evaluates to the null value, the default error message is printed:

```
Message ID is NULL: undefined error
```

Expressions separated by commas are printed at preset tab positions. The default tabstop setting is 10 characters. For information about changing the default setting, see the TABSTOP statement, on page 399. Multiple commas can be used together to cause multiple tabulations between expressions.

Expressions separated by colons are concatenated: that is, the expression following the colon is printed immediately after the expression preceding the colon.

INPUTIF statement

Use the INPUTIF statement to assign the contents of the type-ahead buffer to a variable.

For details, see the INPUTIF statement, on page 221.

INPUTNULL statement

Use the INPUTNULL statement to define a character to be recognized as an empty string when it is input in response to an INPUT statement. If the only input to the INPUT statement is *character*, that character is recognized as an empty string. *character* replaces the default value of the INPUT variable with an empty string. If *character* evaluates to the null value, the INPUTNULL statement fails and the program terminates with a run-time error message.

Syntax

```
INPUTNULL character
```

You can also assign an empty string to the variable used in the INPUT @ statement before executing the INPUT @. In this case entering a RETURN leaves the variable set to the empty string.

Note: Although the name of this statement is INPUTNULL, it does not define *character* to be recognized as the null value. It defines it to be recognized as an empty string.

INPUTTRAP statement

Use the INPUTTRAP statement to branch to a program label or subroutine when a trap character is input. Execution is passed to the statement label which corresponds to the trap number of the trap character. If the trap number is larger than the number of labels, execution is passed to the statement specified by the last label in the list.

Syntax

```
INPUTTRAP [trap.chars] {GOTO | GOSUB} label [ ,label …]
```

trap.chars is an expression that evaluates to a string of characters, each of which defines a trap character. The first character in the string is defined as trap one. Additional characters are assigned consecutive trap numbers. Each trap character corresponds to one of the labels in the label list. If *trap.chars* evaluates to the null value, the INPUTTRAP statement fails and the program terminates with a run-time error message.

Using GOTO causes execution to be passed to the specified statement label. Control is not returned to the INPUTTRAP statement except by the use of another trap. Using GOSUB causes execution to be passed to the specified subroutine, but control can be returned to the INPUTTRAP statement by a RETURN statement. Control is returned to the statement following the INPUTTRAP statement, not the INPUT @ statement that received the trap.

INS statement

Use the INS statement to insert a new field, value, or subvalue into the specified *dynamic.array*.

Syntax

```
INS expression BEFORE dynamic.array < field# [,value# [ ,subvalue#]] >
```

expression specifies the value of the new element to be inserted.

dynamic.array is an expression that evaluates to the dynamic array to be modified.

field#, *value#*, and *subvalue#* specify the type and position of the new element to be inserted and are called delimiter expressions.

There are three possible outcomes of the INS statement, depending on the delimiter expressions specified.

Case	Result
Case 1:	If both *value#* and *subvalue#* are omitted or are 0, INS inserts a new field with the value of *expression* into the dynamic array. If *field#* is positive and less than or equal to the number of fields in *dynamic.array*, the value of *expression* followed by a field mark is inserted before the field specified by *field#*. If *field#* is -1, a field mark followed by the value of *expression* is appended to the last field in *dynamic.array*. If *field#* is positive and greater than the number of fields in *dynamic.array*, the proper number of field marks followed by the value of *expression* are appended so that the value of *field#* is the number of the new field.
Case 2:	If *value#* is nonzero and *subvalue#* is omitted or is 0, INS inserts a new value with the value of *expression* into the dynamic array. If *value#* is positive and less than or equal to the number of values in the field, the value of *expression* followed by a value mark is inserted before the value specified by *value#*. If *value#* is -1, a value mark followed by the value of *expression* is appended to the last value in the field. If *value#* is positive and greater than the number of values in the field, the proper number of value marks followed by the value of *expression* are appended to the last value in the specified field so that the number of the new value in the field is *value#*.
Case 3:	If *field#*, *value#*, and *subvalue#* are all specified, INS inserts a new subvalue with the value of *expression* into the dynamic array. If *subvalue#* is positive and less than or equal to the number of subvalues in the value, the value of *expression* following by a subvalue mark is inserted before the subvalue specified by *subvalue#*. If *subvalue#* is -1, a subvalue mark followed by *expression* is appended to the last subvalue in the value. If *subvalue#* is positive and greater than the number of subvalues in the value, the proper number of subvalue marks followed by the value of *expression* are appended to the last subvalue in the specified value so that the number of the new subvalue in the value is *subvalue#*.

If all delimiter expressions are 0, the original string is returned.

In IDEAL, PICK, PIOPEN, and REALITY flavor accounts, if *expression* is an empty string and the new element is appended to the end of the dynamic array, the end of a field, or the end of a value, the dynamic array, field, or value is left unchanged. Additional delimiters are not appended. Use the EXTRA.DELIM option of the $OPTIONS statement to make the INS statement append a delimiter to the dynamic array, field, or value.

If *expression* evaluates to the null value, null is inserted into *dynamic.array*. If *dynamic.array* evaluates to the null value, it remains unchanged by the insertion. If the INS statement references a subelement of an element whose value is the null value, the dynamic array is unchanged.

If any delimiter expression is the null value, the INS statement fails and the program terminates with a run-time error message.

INFORMATION and IN2 flavors

In INFORMATION and IN2 flavor accounts, if *expression* is an empty string and the new element is appended to the end of the dynamic array, the end of a field, or the end of a value, a delimiter is appended to the dynamic array, field, or value. Use the -EXTRA.DELIM option of the $OPTIONS statement to make the INS statement work as it does in IDEAL, PICK, and REALITY flavor accounts.

Examples

In the following examples a field mark is shown by F, a value mark is shown by V, and a subvalue mark is shown by S.

The first example inserts the character # before the first field and sets Q to #FFF1V2V3S6F9F5F7V3:

```
R=@FM:@FM:1:@VM:2:@VM:3:@SM:6:@FM:9:@FM:5:@FM:7:@VM:3
Q=R
INS "#" BEFORE Q<1,0,0>
```

The next example inserts a # before the third value of field 3 and sets the value of Q to FF1V2V#V3S6F9F5F7V3:

```
Q=R
 INS "#" BEFORE Q<3,3,0>
```

The next example inserts a value mark followed by a # after the last value in the field and sets Q to FF1V2V3S6F9V#F5F7V3:

```
Q=R
 INS "#" BEFORE Q<4,-1,0>
```

The next example inserts a # before the second subvalue of the second value of field 3 and sets Q to FF1V2S#V3S6F9F5F7V3:

```
Q=R
 INS "#" BEFORE Q<3,2,2>
```

INSERT function

Use the `INSERT` function to return a dynamic array that has a new field, value, or subvalue inserted into the specified dynamic array.

Syntax

INSERT (*dynamic.array, field#, value#, subvalue#, expression*)

INSERT (*dynamic.array, field# [,value# [,subvalue#]] ; expression*)

dynamic.array is an expression that evaluates to a dynamic array.

field#, *value#*, and *subvalue#* specify the type and position of the new element to be inserted and are called delimiter expressions. *value#* and *subvalue#* are optional, but if either is omitted, a semicolon (;) must precede *expression*, as shown in the second syntax line.

expression specifies the value of the new element to be inserted.

There are three possible outcomes of the INSERT function, depending on the delimiter expressions specified.

Case	Result
Case 1:	If both *value#* and *subvalue#* are omitted or are 0, INSERT inserts a new field with the value of *expression* into the dynamic array. If *field#* is positive and less than or equal to the number of fields in *dynamic.array*, the value of *expression* followed by a field mark is inserted before the field specified by *field#*. If *field#* is -1, a field mark followed by the value of *expression* is appended to the last field in *dynamic.array*. If *field#* is positive and greater than the number of fields in *dynamic.array*, the proper number of field marks followed by the value of *expression* are appended so that the value of *field#* is the number of the new field.
Case 2:	If *value#* is nonzero and *subvalue#* is omitted or is 0, INSERT inserts a new value with the value of *expression* into the dynamic array. If *value#* is positive and less than or equal to the number of values in the field, the value of *expression* followed by a value mark is inserted before the value specified by *value#*. If *value#* is -1, a value mark followed by the value of *expression* is appended to the last value in the field. If *value#* is positive and greater than the number of values in the field, the proper number of value marks followed by the value of *expression* are appended to the last value in the specified field so that the number of the new value in the field is *value#*.
Case 3:	If *field#*, *value#*, and *subvalue#* are all specified, INSERT inserts a new subvalue with the value of *expression* into the dynamic array. If *subvalue#* is positive and less than or equal to the number of subvalues in the value, the value of *expression* following by a subvalue mark is inserted before the subvalue specified by *subvalue#*. If *subvalue#* is -1, a subvalue mark followed by *expression* is appended to the last subvalue in the value. If *subvalue#* is positive and greater than the number of subvalues in the value, the proper number of subvalue marks followed by the value of *expression* are appended to the last subvalue in the specified value so that the number of the new subvalue in the value is *subvalue#*.

In IDEAL, PICK, PIOPEN, and REALITY accounts, if *expression* is an empty string and the new element is appended to the end of the dynamic array, the end of a field, or the end of a value, the dynamic array, field, or value is left unchanged. Additional delimiters are not appended. Use the EXTRA.DELIM option of the $OPTIONS statement to make the INSERT function append a delimiter to the dynamic array, field, or value.

If *expression* evaluates to the null value, null is inserted into *dynamic.array*. If *dynamic.array* evaluates to the null value, it remains unchanged by the insertion. If any delimiter expression is the null value, the INSERT function fails and the program terminates with a run-time error message.

INFORMATION and IN2 flavors

In INFORMATION and IN2 flavor accounts, if *expression* is an empty string and the new element is appended to the end of the dynamic array, the end of a field, or the end of a value, a delimiter

is appended to the dynamic array, field, or value. Use the -EXTRA.DELIM option of the $OPTIONS statement to make the INSERT function work as it does in IDEAL, PICK, and REALITY flavor accounts.

Examples

In the following examples a field mark is shown by F, a value mark is shown by V, and a subvalue mark is shown by S.

The first example inserts the character # before the first field and sets Q to #FFF1V2V3S6F9F5F7V:

```
R=@FM:@FM:1:@VM:2:@VM:3:@SM:6:@FM:9:@FM:5:@FM:7:@VM:3
Q=INSERT(R,1,0,0,"#")
```

The next example inserts a # before the third value of field 3 and sets the value of Q to FF1V2V#V3S6F9F5F7V3:

```
Q=INSERT(R,3,3;"#")
```

The next example inserts a value mark followed by a # after the last value in the field and sets Q to FF1V2V3S6F9V#F5F7V3:

```
Q=INSERT(R,4,-1,0,"#")
```

The next example inserts a # before the second subvalue of the second value of field 3 and sets Q to FF1V2S#V3S6F9F5F7V3:

```
Q=INSERT(R,3,2,2;"#")
```

INT function

Use the INT function to return the integer portion of an expression.

Syntax

INT (*expression*)

expression must evaluate to a numeric value. Any arithmetic operations specified are calculated using the full accuracy of the system. The fractional portion of the value is truncated, not rounded, and the integer portion remaining is returned.

If *expression* evaluates to the null value, null is returned.

Example

```
PRINT "123.45 ", INT(123.45)
 PRINT "454.95 ", INT(454.95)
```

This is the program output:

```
123.45      123
 454.95         454
```

ISNULL function

Use the ISNULL function to test whether a variable is the null value. If *variable* is the null value, 1 (true) is returned, otherwise 0 (false) is returned. This is the only way to test for the null value since the null value is not equal to any value, including itself.

Syntax

```
ISNULL (variable)
```

Example

```
X = @NULL
 Y = @NULL.STR
 PRINT ISNULL(X), ISNULL(Y)
```

This is the program output:

```
1 0
```

ISNULLS function

Use the ISNULLS function to test whether any element of *dynamic.array* is the null value. A dynamic array is returned, each of whose elements is either 1 (true) or 0 (false). If an element in *dynamic.array* is the null value, 1 is returned, otherwise 0 is returned. This is the only way to test for the null value since the null value is not equal to any value, including itself.

If you use the subroutine syntax, the resulting dynamic array is returned as *return.array*.

Syntax

```
ISNULLS (dynamic.array)
```

```
CALL -ISNULLS (return.array, dynamic.array)
```

Example

```
DA = ""
 FOR I = 1 TO 7
 DA := I:@FM
 IF I = 5 THEN DA := @NULL.STR:@FM
 NEXT I
 PRINT ISNULLS(DA)
```

This is the program output:

```
0F0F0F0F0F1F0F0F0
```

ITYPE function

Use the ITYPE function to return the value resulting from the evaluation of an I-type expression in a UniVerse file dictionary.

Syntax

```
ITYPE (i.type)
```

i.type is an expression evaluating to the contents of the compiled I-descriptor. The I-descriptor must have been compiled before the ITYPE function uses it, otherwise you get a run-time error message.

i.type can be set to the I-descriptor to be evaluated in several ways. One way is to read the I-descriptor from a file dictionary into a variable, then use the variable as the argument to the ITYPE function. If

the I-descriptor references a record ID, the current value of the system variable @ID is used. If the I-descriptor references field values in a data record, the data is taken from the current value of the system variable @RECORD.

To assign field values to @RECORD, read a record from the data file into @RECORD before invoking the `ITYPE` function.

If *i.type* evaluates to the null value, the `ITYPE` function fails and the program terminates with a run-time error message.

Example

This is the SUN.MEMBER file contents:

```
AW
 F1: ACCOUNTING
 TRX
 F1: MARKETING
 JXA
 F1: SALES
```

This is the DICT.ITME contents:

```
DEPARTMENTF1:D
 2:1
 3:
 4:
 5:10L
 6:L
```

This is the program source code:

```
OPEN 'SUN.MEMBER' TO FILE ELSE STOP
 OPEN 'DICT','SUN.MEMBER' TO D.FILE ELSE STOP
 *
 READ ITEM.ITYPE FROM D.FILE, 'DEPARTMENT' ELSE STOP
 *
 EXECUTE 'SELECT SUN.MEMBER'
 LOOP
 READNEXT @ID DO
 *
 READ @FRECORD FROM FILE, @ID THEN
 *
 PRINT @ID: "WORKS IN DEPARTMENT" ITYPE(ITEM.ITYPE)
 END
 REPEAT
 STOP
 END
```

This is the program output:

```
3 records selected to Select List #0
 FAW WORKS IN DEPARTMENT ACCOUNTING
 TRX WORKS IN DEPARTMENT MARKETING
 JXA WORKS IN DEPARTMENT SALES
```

KEYEDIT statement

Use the KEYEDIT statement to assign specific keyboard keys to the editing functions of the INPUT @ statement, and to the !EDIT.INPUT and !GET.KEY subroutines. KEYEDIT supports the following editing functions:

Syntax

KEYEDIT (*function, key*) [, (*function, key*)] …

- Left arrow (<—)
- Enter (Return)
- Back space
- Right arrow (—>)
- Insert character
- Delete character
- Insert mode on
- Insert mode off
- Clear field
- Erase to end-of-line
- Insert mode toggle

In addition to the supported editing functions, two codes exist to designate the Esc and function keys.

function is an expression that evaluates to a numeric code assigned to a particular editing function.

Code	Function
1	Function key
2	Left arrow (<—)
3	Return key
4	Back space
5	Esc key
6	Right arrow (—>)
7	Insert character
8	Delete character
9	Insert mode ON
10	Insert mode OFF
11	Clear from current position to end-of-line
12	Erase entire line
13	Insert mode toggle

key is an expression evaluating to a decimal value that designates the keyboard key to assign to the editing function. There are three key types, described in the following table:

Type	Decimal value	Description
Control	1 through 31	Single character control codes ASCII 1 through 31.

Type	Decimal value	Description
Escape	32 through 159	Consists of the characters defined by the Esc key followed by the ASCII value 0 through 127 (see Defining escape keys, on page 230).
Function	160 through 2,139,062,303	Consists of the characters defined by the FUNCTION key followed by the ASCII value 0 through 127. You can specify up to four ASCII values for complex keys (see Defining function keys, on page 230).

If either *function* or *key* evaluates to the null value or an empty string, the KEYEDIT statement fails, the program terminates, and a run-time error message is produced.

To define *key*, you must know the ASCII value generated by the keyboard on the terminal being used. Once you know the ASCII code sequence generated by a particular keyboard key, you can use one of the following three methods for deriving the numeric *key* value.

Defining control keys

A control key is one whose ASCII value falls within the range of 1 through 31. Generally keys of this type consist of pressing a keyboard key while holding down the Ctrl key. The *key* value is the ASCII code value, i.e., Ctrl-A is 1, Ctrl-M is 13, etc.

Defining escape keys

An escape key is one which consists of pressing the Esc key followed by a single ASCII value. The Esc key can be defined by issuing a KEYEDIT statement using a *function* value of 5 and the ASCII value of the escape character for the *key* parameter, e.g., KEYEDIT (5,27).

The *key* value for an escape key is derived by adding the ASCII value of the character following the Esc key and 32. The constant 32 is added to ensure that the final *key* value falls within the range of 32 to 159, i.e., Esc-**a** is 33 (1+32), Esc-**z** is 122 (90+32), Esc-**p** is 144 (112+32), and so on.

Defining function keys

A function key is similar to an escape key but consists of a function key followed by one or more ASCII values. The function key can be defined by issuing a KEYEDIT statement using a *function* value of 1 and the ASCII value of the function character for the *key* parameter, e.g., KEYEDIT(1,1).

Deriving the *key* value for a function key depends on the number of characters in the sequence the keyboard key generates. Because the KEYEDIT statement recognizes function keys that generate character sequences up to five characters long, the following method can be used to derive the *key* value.

Assume that keyboard key F7 generates the following character sequence:

Ctrl-A] 6 ~ <Return>

This character sequence is to be assigned to the Clear Field functionality of the INPUT @ statement. It can be broken into five separate characters, identified as follows:

Character	ASCII value	Meaning
Ctrl-A	1	The preamble character (defines the function key)
]	93	The first character
6	54	The second character
~	126	The third character
<Return>	10	The fourth character

First you define the function key value. Do this by issuing the KEYEDIT statement with a *function* value of 1 and with a *key* value defined as the ASCII value of the preamble character, i.e., KEYEDIT (1, 1).

Once you define the function key, the following formula is applied to the remaining characters in the sequence:

ASCII value * ($2^{(8*(\text{character position} - 1))}$

Using the example above:

Key	ASCII		Formula	Intermediate result				Final result
]	93	*	$(2^{(8*(1-1))}$	= 93 * (2^{0})	=	93 * 1	=	93
6	54	*	$(2^{(8*(2-1))}$	= 54 * (2^{8})	=	54 * 256	=	13,824
~	126	*	$(2^{(8*(3-1))}$	= 126 * (2^{16})	=	126 * 65,536	=	8,257,536
<cr>	10	*	$(2^{(8*(4-1))}$	= 10 * (2^{24})	=	10 * 16,777,216	=	167,772,160

								176,043,613
							+	160
								========
								176,043,773

The results of each calculation are then added together. Finally, the constant 160 is added to insure that the final *key* parameter value falls within the range of 160 through 2,139,062,303. For our example above, this would yield 176,043,613 + 160, or 176,043,773. To complete this example and assign this key to the Clear Field functionality, use the following KEYEDIT statement:

```
KEYEDIT (11, 176043773)
```

Historically, *key* values falling in the range of 160 through 287 included an implied Return, as there was no method for supporting multiple character sequences. With the support of multiple character sequences, you must now include the Return in the calculation for proper key recognition, with one exception. For legacy *key* values that fall within the range of 160 through 287, a Return is automatically appended to the end of the character sequence, yielding an internal *key* parameter of greater value.

A function key generates the character sequence:

Ctrl-A B <Return>

Before supporting multiple character sequences, this function key would have been defined as:

```
KEYEDIT (1, 1), (11, 225)
```

(1,1) defined the preamble of the function key, and (11, 225) defined the Clear-to-end-of-line key. The 225 value was derived by adding 160 to B (ASCII 65). The <Return> (ASCII 10) was implied. This can be shown by using the SYSTEM(1050) function to return the internal trap table contents:

#	Type	Value	Key
0	1	3	10
1	1	3	13
2	1	1	1
3	1	11	2785

The value 2785 is derived as follows:

```
(65 * 1) + (10 * 256) + 160 = 65 + 2560 + 160 = 2785.
```

Defining unsupported keys

You can use the KEYEDIT statement to designate keys that are recognized as unsupported by the !EDIT.INPUT subroutine. When the !EDIT.INPUT subroutine encounters an unsupported key, it sounds the terminal bell.

An unsupported key can be any of the three key types:

- Control key
- Escape key
- Function key

Define an unsupported key by assigning any negative decimal value for the *function* parameter.

The *key* parameter is derived as described earlier.

See the !EDIT.INPUT or !GET.KEY subroutine for more information.

Retrieving defined keys

The SYSTEM function(1050) returns a dynamic array of defined KEYEDIT, KEYEXIT statement and KEYTRAP statement keys. Field marks (ASCII 254) delimit the elements of the dynamic array. Each field in the dynamic array has the following structure:

key.type V *function.parameter* V *key.parameter*

key.type is one of the following values:

Value	Description
1	A KEYEDIT value
2	A KEYTRAP value
3	A KEYEXIT value
4	The INPUTNULL value
5	An unsupported value

function.parameter and *key.parameter* are the values passed as parameters to the associated statement, except for the INPUTNULL value.

Example

The following example illustrates the use of the KEYEDIT statement and the SYSTEM(1050) function:

```
KEYEDIT (1,1), (2,21), (3,13), (4,8), (6,6), (12,176043773)
 KEYTRAP (1,2)
 keys.dfn=SYSTEM(1050)
 PRINT "#","Type","Value","Key"
 XX=DCOUNT(keys.dfn,@FM)
 FOR I=1 TO XX
 print I-1,keys.dfn<I,1>,keys.dfn<I,2>,keys.dfn<I,3>
  NEXT I
```

The program output is:

```
#       Type              Value               Key
 0      1                   3                  10
 1      1                   3                  13
 2      1                  4                  8
 3      1                  1                   1
 4      1                   2                21
```

```
5     1                6                6
6     1                 12              176043773
7     2                1              2
```

KEYEXIT statement

Use the KEYEXIT statement to specify exit traps for the keys assigned specific functions by the KEYEDIT statement. When an exit trap key is typed, the variable being edited with the INPUT @ statement or the !EDIT.INPUT subroutine remains in its last edited state. Use the KEYTRAP statement to restore the variable to its initial state.

Syntax

KEYEXIT (*value, key*) [, (*value, key*)] …

value is an expression that specifies a user-defined trap number for each key assigned by the KEYEDIT statement.

key is a decimal value that designates the specific keyboard key assigned to the editing function. There are three key types, described in the following table:

Type	Decimal value	Description
Control	1 through 31	Single character control codes ASCII 1 through 31.
Escape	32 through 159	Consists of the characters defined by the Esc key followed by the ASCII value 0 through 127.
Function	160 through 2,139,062,303	Consists of the characters defined by the function key followed by the ASCII value 0 through 127. A maximum of four ASCII values can be specified for complex keys.

See the KEYEDIT statement for how to derive the decimal value of control, escape, and function keys.

If either the *value* or *key* expression evaluates to the null value or an empty string, the KEYEXIT statement fails, the program terminates, and a run-time error message is produced.

KEYEXIT sets the STATUS function to the trap number of any trap key typed by the user.

Examples

The following example sets up Ctrl-B as an exit trap key. The STATUS function is set to 1 when the user types the key.

```
KEYEXIT (1,2)
```

The next example sets up Ctrl-K as an exit trap key. The `STATUS` function is set to 2 when the user types the key.

```
KEYEXIT (2,11)
```

KEYIN function

Use the `KEYIN` function to read a single character from the input buffer and return it. All UniVerse special character handling (such as case inversion, erase, kill, and so on) is disabled. UNIX special

character handling (processing of interrupts, XON/XOFF, conversion of CR to LF, and so on) still takes place.

Calculations are based on display length rather than character length.

No arguments are required with the `KEYIN` function; however, parentheses are required.

Syntax

KEYIN ()

KEYTRAP statement

Use the KEYTRAP statement to specify traps for the keys assigned specific functions by the KEYEDIT statement. When a trap key is typed, the variable being edited with the INPUT @ statement or the ! EDIT.INPUT subroutine is restored to its initial state. Use the KEYEXIT statement to leave the variable in its last edited state.

Syntax

KEYTRAP (*value, key*) [, (*value, key*)] …

value is an expression that evaluates to a user-defined trap number for each key assigned by the KEYEDIT statement.

key is a decimal value which designates the specific keyboard key assigned to the editing function. There are three key types, described in the following table:

Type	Decimal value	Description
Control	1 through 31	Single character control codes ASCII 1 through 31.
Escape	32 through 159	Consists of the characters defined by the Esc key followed by the ASCII value 0 through 127.
Function	160 through 2,139,062,303	Consists of the characters defined by the function key followed by the ASCII value 0 through 127. A maximum of four ASCII values may be specified for complex keys.

See the KEYEDIT statement, on page 229 for how to derive the decimal value of control, escape, and function keys.

If either the *value* or *key* expression evaluates to the null value or an empty string, the KEYEXIT statement fails, the program terminates, and a run-time error message is produced.

KEYTRAP sets the STATUS function to the trap number of any trap key typed by the user.

Examples

The following example sets up Ctrl-B as a trap key. The `STATUS` function is set to 1 when the user types the key.

```
KEYTRAP (1, 2)
```

The next example defines function key values for the F1, F2, F3, and F4 keys on a Wyse 50 terminal:

```
KEYEDIT (1,1)
 KEYTRAP (1,224), (2,225), (3,226), (4,227)
 PRINT @(-1)
 VALUE = "KEY"
```

```
INPUT @ (10,10):VALUE
X=STATUS()
BEGIN CASE
   CASE X = 1
   PRINT "FUNCTION KEY 1"
   CASE X =2
   PRINT "FUNCTION KEY 2"
   CASE X =3
   PRINT "FUNCTION KEY 3"
   CASE X =4
   PRINT "FUNCTION KEY 4"
END CASE
PRINT VALUE
STOP
END
```

LEFT function

Use the LEFT function to extract a substring comprising the first *n* characters of a string, without specifying the starting character position. It is equivalent to the following substring extraction operation:

string [1, *length*]

If *string* evaluates to the null value, null is returned. If *n* evaluates to the null value, the LEFT function fails and the program terminates with a run-time error message.

Syntax

LEFT (*string, n*)

Example

```
PRINT LEFT("ABCDEFGH",3)
```

This is the program output:

```
ABC
```

LEN function

Use the LEN function to return the number of characters in *string*. Calculations are based on character length rather than display length.

Syntax

LEN (*string*)

string must be a string value. The characters in *string* are counted, and the count is returned.

The LEN function includes all blank spaces, including trailing blanks, in the calculation.

If *string* evaluates to the null value, 0 is returned.

If NLS is enabled, use the LENDP function to return the length of a string in display positions rather than character length. For more information about display length, see the *UniVerse NLS Guide*.

Example

```
P="PORTLAND, OREGON"
 PRINT "LEN(P)= ",LEN(P)
 *
 NUMBER=123456789
 PRINT "LENGTH OF NUMBER IS ",LEN(NUMBER)
```

This is the program output:

```
LEN(P)=     16
 LENGTH OF NUMBER IS                9
```

LENDP function

In NLS mode, use the LENDP function to return the number of display positions occupied by *string* when using the specified map. Calculations are based on display length rather than character length.

Syntax

LENDP (*string* [,*mapname*])

string must be a string value. The display length of *string* is returned.

mapname is the name of an installed map. If *mapname* is not installed, the character length of *string* is returned.

If *mapname* is omitted, the map associated with the channel activated by PRINTER ON is used, otherwise it uses the map for print channel 0. You can also specify *mapname* as CRT, AUX, LPTR, and OS. These values use the maps associated with the terminal, auxiliary printer, print channel 0, or the operating system, respectively. If you specify *mapname* as NONE, the string is not mapped.

Any unmappable characters in *string* have a display length of 1.

The LENDP function includes all blank spaces, including trailing blanks, in the calculation.

If *string* evaluates to the null value, 0 is returned.

If you use the LENDP function with NLS disabled, the program behaves as if the LEN function is used. See the LEN function, on page 235 to return the length of a string in character rather than display positions.

For more information about display length, see the *UniVerse NLS Guide*.

LENS function

Use the LENS function to return a dynamic array of the number of display positions in each element of *dynamic.array*. Calculations are based on character length rather than display length.

Syntax

LENS (*dynamic.array*)

CALL **-LENS** (*return.array, dynamic.array*)

CALL **!LENS** (*return.array, dynamic.array*)

Each element of *dyamic.array* must be a string value. The characters in each element of *dynamic.array* are counted, and the counts are returned.

The LENS function includes all blank spaces, including trailing blanks, in the calculation.

If *dynamic.array* evaluates to the null value, 0 is returned. If any element of *dynamic.array* is null, 0 is returned for that element.

If NLS is enabled, use the LENSDP function to return a dynamic array of the number of characters in each element of *dynamic.array* in display positions. For more information about display length, see the *UniVerse NLS Guide*.

LENSDP function

In NLS mode, use the LENSDP function to return a dynamic array of the number of display positions occupied by each element of *dynamic.array*. Calculations are based on display length rather than character length.

Syntax

```
LENSDP (dynamic.array [, mapname] )
CALL -LENSDP (return.array, dynamic.array [ ,mapname] )
CALL !LENSDP (return.array, dynamic.array [,mapname] )
```

Each element of *dynamic.array* must be a string value. The display lengths of each element of *dynamic.array* are counted, and the counts are returned.

mapname is the name of an installed map. If *mapname* is not installed, the character length of *string* is returned.

If *mapname* is omitted, the map associated with the channel activated by PRINTER ON is used, otherwise it uses the map for print channel 0. You can also specify *mapname* as CRT, AUX, LPTR, and OS. These values use the maps associated with the terminal, auxiliary printer, print channel 0, or the operating system, respectively. If you specify *mapname* as NONE, the string is not mapped.

Any unmappable characters in *dynamic.array* have a display length of 1.

The LENSDP function includes all blank spaces, including trailing blanks, in the calculation.

If *dynamic.array* evaluates to the null value, 0 is returned. If any element of *dynamic.array* is null, 0 is returned for that element.

If you use the subroutine syntax, the resulting dynamic array is returned as *return.array*.

If you use the LENSDP function with NLS disabled, the program behaves as if the LENS function is used. See the LENS function to return the length of a string in character length rather than display length.

For more information about display length, see the *UniVerse NLS Guide*.

LES function

Use the LES function to test if elements of one dynamic array are less than or equal to the elements of another dynamic array.

Syntax

```
LES (array1, array2)
```

```
CALL -LES (return.array, array1, array2)
CALL !LES (return.array, array1, array2)
```

Each element of *array1* is compared with the corresponding element of *array2*. If the element from *array1* is less than or equal to the element from *array2*, a 1 is returned in the corresponding element of a new dynamic array. If the element from *array1* is greater than the element from *array2*, a 0 is returned. If an element of one dynamic array has no corresponding element in the other dynamic array, the undefined element is evaluated as empty, and the comparison continues.

If either of a corresponding pair of elements is the null value, null is returned for that element.

If you use the subroutine syntax, the resulting dynamic array is returned as *return.array*.

LET statement

Use the LET statement to assign the value of *expression* to *variable*.

Syntax

```
[LET] variable = expression
```

Example

```
 LET A=55
  LET B=45
  LET C=A+B
  LET D="55+45="
  LET E=D:C
  PRINT E
```

This is the program output:

```
55+45=100
```

LN function

Use the `LN` function to calculate the natural logarithm of the value of an expression, using base "e". The value of "e" is approximately 2.71828. *expression* must evaluate to a numeric value greater than 0.

If *expression* is 0 or negative, 0 is returned and a warning is printed. If *expression* evaluates to the null value, null is returned.

Syntax

```
LN (expression)
```

Example

```
PRINT LN(6)
```

This is the program output:

```
1.7918
```

loadSecurityContext function

The `loadSecurityContext`() function loads a saved security context record into the current session.

The *name* and *passPhrase* parameters are needed to retrieve and decrypt the saved context. An internal data structure is created and its handle is returned in the *context* parameter.

Syntax

loadSecurityContext(*context*, *name*, *passPhrase*)

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
context	The security context handle.
name	String containing the name of the saved context.
PassPhrase	String containing the *passPhrase* needed to decrypt the saved data.

Return code status

The following table describes the status of each return code.

Return code	Status
0	Success.
1	Context record does not exist.
2	Context record could not be accessed (for example, wrong password).
3	Invalid content (file was not saved by the `saveSecurityContext`() function).
4	Other problems that caused context load failure. Refer to the log file for more information.

LOCALEINFO function

In NLS mode, use the `LOCALEINFO` function to retrieve the settings of the current locale.

Syntax

LOCALEINFO (*category*)

category is one of the following tokens that are defined in the UniVerse include file UVNLSLOC.H:

Category	Description
UVLC$TIME UVLC$NUMERIC UVLC$MONETARY UVLC$CTYPE UVLC$COLLATE	Each token returns a dynamic array containing the data being used by the specified category. The meaning of the data depends on the category; field 1 is always the name of the category or the value OFF. OFF means that locale support is disabled for a category. The elements of the array are separated by field marks.

Category	Description
UVLC$WEIGHTS	Returns the weight table.
UVLC$INDEX	Returns information about the hooks defined for the locale.

If the specified category is set to OFF, LOCALEINFO returns the string OFF.

If the LOCALEINFO function fails to execute, LOCALEINFO returns one of the following:

Error	Description
LCE$NO.LOCALES	NLS locales are not in force.
LCE $BAD.CATEGORY	Category is invalid.

For more information about locales, see the *UniVerse NLS Guide*.

Example

The following example shows the contents of the multivalued DAYS field when the locale FR-FRENCH is current. Information for LCT$DAYS is contained in the UVNLSLOC.H file in the INCLUDE directory in the UV account directory.

```
category.info = LOCALEINFO(LC$TIME)
 PRINT category.info<LCT$DAYS>
```

This is the program output:

```
lundi}mardi}mercredi}jeudi}vendredi}samedi}dimanche
```

LOCATE statement (IDEAL and REALITY syntax)

Use the LOCATE statement to search *dynamic.array* for a field, value, or subvalue. LOCATE returns a value indicating one of the following:

Syntax

```
LOCATE expression IN dynamic.array [< field# [, value#] >]
   [, start] [BY seq]
    SETTING variable
    {THEN statements [ELSE statements] | ELSE statements}
```

- Where *expression* was found in *dynamic.array*
- Where *expression* should be inserted in *dynamic.array* if it was not found

The search can start anywhere in *dynamic.array*.

> **Note:** The REALITY syntax of LOCATE works in IDEAL, REALITY, IN2, and PICK flavors by default. To make the INFORMATION syntax of LOCATE available in these flavors, use the INFO.LOCATE option of $OPTIONS statement. To make the REALITY syntax of LOCATE available in INFORMATION and PIOPEN flavors, use $OPTIONS –INFO.LOCATE.

expression evaluates to the content of the field, value, or subvalue to search for in *dynamic.array*. If *expression* or *dynamic.array* evaluates to the null value, *variable* is set to 0 and the ELSE statements are executed. If *expression* and *dynamic.array* both evaluate to empty strings, *variable* is set to 1 and the THEN statements are executed.

field# and *value#* are delimiter expressions that restrict the scope of the search. If you do not specify *field#*, *dynamic.array* is searched field by field. If you specify *field#* but not *value#*, the specified field is searched value by value. If you specify *field#* and *value#*, the specified value is searched subvalue by subvalue.

start is an expression that evaluates to a number specifying the field, value, or subvalue from which to start the search.

Case	Description
Case 1:	If *field#* and *value#* are omitted, the search starts in *dynamic.array* at the field specified by *start*. If *start* is also omitted, the search starts at field 1 of *dynamic.array*.
Case 2:	If only *field#* is specified and it is greater than 0, the search starts at the value specified by *start*. If *start* is also omitted, the search starts at value 1 in *field#*. If *field#* is less than or equal to 0, both *field#* and *value#* are ignored.
Case 3:	If both *field#* and *value#* are specified, the search starts at the subvalue specified by *start*. If *start* is also omitted, the search starts at subvalue 1 of *value#*, in the field specified by *field#*. If *field#* is greater than 0, but *value#* is less than or equal to 0, LOCATE behaves as though only *field#* is specified.

If a field, value, or subvalue containing *expression* is found, *variable* returns the index of the located field, value, or subvalue relative to the start of *dynamic.array*, *field#*, or *value#*, respectively, not relative to the start of the search. If a field, value, or subvalue containing *expression* is not found, *variable* is set to the number of fields, values, or subvalues in the array plus 1, and the ELSE statements are executed. The format of the ELSE statement is the same as that used in the IF…THEN statement.

If *field#*, *value#*, or *start* evaluates to the null value, the LOCATE statement fails and the program terminates with a run-time error message.

variable stores the index of *expression*. *variable* returns a field number, value number, or subvalue number, depending on the delimiter expressions used. *variable* is set to a number representing one of the following:

- The index of the element containing *expression*, if such an element is found
- An index that can be used in an INSERT function to create a new element with the value specified by *expression*

The search stops when one of the following conditions is met:

- A field containing *expression* is found.
- The end of the dynamic array is reached.
- A field that is higher or lower, as specified by *seq*, is found.

If the elements to be searched are sorted in one of the ascending or descending ASCII sequences listed below, you can use the BY *seq* expression to end the search. The search ends at the place where *expression* should be inserted to maintain the ASCII sequence, rather than at the end of the list of specified elements.

Use the following values for *seq* to describe the ASCII sequence being searched:

Value	Description
AL or A	Ascending, left-justified (standard alphanumeric sort)
AR	Ascending, right-justified (numeric sort)
DL or D	Descending, left-justified (standard alphanumeric sort)
DR	Descending, right-justified (numeric sort)

seq does not reorder the elements in *dynamic.array*; it specifies the terminating conditions for the search. If a *seq* expression is used and the elements are not in the sequence indicated by *seq*, an element with the value of *expression* may not be found. If *seq* evaluates to the null value, the statement fails and the program terminates.

If NLS is enabled, the LOCATE statement with a BY *seq* expression uses the Collate convention as specified in the NLS.LC.COLLATE file to determine the sort order for characters with ascending or descending sequences. The Collate convention defines rules for casing, accents, and ordering. For more information about how NLS calculates the order, see the *UniVerse NLS Guide*.

Examples

The examples show the REALITY flavor of the LOCATE statement. A field mark is shown by F, a value mark is shown by V, and a subvalue mark is shown by S.

```
 Q='X':@SM:"$":@SM:'Y':@VM:'Z':@SM:4:@SM:2:@VM:'B'
 PRINT "Q= ":Q
LOCATE "$" IN Q <1> SETTING WHERE ELSE PRINT 'ERROR'
 PRINT "WHERE= ",WHERE
LOCATE "$" IN Q <1,1>   SETTING HERE ELSE PRINT 'ERROR'
 PRINT "HERE= ", HERE
 NUMBERS=122:@FM:123:@FM:126:@FM:130:@FM
 PRINT "BEFORE INSERT, NUMBERS= ",NUMBERS
 NUM= 128
 LOCATE NUM IN NUMBERS BY "AR" SETTING X ELSE
 NUMBERS = INSERT(NUMBERS,X,0,0,NUM)
    PRINT "AFTER INSERT, NUMBERS= ",NUMBERS
 END
```

This is the program output:

```
Q= XS$SYVZS4S2VB
ERROR
WHERE= 4
HERE= 2
BEFORE INSERT, NUMBERS= 122F123F126F130F
AFTER INSERT, NUMBERS=  122F123F126F128F130F
```

LOCATE statement (INFORMATION syntax)

Use the LOCATE statement to search *dynamic.array* for a field, value, or subvalue. LOCATE returns a value indicating one of the following:

Syntax

```
LOCATE expression IN dynamic.array <field# [, value#  [, subvalue#]] >
   [BY seq] SETTING variable
        {THEN statements [ELSE statements] | ELSE statements}
```

- Where *expression*was found in *dynamic.array*
- Where *expression* should be inserted in *dynamic.array* if it was not found

The search can start anywhere in *dynamic.array*.

Note: The INFORMATION syntax of LOCATE works in INFORMATION and PIOPEN flavors by default. To make the REALITY syntax of LOCATE available in INFORMATION and PIOPEN flavors, use $OPTIONS -INFO.LOCATE.

expression evaluates to the contents of the field, value, or subvalue to search for in *dynamic.array*. If *expression* or *dynamic.array* evaluates to the null value, *variable* is set to 0 and the ELSE statements are executed. If *expression* and *dynamic.array* both evaluate to empty strings, *variable* is set to 1 and the THEN statements are executed.

field#, *value#*, and *subvalue#* are delimiter expressions specifying where to start the search in *dynamic.array*. If you specify *field#* only, *dynamic.array* is searched field by field. If you specify *field#* and *value#* only, the specified field is searched value by value. If you also specify *subvalue#*, the specified value is searched subvalue by subvalue.

When the search is field by field, each field is treated as a single string, including any value marks and subvalue marks. When the search is value by value, each value is treated as a single string, including any subvalue marks. For the search to be successful, *expression* must match the entire contents of the field, value, or subvalue found, including any embedded value marks or subvalue marks.

Case	Description
Case 1:	If both *value#* and *subvalue#* are omitted or are both less than or equal to 0, the search starts at the field indicated by *field#*.
Case 2:	If *subvalue#* is omitted or is less than or equal to 0, the search starts at the value indicated by *value#*, in the field indicated by *field#*. If *field#* is less than or equal to 0, *field#* defaults to 1.
Case 3:	If *field#*, *value#*, and *subvalue#* are all specified and are all nonzero, the search starts at the subvalue indicated by *subvalue#*, in the value specified by *value#*, in the field specified by *field#*. If *field#* or *value#* are less than or equal to 0, they default to 1.

If a field, value, or subvalue containing *expression* is found, *variable* is set to the index of the located field relative to the start of *dynamic.array*, the field, or the value, respectively, not relative to the start of the search.

If no field containing *expression* is found, *variable* is set to the number of the field at which the search terminated, and the ELSE statements are executed. If no value or subvalue containing *expression* is found, *variable* is set to the number of values or subvalues plus 1, and the ELSE statements are executed. If *field#*, *value#*, or *subvalue#* is greater than the number of fields in *dynamic.array*, *variable* is set to the value of *field#*, *value#*, or *subvalue#*, respectively, and the ELSE statements are executed. The format of the ELSE statement is the same as that used in the IF…THEN statement.

If any delimiter expression evaluates to the null value, the LOCATE statement fails and the program terminates with a run-time error message.

variable stores the index of *expression*. *variable* returns a field number, value number, or a subvalue number, depending on the delimiter expressions used. *variable* is set to a number representing one of the following:

- The index of the element containing *expression*, if such an element is found
- An index that can be used in an `INSERT` function to create a new element with the value specified by *expression*.

The search stops when one of the following conditions is met:

- A field containing *expression* is found.
- The end of the dynamic array is reached.
- A field that is higher or lower, as specified by *seq*, is found.

If the elements to be searched are sorted in one of the ascending or descending ASCII sequences listed below, you can use the BY *seq* expression to end the search. The search ends at the place where *expression* should be inserted to maintain the ASCII sequence, rather than at the end of the list of specified elements.

Use the following values for *seq* to describe the ASCII sequence being searched:

Value	Description
AL or A	Ascending, left-justified (standard alphanumeric sort)
AR	Ascending, right-justified (numeric sort)
DL or D	Descending, left-justified (standard alphanumeric sort)
DR	Descending, right-justified (numeric sort)

seq does not reorder the elements in *dynamic.array*; it specifies the terminating conditions for the search. If a *seq* expression is used and the elements are not in the sequence indicated by *seq*, an element with the value of *expression* may not be found. If *seq* evaluates to the null value, the statement fails and the program terminates.

If NLS is enabled, the LOCATE statement with a BY *seq* expression uses the Collate convention as specified in the NLS.LC.COLLATE file to determine the sort order for characters with ascending or descending sequences. The Collate convention defines rules for casing, accents, and ordering. For more information about how NLS calculates the order, see the *UniVerse NLS Guide*.

Examples

The examples show the INFORMATION flavor of the LOCATE statement. A field mark is shown by F, a value mark is shown by V, and a subvalue mark is shown by S.

```
 Q='X':@SM:"$":@SM:'Y':@VM:'Z':@SM:4:@SM:2:@VM:'B'
 PRINT "Q= ":Q
LOCATE "$" IN Q <1> SETTING WHERE ELSE PRINT 'ERROR'
 PRINT "WHERE= ",WHERE
LOCATE "$" IN Q <1,1>   SETTING HERE ELSE PRINT 'ERROR'
 PRINT "HERE= ", HERE
 NUMBERS=122:@FM:123:@FM:126:@FM:130:@FM
 PRINT "BEFORE INSERT, NUMBERS= ",NUMBERS
 NUM= 128
 LOCATE NUM IN NUMBERS <2> BY "AR" SETTING X ELSE
    NUMBERS = INSERT(NUMBERS,X,0,0,NUM)
    PRINT "AFTER INSERT, NUMBERS= ",NUMBERS
 END
```

This is the program output:

```
Q= XS$SYVZS4S2VB
ERROR
 WHERE= 2
ERROR
 HERE=  4
BEFORE INSERT, NUMBERS= 122F123F126F130F
AFTER INSERT, NUMBERS=  122F123F126F128F130F
```

LOCATE statement (PICK syntax)

Use the LOCATE statement to search *dynamic.array* for a field, value, or subvalue. LOCATE returns a value indicating one of the following:

Syntax

```
LOCATE (expression, dynamic.array [, field# [, value#]] ; variable [;seq] )
      { THEN statements [ELSE statements] | ELSE statements }
```

- Where *expression* was found in *dynamic.array*
- Where *expression* should be inserted in *dynamic.array* if it was not found

> **Note:** The PICK syntax of LOCATE works in all flavors of UniVerse.

expression evaluates to the content of the field, value, or subvalue to search for in *dynamic.array*. If *expression* or *dynamic.array* evaluates to the null value, *variable* is set to 0 and the ELSE statements are executed. If *expression* and *dynamic.array* both evaluate to empty strings, *variable* is set to 1 and the THEN statements are executed.

field# and *value#* are delimiter expressions that restrict the scope of the search. If you do not specify *field#*, *dynamic.array* is searched field by field. If you specify *field#* but not *value#*, the specified field is searched value by value. If you specify *field#* and *value#*, the specified value is searched subvalue by subvalue.

When the search is field by field, each field is treated as a single string, including any value marks and subvalue marks. When the search is value by value, each value is treated as a single string, including any subvalue marks. For the search to be successful, *expression* must match the entire contents of the field, value, or subvalue found, including any embedded value marks or subvalue marks.

Case	Result
Case 1:	If *field#* and *value#* are omitted, the search starts at the first field in *dynamic.array*.
Case 2:	If only *field#* is specified and it is greater than 0, the search starts at the first value in the field indicated by *field#*. If *field#* is less than or equal to 0, both *field#* and *value#* are ignored.
Case 3:	If both *field#* and *value#*are specified, the search starts at the first subvalue in the value specified by *value#*, in the field specified by *field#*. If *field#* is greater than 0, but *value#* is less than or equal to 0, LOCATE behaves as though only *field#* is specified.

If a field, value, or subvalue containing *expression* is found, *variable* returns the index of the located field, value, or subvalue relative to the start of *dynamic.array*, *field#*, or *value#*, respectively, not relative to the start of the search. If a field, value, or subvalue containing *expression* is not found, *variable* is set to the number of fields, values, or subvalues in the array plus 1, and the ELSE statements are executed. The format of the ELSE statement is the same as that used in the IF…THEN statement.

If *field#* or *value#* evaluates to the null value, the LOCATE statement fails and the program terminates with a run-time error message.

variable stores the index of *expression*. *variable* returns a field number, value number, or a subvalue number, depending on the delimiter expressions used. *variable* is set to a number representing one of the following:

- The index of the element containing *expression*, if such an element is found
- An index that can be used in an `INSERT` function to create a new element with the value specified by *expression*

The search stops when one of the following conditions is met:

- A field containing *expression* is found.
- The end of the dynamic array is reached.
- A field that is higher or lower, as specified by *seq*, is found.

If the elements to be searched are sorted in one of the ascending or descending ASCII sequences listed below, you can use the BY *seq* expression to end the search. The search ends at the place where *expression* should be inserted to maintain the ASCII sequence, rather than at the end of the list of specified elements.

Use the following values for *seq* to describe the ASCII sequence being searched:

Value	Description
AL or A	Ascending, left-justified (standard alphanumeric sort)
AR	Ascending, right-justified (numeric sort)
DL or D	Descending, left-justified (standard alphanumeric sort)
DR	Descending, right-justified (numeric sort)

seq does not reorder the elements in *dynamic.array*; it specifies the terminating conditions for the search. If a *seq* expression is used and the elements are not in the sequence indicated by *seq*, an element with the value of *expression* may not be found. If *seq* evaluates to the null value, the statement fails and the program terminates.

If NLS is enabled, the LOCATE statement with a *seq* expression uses the Collate convention as specified in the NLS.LC.COLLATE file to determine the sort order for characters with ascending or descending sequences. The Collate convention defines rules for casing, accents, and ordering. For more information about how NLS calculates the order, see the *UniVerse NLS Guide*.

Examples

The examples show the PICK flavor of the LOCATE statement. A field mark is shown by F, a value mark is shown by V, and a subvalue mark is shown by S.

```
  Q='X':@SM:"$":@SM:'Y':@VM:'Z':@SM:4:@SM:2:@VM:'B'
  PRINT "Q= ":Q
LOCATE ("$", Q, 1; WHERE) ELSE PRINT 'ERROR'
  PRINT "WHERE= ",WHERE
LOCATE ("$", Q, 1, 1; HERE) ELSE PRINT 'ERROR'
  PRINT "HERE= ", HERE
  NUMBERS=122:@FM:123:@FM:126:@FM:130:@FM
  PRINT "BEFORE INSERT, NUMBERS= ",NUMBERS
  NUM= 128
  LOCATE (NUM, NUMBERS; X; "AR") ELSE
     NUMBERS = INSERT(NUMBERS,X,0,0,NUM)
     PRINT "AFTER INSERT, NUMBERS= ",NUMBERS
  END
```

This is the program output:

```
Q= XS$SYVZS4S2VB
ERROR
 WHERE= 4
HERE=  2
BEFORE INSERT, NUMBERS= 122F123F126F130F
AFTER INSERT, NUMBERS=  122F123F126F128F130F
```

LOCK statement

Use the LOCK statement to protect specified user-defined resources or events against unauthorized use or simultaneous data file access by different users.

Syntax

LOCK *expression* [THEN *statements*] [ELSE *statements*]

There are 64 public semaphore locks in the UniVerse system. They are task synchronization tools but have no intrinsic definitions. You must define the resource or event associated with each semaphore, ensuring that there are no conflicts in definition or usage of these semaphores throughout the entire system.

expression evaluates to a number in the range of 0 through 63 that specifies the lock to be set. A program can reset a lock any number of times and with any frequency desired. If *expression* evaluates to the null value, the LOCK statement fails and the program terminates with a run-time error message.

If program B tries to set a lock already set by program A, execution of program B is suspended until the first lock is released by program A; execution of program B then continues.

The ELSE clause provides an alternative to this procedure. When a LOCK statement specifies a lock that has already been set, the ELSE clause is executed rather than program execution being suspended.

Program termination does not automatically release locks set in the program. Each LOCK statement must have a corresponding UNLOCK statement. If a program locks the same semaphore more than once during its execution, a single UNLOCK statement releases that semaphore.

The UNLOCK statement can specify the expression used in the LOCK statement to be released. If no expression is used in the UNLOCK statement, all locks set by the program are released.

Alternatively, locks can be released by logging off the system or by executing either the `QUIT` command or the `CLEAR.LOCKS` command.

You can check the status of locks with the `LIST.LOCKS` command; this lists the locks on the screen. The unlocked state is indicated by 0. The locked state is indicated by a number other than 0 (including both positive and negative numbers). The number is the unique signature of the user who has set the lock.

Note: The LOCK statement protects user-defined resources only. TheREADL statement, READU statement, READVL statement, READVU statement, MATREADL statement, and MATREADU statement use a different method of protecting files and records.

Example

The following example sets lock 60, executes the `LIST.LOCKS` command, then unlocks all locks set by the program:

```
LOCK 60 ELSE PRINT "ALREADY LOCKED"
 EXECUTE "LIST.LOCKS"
 UNLOCK
```

The program displays the `LIST.LOCKS` report. Lock 60 is set by user 4.

```
0:--    1:--    2:--    3:--    4:--    5:--    6:--    7:--
 8:--   9:-- 10:-- 11:-- 12:-- 13:-- 14:-- 15:--
 16:-- 17:-- 18:-- 19:-- 20:-- 21:-- 22:-- 23:--
 24:-- 25:-- 26:-- 27:-- 28:-- 29:-- 30:-- 31:--
```

```
32:-- 33:-- 34:-- 35:-- 36:-- 37:-- 38:-- 39:--
40:-- 41:-- 42:-- 43:-- 44:-- 45:-- 46:-- 47:--
48:-- 49:-- 50:-- 51:-- 52:-- 53:-- 54:-- 55:--
56:-- 57:-- 58:-- 59:-- 60:4   61:-- 62:-- 63:--
```

LOOP statement

Use the LOOP statement to start a LOOP…REPEAT program loop. A program loop is a series of statements that executes for a specified number of repetitions or until specified conditions are met.

Syntax

```
LOOP
       [loop.statements]
       [CONTINUE | EXIT]
 [{WHILE | UNTIL} expression [DO]]
[loop.statements]
  [CONTINUE | EXIT]
 REPEAT
```

Use the WHILE clause to indicate that the loop should execute repeatedly as long as the WHILE expression evaluates to true (1). When the WHILE expression evaluates to false (0), repetition of the loop stops, and program execution continues with the statement following the REPEAT statement.

Use the UNTIL clause to put opposite conditions on the LOOP statement. The UNTIL clause indicates that the loop should execute repeatedly as long as the UNTIL expression evaluates to false (0). When the UNTIL expression evaluates to true (1), repetition of the loop stops, and program execution continues with the statement following the REPEAT statement.

If a WHILE or UNTIL expression evaluates to the null value, the condition is false.

expression can also contain a conditional statement. Any statement that takes a THEN or an ELSE clause can be used as *expression*, but without the THEN or ELSE clause. When the conditional statement would execute the ELSE clause, *expression* evaluates to false; when the conditional statement would execute the THEN clause, *expression* evaluates to true. A LOCKED clause is not supported in this context.

You can use multiple WHILE and UNTIL clauses in a LOOP…REPEAT loop. You can also nest LOOP…REPEAT loops. If a REPEAT statement is encountered without a previous LOOP statement, an error occurs during compilation.

Use the CONTINUE statement within LOOP…REPEAT to transfer control to the next iteration of the loop from any point in the loop.

Use the EXIT statement within LOOP…REPEAT to terminate the loop from any point within the loop.

Although it is possible to exit the loop by means other than the conditional WHILE and UNTIL statements (for example, by using GOTO or GOSUB in the DO statements), it is not recommended. Such a programming technique is not in keeping with good structured programming practice.

Examples

Source lnes	Program output
X=0 LOOP UNTIL X>4 DO PRINT "X= ",X X=X+1 REPEAT	X= 0 X= 1 X= 2 X= 3 X= 4
A=20 LOOP PRINT "A= ", A A=A-1 UNTIL A=15 REPEAT	A= 20 A= 19 A= 18 A= 17 A= 16

Q=3
LOOP
PRINT "Q= ",Q
WHILE Q DO
Q=Q-1
REPEAT
	Q= 3
	Q= 2
	Q= 1
	Q= 0

EXECUTE "SELECT VOC FIRST 5"
MORE=1
LOOP
READNEXT ID
ELSE MORE=0
WHILE MORE DO
PRINT ID
REPEAT
	5 record(s) selected to SELECT list #0.
	LOOP
	HASH.TEST
	QUIT.KEY
	P
	CLEAR.LOCKS

EXECUTE "SELECT VOC FIRST 5"
LOOP
WHILE READNEXT ID DO
PRINT ID
REPEAT
	5 record(s) selected to SELECT list #0.
	LOOP
	HASH.TEST
	QUIT.KEY
	P
	CLEAR.LOCKS

LOWER function

Use the `LOWER` function to return a value equal to *expression*, except that system delimiters which appear in *expression* are converted to the next lower-level delimiter: field marks are changed to value

marks, value marks are changed to subvalue marks, and so on. If *expression* evaluates to the null value, null is returned.

Syntax

LOWER (*expression*)

The conversions are:

IM	CHAR(255)	to	FM	CHAR(254)
FM	CHAR(254)	to	VM	CHAR(253)
VM	CHAR(253)	to	SM	CHAR(252)
SM	CHAR(252)	to	TM	CHAR(251)
TM	CHAR(251)	to		CHAR(250)
	CHAR(250)	to		CHAR(249)
	CHAR(249)	to		CHAR(248)

PIOPEN flavor

In PIOPEN flavor, the delimiters that can be lowered are CHAR(255) through CHAR(252). All other characters are left unchanged. You can obtain PIOPEN flavor for the LOWER function by:

- Compiling your program in a PIOPEN flavor account
- Specifying the $OPTIONS INFO.MARKS statement

Examples

In the following examples an item mark is shown by I, a field mark is shown by F, a value mark is shown by V, a subvalue mark is shown by S, and a text mark is shown by T. CHAR(250) is shown as Z.

The following example sets A to DDFEEV123V77:

```
A= LOWER('DD':IM'EE':FM:123:FM:777)
```

The next example sets B to 1F2S3V4T5:

```
B= LOWER(1:IM:2:VM:3:FM:4:SM:5)
```

The next example sets C to 999Z888:

```
C= LOWER(999:TM:888)
```

LTS function

Use the LTS function to test if elements of one dynamic array are less than elements of another dynamic array.

Syntax

LTS (*array1*, *array2*)

CALL **-LTS** (*return.array*, *array1*, *array2*)

CALL **!LTS** (*return.array*, *array1*, *array2*)

Each element of *array1* is compared with the corresponding element of *array2*. If the element from *array1* is less than the element from *array2*, a 1 is returned in the corresponding element of a new

dynamic array. If the element from *array1* is greater than or equal to the element from *array2*, a 0 is returned. If an element of one dynamic array has no corresponding element in the other dynamic array, the undefined element is evaluated as an empty string, and the comparison continues.

If either of a corresponding pair of elements is the null value, null is returned for that element.

If you use the subroutine syntax, the resulting dynamic array is returned as *return.array*.

MAT statement

Use the MAT statement to assign one value to all of the elements in the array or to assign all the values of one array to the values of another array.

Use the first syntax to assign the same value to all array elements. Use any valid expression. The value of *expression* becomes the value of each array element.

Use the second syntax to assign values from the elements of *array2* to the elements of *array1*. Both arrays must previously be named and dimensioned. The dimensioning of the two arrays can be different. The values of the elements of the new array are assigned in consecutive order, regardless of whether the dimensions of the arrays are the same or not. If *array2* has more elements than in *array1*, the extra elements are ignored. If *array2* has fewer elements, the extra elements of *array1* are not assigned.

Note: Do not use the MAT statement to assign individual elements of an array.

Syntax

```
MAT array = expression
MAT array1 = MAT array2
```

Examples

Source lines	Program output
DIM ARRAY(5) QTY=10 MAT ARRAY=QTY FOR X=1 TO 5 PRINT "ARRAY(":X:")=",ARRAY(X) NEXT X	ARRAY(1)= 10 ARRAY(2)= 10 ARRAY(3)= 10 ARRAY(4)= 10 ARRAY(5)= 10
DIM ONE(4,1) MAT ONE=1 DIM TWO(2,2) MAT TWO = MAT ONE FOR Y=1 TO 4 PRINT "ONE(":Y:",1)=",ONE(Y,1) NEXT Y	ONE(1,1)= 1 ONE(2,1)= 1 ONE(3,1)= 1 ONE(4,1)= 1

Source lines	Program output
DIM ONE(4,1)	TWO(1,1)= 1
MAT ONE=1	TWO(1,2)= 1
DIM TWO(2,2)	TWO(2,1)= 1
MAT TWO = MAT ONE	TWO(2,2)= 1
FOR X=1 TO 2	
FOR Y=1 TO 2	
PRINT "TWO(":X:",":Y:")=",TWO(X,Y)	
NEXT Y	
NEXT X	

The following example sets all elements in ARRAY to the empty string:

```
MAT ARRAY=''
```

MATBUILD statement

Use the MATBUILD statement to build a dynamic array from a dimensioned array.

Syntax

MATBUILD *dynamic.array* FROM *array* [,*start* [,*end*]] [USING *delimiter*]

dynamic.array is created by concatenating the elements of *array* beginning with *start* and finishing with *end*. If *start* and *end* are not specified or are out of range, they default to 1 and the size of the array respectively.

array must be named and dimensioned in a MATBUILD statement or COMMON statement before it is used in this statement.

delimiter specifies characters to be inserted between fields of the dynamic array. If *delimiter* is not specified, it defaults to a field mark. To specify no delimiter, specify USING without *delimiter*.

If an element of *array* is the null value, the dynamic array will contain CHAR(128) for that element. If *start*, *end*, or *delimiter* is the null value, the MATBUILD statement fails and the program terminates with a run-time error.

Overflow elements

PICK, IN2, and REALITY flavor dimensioned arrays contain overflow elements in the last element. INFORMATION and IDEAL flavor dimensioned arrays contain overflow elements in element 0.

In PICK, IN2, and REALITY flavor accounts, if *end* is not specified, *dynamic.array* contains the overflow elements of *array*. In IDEAL and INFORMATION flavor accounts, to get the overflow elements you must specify *end* as less than or equal to 0, or as greater than the size of *array*.

REALITY flavor accounts use only the first character of *delimiter*, and if USING is specified without a delimiter, *delimiter* defaults to a field mark rather than an empty string.

MATCH operator

Use the MATCH operator or its synonym MATCHES to compare a string expression with a pattern.

Syntax

string **MATCH[ES]** *pattern*

pattern is a general description of the format of *string*. It can consist of text or the special characters X, A, and N preceded by an integer used as a repeating factor. For example, *n*N is the pattern for strings of *n* numeric characters.

The following table lists the *pattern* codes and their definitions:

Pattern	Definition
…	Any number of any characters (including none).
0X	Any number of any characters (including none).
*n*X	*n* number of any characters.
0A	Any number of alphabetic characters (including none).
nA	*n* number of alphabetic characters.
0N	Any number of numeric characters (including none).
*n*N	*n* number of numeric characters.
'*text*'	Exact text; any literal string (quotation marks required).
"*text*"	Exact text; any literal string (quotation marks required).

If *n* is longer than nine digits, it is used as text in a pattern rather than as a repeating factor for a special character. For example, the pattern "1234567890N" is treated as a literal string, not as a pattern of 1,234,567,890 numeric characters.

If the string being evaluated matches the pattern, the expression evaluates as true (1); otherwise, it evaluates as false (0). If either *string* or *pattern* is the null value, the match evaluates as false.

A tilde (~) placed immediately before *pattern* specifies a negative match. That is, it specifies a pattern or a part of a pattern that does not match the expression or a part of the expression. The match is true only if *string* and *pattern* are of equal length and differ in at least one character. An example of a negative match pattern is:

'A'~'X'5N

This pattern returns a value of true if the expression begins with the letter A, which is not followed by the letter X, and which is followed by any five numeric characters. Thus AB55555 matches the pattern, but AX55555, A55555, AX5555, and A5555 do not.

You can specify multiple patterns by separating them with value marks (ASCII CHAR(253)). The following expression is true if the address is either 16 alphabetic characters or 4 numeric characters followed by 12 alphabetic characters; otherwise, it is false:

ADDRESS MATCHES "16A": CHAR(253): "4N12A"

An empty string matches the following patterns: "0A", "0X", "0N", "…", "", '', or \\.

If NLS is enabled, the MATCH operator uses the current values for alphabetic and numeric characters specified in the NLS.LC.CTYPE file. For more information about the NLS.LC.CTYPE file, see the *UniVerse NLS Guide*.

MATCHFIELD function

Use the MATCHFIELD function to check a string against a match pattern.

See the MATCH operator, on page 253 for information about pattern matching.

field is an expression that evaluates to the portion of the match string to be returned.

If *string* matches *pattern*, the MATCHFIELD function returns the portion of *string* that matches the specified field in *pattern*. If *string* does not match *pattern*, or if *string* or *pattern* evaluates to the null value, the MATCHFIELD function returns an empty string. If field evaluates to the null value, the MATCHFIELD function fails and the program terminates with a run-time error.

pattern must contain specifiers to cover all characters contained in *string*. For example, the following statement returns an empty string because not all parts of *string* are specified in the pattern:

MATCHFIELD ("XYZ123AB", "3X3N", 1)

To achieve a positive pattern match on *string* above, the following statement might be used:

MATCHFIELD ("XYZ123AB", "3X3N0X", 1)

This statement returns a value of "XYZ".

Syntax

MATCHFIELD (*string*, *pattern*, *field*)

Examples

Source lines	Program output
Q=MATCHFIELD("AA123BBB9","2A0N3A0N",3) PRINT "Q= ",Q	Q= BBB
ADDR='20 GREEN ST. NATICK, MA.,01234' ZIP=MATCHFIELD(ADDR,"0N0X5N",3) PRINT "ZIP= ",ZIP	ZIP= 01234
INV='PART12345 BLUE AU' COL=MATCHFIELD(INV,"10X4A3X",2) PRINT "COL= ",COL	COL= BLUE

In the following example the string does not match the pattern:

Source lines	Program output
XYZ=MATCHFIELD('ABCDE1234',"2N3A4N",1) PRINT "XYZ= ",XYZ	XYZ=

In the following example the entire string does not match the pattern:

Source lines	Program output
XYZ=MATCHFIELD('ABCDE1234',"2N3A4N",1) PRINT "XYZ= ",XYZ	XYZ=

MATPARSE statement

Use the MATPARSE statement to separate the fields of *dynamic.array* into consecutive elements of *array*.

Syntax

```
MATPARSE array FROM dynamic.array [,delimiter]

 MATPARSE array [,start [,end]] FROM dynamic.array [USING delimiter]
           [SETTING elements]
```

array must be named and dimensioned in a MATPARSE statement or COMMON statement before it is used in this statement.

start specifies the starting position in *array*. If *start* is less than 1, it defaults to 1.

end specifies the ending position in *array*. If *end* is less than 1 or greater than the length of *array*, it defaults to the length of *array*.

delimiter is an expression evaluating to the characters used to delimit elements in *dynamic.array*. Use a comma or USING to separate delimiter from *dynamic.array*. *delimiter* can have no characters (an empty delimiter), one character, or more than one character with the following effects:

- An empty delimiter (a pair of quotation marks) parses *dynamic.array* so that each character becomes one element of *array* (see the second example). The default delimiter is a field mark. This is different from the empty delimiter. To use the default delimiter, omit the comma or USING following *dynamic.array*.
- A single character delimiter parses *dynamic.array* into fields delimited by that character by storing the substrings that are between successive delimiters as elements in the array. The delimiters are not stored in the array (see the first example).
- A multicharacter delimiter parses *dynamic.array* by storing as elements both the substrings that are between any two successive delimiters and the substrings consisting of one or more consecutive delimiters in the following way: *dynamic.array* is searched until any of the delimiter characters are found. All of the characters up to but not including the delimiter character are stored as an element of *array*. The delimiter character and any identical consecutive delimiter characters are stored as the next element. The search then continues as at the start of *dynamic.array* (see the third example).
- If *delimiter* is a system delimiter and a single CHAR(128) is extracted from *dynamic.array*, the corresponding element in *array* is set to the null value.

The characters in a multicharacter delimiter expression can be different or the same. A delimiter expression of /: might be used to separate hours, minutes, seconds and month, day, year in the formats 12:32:16 and 1/23/85. A delimiter expression of two spaces " " might be used to separate tokens on a command line that contain multiple blanks between tokens.

The SETTING clause sets the variable *elements* to the number of elements in *array*. If *array* overflows, *elements* is set to 0. The value of *elements* is the same as the value returned by the INMAT function after a MATPARSE statement.

If all the elements of *array* are filled before MATPARSE reaches the end of *dynamic.array*, MATPARSE puts the unprocessed part of *dynamic.array* in the zero element of *array* for IDEAL, INFORMATION, or PIOPEN flavor accounts, or in the last element of *array* for PICK, IN2, or REALITY flavor accounts.

Use the `INMAT` function after a MATPARSE statement to determine the number of elements loaded into the array. If there are more delimited fields in *dynamic.array* than elements in *array*, INMAT returns 0; otherwise, it returns the number of elements loaded.

If *start* is greater than *end* or greater than the length of *array*, no action is taken, and `INMAT` returns 0.

If *start*, *end*, *dynamic.array*, or *delimiter* evaluates to the null value, the MATPARSE statement fails and the program terminates with a run-time error message.

Examples

Source lines	Program output
DIM X(4) Y='1#22#3#44#5#66#7' MATPARSE X FROM Y, '#' FOR Z=0 TO 4 PRINT "X(":Z:")",X(Z) NEXT Z PRINT	X(0) 5#66#7 X(1) 1 X(2) 22 X(3) 3 X(4) 44
DIM Q(6) MATPARSE Q FROM 'ABCDEF', '' FOR P=1 TO 6 PRINT "Q(":P:")",Q(P) NEXT P PRINT	Q(1) A Q(2) B Q(3) C Q(4) D Q(5) E Q(6) F
DIM A(8,2) MATPARSE A FROM 'ABCDEFGDDDHIJCK', 'CD' FOR I = 1 TO 8 FOR J = 1 TO 2 PRINT "A(":I:",":J:")=",A(I,J)," ": NEXT J PRINT NEXT I END	A(1,1)= AB A(1,2)= C A(2,1)= A(2,2)= D A(3,1)= EFG A(3,2)= DDD A(4,1)= HIJ A(4,2)= C A(5,1)= K A(5,2)= A(6,1)= A(6,2)= A(7,1)= A(7,2)= A(8,1)= A(8,2)=

MATREAD statements

Use the MATREAD statement to assign the contents of the fields of a record from a UniVerse file to consecutive elements of *array*. The first field of the record becomes the first element of *array*, the second field of the record becomes the second element of *array*, and so on. The array must be named and dimensioned in a DIMENSION statement or COMMON statement before it is used in this statement.

Syntax

```
MATREAD array FROM [file.variable,] record.ID [ON ERROR statements]
```

```
        {THEN statements [ELSE statements] | ELSE statements}

{ MATREADL | MATREADU } array FROM [file.variable,] record.ID
        [ON ERROR statements] [LOCKED statements]
        {THEN statements [ELSE statements] | ELSE statements}
```

file.variable specifies an open file. If *file.variable* is not specified, the default file is assumed (for more information about default files, see the OPEN statement). If the file is neither accessible nor open, the program terminates with a run-time error message.

If *record.ID* exists, *array* is set to the contents of the record, and the THEN statements are executed; any ELSE statements are ignored. If no THEN statements are specified, program execution continues with the next sequential statement. If *record.ID* does not exist, the elements of *array* are not changed, and the ELSE statements are executed; any THEN statements are ignored.

If either *file.variable* or *record.ID* evaluates to the null value, the MATREAD statement fails and the program terminates with a run-time error. If any field in the record is the null value, null becomes an element in *array*. If a value or a subvalue in a multivalued field is the null value, it is read into the field as the stored representation of null (CHAR(128)).

If the file is an SQL table, the effective user of the program must have SQL SELECT privilege to read records in the file. For information about the effective user of a program, see the AUTHORIZATION statement, on page 71.

A MATREAD statement does not set an update record lock on the specified record. That is, the record remains available for update to other users. To prevent other users from updating the record until it is released, use a MATREADL or MATREADU statement.

If the number of elements in *array* is greater than the number of fields in the record, the extra elements in *array* are assigned empty string values. If the number of fields in the record is greater than the number of elements in the array, the extra values are stored in the zero element of *array* for IDEAL or INFORMATION flavor accounts, or in the last element of *array* for PICK, IN2, or REALITY flavor accounts. The zero element of an array can be accessed with a 0 subscript as follows:

MATRIX (0)

or:

MATRIX (0, 0)

Use the `INMAT` function after a MATREAD statement to determine the number of elements of the array that were actually used. If the number of fields in the record is greater than the number of elements in the array, the value of the `INMAT` function is set to 0.

If NLS is enabled, MATREAD and other BASIC statements that perform I/O operations always map external data to the UniVerse internal character set using the appropriate map for the input file. For details, see the READ statements, on page 303.

The ON ERROR clause

The ON ERROR clause is optional in MATREAD statements. Its syntax is the same as that of the ELSE clause. The ON ERROR clause lets you specify an alternative for program termination when a fatal error is encountered during processing of the MATREAD statement.

If a fatal error occurs, and the ON ERROR clause was not specified, or was ignored (as in the case of an active transaction), the following occurs:

- An error message appears.
- Any uncommitted transactions begun within the current execution environment roll back.

- The current program terminates.
- Processing continues with the next statement of the previous execution environment, or the program returns to the UniVerse prompt.

A fatal error can occur if any of the following occur:

- A file is not open.
- *file.variable* is the null value.
- A distributed file contains a part file that cannot be accessed.

If the ON ERROR clause is used, the value returned by the STATUS function is the error number.

The LOCKED clause

The LOCKED clause is optional, but recommended. Its syntax is the same as that of the ELSE clause.

The LOCKED clause handles a condition caused by a conflicting lock (set by another user) that prevents the MATREAD statement from processing. The LOCKED clause is executed if one of the following conflicting locks exists:

In this statement...	This requested lock...	Conflicts with these locks...
MATREADL	Shared record lock	Exclusive file lock Update record lock
MATREADU	Update record lock	Exclusive file lock Intent file lock Shared file lock Update record lock Shared record lock

If a MATREAD statement does not include a LOCKED clause, and a conflicting lock exists, the program will timeout after 60 minutes or until the lock is released.

If a LOCKED clause is used, the value returned by the STATUS function is the terminal number of the user who owns the conflicting lock.

Releasing Locks

A shared record lock can be released with a CLOSE statement, RELEASE statement, or STOP statement. An update record lock can be released with a CLOSE statement, DELETE statements, MATWRITE statements, RELEASE statement, STOP statement, WRITE statements, or WRITEV statement.

Locks acquired or promoted within a transaction are not released when the previous statements are processed.

MATREADL and MATREADU statements

Use the MATREADL syntax to acquire a shared record lock and then perform a MATREAD. This lets other programs read the record with no lock or a shared record lock.

Use the MATREADU syntax to acquire an update record lock and then perform a MATREAD. The update record lock prevents other users from updating the record until the user who set it releases it.

An update record lock can be acquired when no shared record lock exists, or promoted from a shared record lock owned by you if no other shared record locks exist.

Example

```
DIM ARRAY(10)
 OPEN 'SUN.MEMBER' TO SUN.MEMBER ELSE STOP
 MATREAD ARRAY FROM SUN.MEMBER, 6100 ELSE STOP
 *
 FOR X=1 TO 10
PRINT "ARRAY(":X:")",ARRAY(X)
 NEXT X
 *
 PRINT
 *
 DIM TEST(4)
 OPEN '','SUN.SPORT'    ELSE STOP 'CANNOT OPEN SUN.SPORT'
 MATREAD TEST FROM 851000 ELSE STOP
 *
 FOR X=0 TO 4
 PRINT "TEST(":X:")",TEST(X)
 NEXT X
```

This is the program output:

```
ARRAY(1) MASTERS
 ARRAY(2) BOB
 ARRAY(3) 55 WESTWOOD ROAD
 ARRAY(4) URBANA
 ARRAY(5) IL
 ARRAY(6) 45699
 ARRAY(7) 1980
 ARRAY(8) SAILING
 ARRAY(9)
 ARRAY(10)              II
 TEST(0) 6258
 TEST(1) 6100
 TEST(2) HARTWELL
 TEST(3) SURFING
 TEST(4) 4
```

MATREADL statement

Use the MATREADL statement to set a shared record lock and perform the MATREAD statement.

For details, see the MATREAD statements, on page 256.

MATREADU statement

Use the MATREADU statement to set an update record lock and perform the MATREAD statement.

For details, see the MATREAD statements, on page 256.

MATWRITE statements

Use the MATWRITE statement to write data from the elements of a dimensioned array to a record in a UniVerse file. The elements of *array* replace any data stored in the record. MATWRITE strips any trailing empty fields from the record.

Syntax

```
MATWRITE[U] array ON | TO [file.variable,] record.ID
      [ON ERROR statements] [LOCKED statements]
      [THEN statements] [ELSE statements]
```

file.variable specifies an open file. If *file.variable* is not specified, the default file is assumed (for more information on default files, see the OPEN statement). If the file is neither accessible nor open, the program terminates with a run-time message, unless ELSE statements are specified.

If the file is an SQL table, the effective user of the program must have SQL INSERT and UPDATE privileges to read records in the file. For information about the effective user of a program, see the AUTHORIZATION statement.

If the OPENCHK configurable parameter is set to TRUE, or if the file is opened with the OPENCHECK statement, all SQL integrity constraints are checked for every MATWRITE to an SQL table. If an integrity check fails, the MATWRITE statement uses the ELSE clause. Use the ICHECK function to determine what specific integrity constraint caused the failure.

The system searches the file for the record specified by *record.ID*. If the record is not found, MATWRITE creates a new record.

If NLS is enabled, MATWRITE and other BASIC statements that perform I/O operations always map internal data to the external character set using the appropriate map for the output file. For details, see the WRITE statements, on page 449. For more information about maps, see the *UniVerse NLS Guide*.

The ON ERROR clause

The ON ERROR clause is optional in the MATWRITE statement. The ON ERROR clause lets you specify an alternative for program termination when a fatal error is encountered while the MATWRITE is being processed.

If a fatal error occurs, and the ON ERROR clause was not specified, or was ignored (as in the case of an active transaction), the following occurs:

- An error message appears.
- Any uncommitted transactions begun within the current execution environment roll back.
- The current program terminates.
- Processing continues with the next statement of the previous execution environment, or the program returns to the UniVerse prompt.

A fatal error can occur if any of the following occur:

- A file is not open.
- *file.variable* is the null value.
- A distributed file contains a part file that cannot be accessed.

If the ON ERROR clause is used, the value returned by the STATUS function is the error number.

The LOCKED clause

The LOCKED clause is optional, but recommended.

The LOCKED clause handles a condition caused by a conflicting lock (set by another user) that prevents the MATWRITE statement from processing. The LOCKED clause is executed if one of the following conflicting locks exists:

- Exclusive file lock

- Intent file lock
- Shared file lock
- Update record lock
- Shared record lock

If the MATWRITE statement does not include a LOCKED clause, and a conflicting lock exists, the program pauses until the lock is released.

When updating a file, MATWRITE releases the update record lock set with a MATREADU statement, on page 259. To maintain the update record lock set with the MATREADU statement, use MATWRITEU instead of MATWRITE.

The new values are written to the record, and the THEN clauses are executed. If no THEN statements are specified, execution continues with the statement following the MATWRITE statement.

If either *file.variable* or *record.ID* evaluates to the null value, the MATWRITE statement fails and the program terminates with a run-time error message. Null elements of *array* are written to *record.ID* as the stored representation of the null value, CHAR(128).

The MATWRITEU statement

Use the MATWRITEU statement to update a record without releasing the update record lock set by a previous MATREADU statement (see the MATREADU statement, on page 259). To release the update record lock set by a MATREADU statement and maintained by a MATWRITEU statement, you must use a RELEASE or MATWRITE statement. If you do not explicitly release the lock, the record remains locked until the program executes the STOP statement. When more than one program or user could modify the same record, use a MATREADU statement to lock the record before doing the MATWRITE or MATWRITEU.

IDEAL and INFORMATION flavors

In IDEAL and INFORMATION flavor accounts, if the zero element of the array has been assigned a value by a MATREAD or MATREADU statement, the zero element value is written to the record as the *n*+1 field, where *n* is the number of elements dimensioned in the array. If the zero element is assigned an empty string, only the assigned elements of the array are written to the record; trailing empty fields are ignored. The new record is written to the file (replacing any existing record) without regard for the size of the array.

It is generally good practice to use the MATWRITE statement with arrays that have been loaded with either a MATREAD or a MATREADU statement.

After executing a MATWRITE statement, you can use the STATUS function to determine the result of the operation as follows (see the STATUS function, on page 380 for more information):

Value	Description
0	The record was locked before the MATWRITE operation.
-2	The record was unlocked before the MATWRITE operation.
-3	The record failed an SQL integrity check.

Example

```
DIM ARRAY(5)
 OPEN 'EX.BASIC'    TO EX.BASIC ELSE STOP 'CANNOT OPEN'
 MATREADU ARRAY FROM EX.BASIC, 'ABS' ELSE STOP
 ARRAY(1)='Y = 100'
 MATWRITE ARRAY TO EX.BASIC, 'ABS'
 PRINT 'STATUS()= ',STATUS()
```

This is the program output:

```
STATUS()= 0
```

MATWRITEU statement

Use the MATWRITEU statement to maintain an update record lock and perform the MATWRITE statement.

For details, see the MATWRITE statements, on page 259.

MAXIMUM function

Use the MAXIMUM function to return the element with the highest numeric value in *dynamic.array*. Nonnumeric values, except the null value, are treated as 0. If *dynamic.array* evaluates to the null value, null is returned. Any element that is the null value is ignored, unless all elements of *dynamic.array* are null, in which case null is returned.

result is the variable that contains the largest element found in *dynamic.array*.

dynamic.array is the array to be tested.

Syntax

MAXIMUM (*dynamic.array*)

CALL **!MAXIMUM** (*result, dynamic.array*)

Examples

```
A=1:@VM:"ZERO":@SM:20:@FM:-25
 PRINT "MAX(A)=",MAXIMUM(A)
```

This is the program output:

```
MAX(A)=20
```

In the following example, the !MAXIMUM subroutine is used to obtain the maximum value contained in array A. The nonnumeric value, Z, is treated as 0.

```
A=1:@VM:25:@VM:'Z':@VM:7
 CALL !MAXIMUM (RESULT,A)
 PRINT RESULT
```

This is the program output:

```
0
```

MINIMUM function

Use the MINIMUM function to return the element with the lowest numeric value in *dynamic.array*. Nonnumeric values and empty strings, except the SQL null value, are treated as 0. If *dynamic.array* evaluates to the null value, null is returned. Any element that is the null value is ignored, unless all elements of *dynamic.array* are null, in which case null is returned.

result is the variable that contains the smallest element found in *dynamic.array*.

dynamic.array is the array to be tested.

Syntax

MINIMUM (*dynamic.array*)

CALL **!MINIMUM** (*result, dynamic.array*)

Examples

```
A=1:@VM:"ZERO":@SM:20:@FM:-25
 PRINT "MIN(A)=",MINIMUM(A)
```

This is the program output:

```
MIN(A)= -25
```

In the following example, the !MINIMUM subroutine is used to obtain the minimum value contained in array A. The nonnumeric value, Q, is treated as 0.

```
A=2:@VM:19:@VM:6:@VM:'Q'
 CALL !MINIMUM (RESULT,A)
 PRINT RESULT
```

This is the program output:

```
0
```

The next example shows the output of the MINIMUM function for an empty string and the SQL null value:

```
MYNULL=@NULL.STR
MYSTR=""
CRT "MINIMUM NULL:":MINIMUM(MYNULL)
CRT "MINIMUM EMPTY STR:":MINIMUM(MYSTR)
```

The output from this program is:

```
MINIMUM NULL:
MINIMUM EMPTY STR: 0
```

MOD function

Use the MOD function to calculate the value of the remainder after integer division is performed on the dividend expression by the divisor expression.

Syntax

MOD (*dividend, divisor*)

The MOD function calculates the remainder using the following formula:

MOD (X, Y) = X - (INT (X / Y) * Y)

dividend and *divisor* can evaluate to any numeric value, except that *divisor* cannot be 0. If *divisor* is 0, a division by 0 warning message is printed, and 0 is returned. If either *dividend* or *divisor* evaluates to the null value, null is returned.

The MOD function works like the REM function, on page 320.

Example

```
X=85; Y=3
 PRINT 'MOD (X,Y)= ',MOD (X,Y)
```

This is the program output:

```
MOD (X,Y)= 1
```

MODS function

Use the MODS function to create a dynamic array of the remainder after the integer division of corresponding elements of two dynamic arrays.

Syntax

MODS (*array1, array2*)

CALL –**MODS** (*return.array, array1, array2*)

CALL !**MODS** (*return.array, array1, array2*)

The MODS function calculates each element according to the following formula:

XY.element = X - (INT (X / Y) * Y)

X is an element of *array1* and Y is the corresponding element of *array2*. The resulting element is returned in the corresponding element of a new dynamic array. If an element of one dynamic array has no corresponding element in the other dynamic array, 0 is returned. If an element of *array2* is 0, 0 is returned. If either of a corresponding pair of elements is the null value, null is returned for that element.

If you use the subroutine syntax, the resulting dynamic array is returned as *return.array*.

Example

```
A=3:@VM:7
 B=2:@SM:7:@VM:4
 PRINT MODS(A,B)
```

This is the program output:

```
1S0V3
```

MQCLOSE function

Use the MQCLOSE() function to close access to a queue or other object. When you close the queue, the queue and all uncommitted messages on the queue are deleted.

Syntax

status=**MQCLOSE**(*hConn, hObj, options*)

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
hConn	A handle denoting the connection to the queue manager. [IN]
hObj	The handle to the WebSphere MQ queue or object being closed. Upon successful completion of MQCLOSE, *hObj* is set to MQHO_UNUSABLE_HOBJ. [IN/OUT]
options^253	One or more option codes (MQCO_*) that specify how the WebSphere MQ queue or object is to be closed. If required, multiple option codes can be supplied by adding them together. For a complete description of the option codes available to MQCLOSE, see the *WebSphere MQ Application Programming Reference* manual. [IN]

Return codes

The following table describes the meaning of each return code.

Return code	Description
0 – MQCC_OK	Function call completed successfully.
1 – MQCC_WARNING	The function call succeeded, but a warning was returned. You can call the `MQGETERROR` function to get further details about the warning.
2 – MQCC_FAILED	The function call failed. You can call the `MQGETERROR` function to get further details about the failure.

Usage notes

`MQGETERROR`() – If the return code status is MQCC_WARNING or MQCC_FAILED, you can call the `MQGETERROR` function to get detailed information about the warning or error.

Refer to the *WebSphere MQ Application Programming Reference* manual for additional information about this function.

Refer to the *WebSphere MQ Messages* manual for more information about the WebSphere MQ reason codes. You can also access the `MQI_ErrMsg` file in the $UVHOME directory to read them.

MQCONN function

The `MQCONN`() function connects an application to a WebSphere MQ queue manager.

Syntax

```
status=MQCONN(qManager, hConn)
```

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
qManager	The name of the queue manager to which you are connecting. [IN]

Parameter	Description
hConn	The handle denoting the connection to the Websphere MQ queue manager. Use this handle in subsequent calls to other `CallMQI` functions. [IN]

Return codes

The following table describes the meaning of each return code.

Return code	Description
0 – MQCC_OK	Function call completed successfully.
1 – MQCC_WARNING	The function call succeeded, but a warning was returned. You can call the `MQGETERROR` function to get further details about the warning.
2 – MQCC_FAILED	The function call failed. You can call the `MQGETERROR` function to get further details about the failure.

Usage notes

`MQGETERROR`() – If the return code status is MQCC_WARNING or MQCC_FAILED, you can call the `MQGETERROR` function to get detailed information about the warning or error.

Refer to the *WebSphere MQ Application Programming Reference* manual for additional information about this function.

Refer to the *WebSphere MQ Messages* manual for more information about the WebSphere MQ reason codes. You can also access the `MQI_ErrMsg` file in the $UVHOME directory to read them.

MQDISC function

The `MQDISC` function terminates connections to the queue manager that were create using the `MQCONN` function. The input for this function is the *hConn* connection handle returned by the `MQCONN` function.

Syntax

status=**MQDISC**(*hConn*)

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
hConn	The handle denoting the connection to the WebSphere MQ queue manager. Upon successful completion, the `MQDISC` function sets this to MQHC_UNUSABLE_CONNECTION. [IN/OUT]

Return codes

The following table describes the meaning of each return code.

Return code	Description
0 – MQCC_OK	Function call completed successfully.
1 – MQCC_WARNING	The function call succeeded, but a warning was returned. You can call the `MQGETERROR` function to get further details about the warning.

Return code	Description
2 – MQCC_FAILED	The function call failed. You can call the MQGETERROR function to get further details about the failure.

Usage notes

MQGETERROR() – If the return code status is MQCC_WARNING or MQCC_FAILED, you can call the MQGETERROR function to get detailed information about the warning or error.

Refer to the *WebSphere MQ Application Programming Reference* manual for additional information about this function.

Refer to the *WebSphere MQ Messages* manual for more information about the WebSphere MQ reason codes. You can also access the MQI_ErrMsg file in the $UVHOME directory to read them.

MULS function

Use the MULS function to create a dynamic array of the element-by-element multiplication of two dynamic arrays.

Each element of *array1* is multiplied by the corresponding element of *array2* with the result being returned in the corresponding element of a new dynamic array. If an element of one dynamic array has no corresponding element in the other dynamic array, 0 is returned. If either of a corresponding pair of elements is the null value, null is returned for that element.

If you use the subroutine syntax, the resulting dynamic array is returned as *return.array*.

Syntax

MULS (*array1, array2*)

CALL -**MULS** (*return.array, array1, array2*)

CALL !**MULS** (*return.array, array1, array2*)

Example

```
A=1:@VM:2:@VM:3:@SM:4
 B=4:@VM:5:@VM:6:@VM:9
 PRINT MULS(A,B)
```

This is the program output:

```
4V10V18S0V0
```

NAP statement

Use the NAP statement to suspend the execution of a BASIC program, pausing for a specified number of milliseconds.

milliseconds is an expression evaluating to the number of milliseconds for the pause. If *milliseconds* is not specified, a value of 1 is used. If *milliseconds* evaluates to the null value, the NAP statement is ignored.

Syntax

NAP [*milliseconds*]

NEG function

Use the NEG function to return the arithmetic inverse of the value of the argument.

number is an expression evaluating to a number.

Syntax

NEG (*number*)

Example

In the following example, A is assigned the value of 10, and B is assigned the value of NEG(A), which evaluates to –10:

```
A = 10
 B = NEG(A)
```

NEGS function

Use the NEGS function to return the negative values of all the elements in a dynamic array. If the value of an element is negative, the returned value is positive. If *dynamic.array* evaluates to the null value, null is returned. If any element is null, null is returned for that element.

If you use the subroutine syntax, the resulting dynamic array is returned as *return.array*.

Syntax

NEGS (*dynamic.array*)

CALL **-NEGS** (*return.array, dynamic.array*)

NES function

Use the NES function to test if elements of one dynamic array are equal to the elements of another dynamic array.

Each element of *array1* is compared with the corresponding element of *array2*. If the two elements are equal, a 0 is returned in the corresponding element of a new dynamic array. If the two elements are not equal, a 1 is returned. If an element of one dynamic array has no corresponding element in the other dynamic array, a 1 is returned. If either of a corresponding pair of elements is the null value, null is returned for that element.

If you use the subroutine syntax, the resulting dynamic array is returned as *return.array*.

Syntax

NES (*array1, array2*)

CALL **-NES** (*return.array, array1, array2*)

```
CALL !NES (return.array, array1, array2)
```

NEXT statement

Use the NEXT statement to end a FOR…NEXT loop, causing the program to branch back to the FOR statement and execute the statements that follow it.

Each FOR statement must have exactly one corresponding NEXT statement.

variable is the name of the variable given as the index counter in the FOR statement. If the variable is not named, the most recently named index counter variable is assumed.

Syntax

```
NEXT [variable]
```

Example

```
FOR I=1 TO 10
    PRINT I:" ":
 NEXT I
 PRINT
```

This is the program output:

```
1 2 3 4 5 6 7 8 9 10
```

NOBUF statement

Use the NOBUF statement to turn off buffering for a file previously opened for sequential processing. Normally UniVerse uses buffering for sequential input and output operations. The NOBUF statement turns off this buffering and causes all writes to the file to be performed immediately. It eliminates the need for FLUSH operations but also eliminates the benefits of buffering. The NOBUF statement must be executed after a successful OPENSEQ statement or CREATE statement and before any input or output operation is performed on the record.

If the NOBUF operation is successful, the THEN statements are executed; the ELSE statements are ignored. If THEN statements are not present, program execution continues with the next statement.

If the specified file cannot be accessed or does not exist, the ELSE statements are executed; the THEN statements are ignored. If *file.variable* evaluates to the null value, the NOBUF statement fails and the program terminates with a run-time error message.

Syntax

```
NOBUF file.variable {THEN statements [ELSE statements] | ELSE
statements}
```

Example

In the following example, if RECORD1 in FILE.E can be opened, buffering is turned off:

```
OPENSEQ 'FILE.E', 'RECORD1' TO DATA THEN NOBUF DATA
 ELSE ABORT
```

NOT function

Use the NOT function to return the logical complement of the value of *expression*. If the value of *expression* is true, the NOT function returns a value of false (0). If the value of *expression* is false, the NOT function returns a value of true (1).

A numeric expression that evaluates to 0 is a logical value of false. A numeric expression that evaluates to anything else, other than the null value, is a logical true.

An empty string is logically false. All other string expressions, including strings that include an empty string, spaces, or the number 0 and spaces, are logically true.

If *expression* evaluates to the null value, null is returned.

Syntax

NOT (*expression*)

Example

```
X=5; Y=5
 PRINT NOT(X-Y)
 PRINT NOT(X+Y)
```

This is the program output:

```
1
 0
```

NOTS function

Use the NOTS function to return a dynamic array of the logical complements of each element of *dynamic.array*. If the value of the element is true, the NOTS function returns a value of false (0) in the corresponding element of the returned array. If the value of the element is false, the NOTS function returns a value of true (1) in the corresponding element of the returned array.

A numeric expression that evaluates to 0 has a logical value of false. A numeric expression that evaluates to anything else, other than the null value, is a logical true.

An empty string is logically false. All other string expressions, including strings which consist of an empty string, spaces, or the number 0 and spaces, are logically true.

If any element in *dynamic.array* is the null value, null is returned for that element.

If you use the subroutine syntax, the resulting dynamic array is returned as *return.array*.

Syntax

NOTS (*dynamic.array*)

CALL -**NOTS** (*return.array, dynamic.array*)

CALL !**NOTS** (*return.array, dynamic.array*)

Example

```
X=5; Y=5
```

```
PRINT NOTS(X-Y:@VM:X+Y)
```

This is the program output:

```
1V0
```

NULL statement

Use the NULL statement when a statement is required but no operation is to be performed. For example, you can use it with the ELSE clause if you do not want any operation performed when the ELSE clause is executed.

Note: This statement has nothing to do with the null value.

Syntax

NULL

Example

```
OPEN '','SUN.MEMBER' TO FILE ELSE STOP
 FOR ID=5000 TO 6000
    READ MEMBER FROM FILE, ID THEN PRINT ID ELSE NULL
 NEXT ID
```

NUM function

Use the NUM function to determine whether *expression* is a numeric or nonnumeric string. If *expression* is a number, a numeric string, or an empty string, it evaluates to true and a value of 1 is returned. If *expression* is a nonnumeric string, it evaluates to false and a value of 0 is returned.

A string that contains a period used as a decimal point (.) evaluates to numeric. A string that contains any other character used in formatting numeric or monetary amounts, for example, a comma (,) or a dollar sign ($) evaluates to nonnumeric.

If *expression* evaluates to the null value, null is returned.

If NLS is enabled, NUM uses the Numeric category of the current locale to determine the decimal separator. For more information about locales, see the *UniVerse NLS Guide*.

Syntax

NUM (*expression*)

Example

```
X=NUM(2400)
 Y=NUM("Section 4")
 PRINT "X= ",X,"Y= ",Y
```

This is the program output:

```
X= Y= 0
```

NUMS function

Use the NUMS function to determine whether the elements of a dynamic array are numeric or nonnumeric strings. If an element is numeric, a numeric string, or an empty string, it evaluates to true, and a value of 1 is returned to the corresponding element in a new dynamic array. If the element is a nonnumeric string, it evaluates to false, and a value of 0 is returned.

The NUMS of a numeric element with a decimal point (.) evaluates to true; the NUMS of a numeric element with a comma (,) or dollar sign ($) evaluates to false.

If *dynamic.array* evaluates to the null value, null is returned. If an element of *dynamic.array* is null, null is returned for that element.

If you use the subroutine syntax, the resulting dynamic array is returned as *return.array*.

If NLS is enabled, NUMS uses the Numeric category of the current locale to determine the decimal separator. For more information about locales, see the *UniVerse NLS Guide*.

Syntax

NUMS (*dynamic.array*)

CALL -**NUMS** (*return.array, dynamic.array*)

CALL !**NUMS** (*return.array, dynamic.array*)

OCONV function

Use the OCONV function to convert *string* to a specified format for external output. The result is always a string expression.

Syntax

OCONV (*string, conversion*)

string is converted to the external output format specified by *conversion*.

conversion must evaluate to one or more conversion codes separated by value marks (ASCII 253).

If multiple codes are used, they are applied from left to right as follows: the leftmost conversion code is applied to *string*, the next conversion code to the right is then applied to the result of the first conversion, and so on.

If *string* evaluates to the null value, null is returned. If *conversion* evaluates to the null value, the OCONV function fails and the program terminates with a run-time error message.

The OCONV function also allows PICK flavor exit codes.

The STATUS function reflects the result of the conversion:

Value	Description
0	The conversion is successful.
1	An invalid string is passed to the OCONV function; the original string is returned as the value of the conversion. If the invalid string is the null value, null is returned.
2	The conversion code is invalid.
3	Successful conversion of possibly invalid data.

For information about converting strings to an internal format, see the ICONV function, on page 205.

Examples

The following examples show date conversions:

Source line	Converted value
DATE=OCONV('9166',"D2")	3 Feb 93
DATE=OCONV(9166,'D/E')	3/2/1993
DATE=OCONV(9166,'DI') (For IN2, PICK, and REALITY flavor accounts.)	3/2/1993
DATE=OCONV('9166',"D2-")	2-3-93
DATE=OCONV(0,'D')	31 Dec 1967

The following examples show time conversions:

Source line	Converted value
TIME=OCONV(10000,"MT")	02:46
TIME=OCONV("10000","MTHS")	02:46:40am
TIME=OCONV(10000,"MTH")	02:46am
TIME=OCONV(10000,"MT.")	02.46
TIME=OCONV(10000,"MTS")	02:46:40

The following examples show hex, octal, and binary conversions:

Source line	Converted value
HEX=OCONV(1024,"MX")	400
HEX=OCONV('CDE',"MX0C")	434445
OCT=OCONV(1024,"MO")	2000
OCT=OCONV('CDE',"MO0C")	103104105
BIN=OCONV(1024,"MB")	10000000000
BIN=OCONV('CDE',"MB0C")	010000110100010001000101

The following examples show masked decimal conversions:

Source line	Converted value
X=OCONV(987654,"MD2")	9876.54
X=OCONV(987654,"MD0")	987654
X=OCONV(987654,"MD2,$")	$9,876.54
X=OCONV(987654,"MD24$")	$98.77
X=OCONV(987654,"MD2-Z")	9876.54
X=OCONV(987654,"MD2,D")	9,876.54
X=OCONV(987654,"MD3,$CPZ")	$987.654
X=OCONV(987654,"MD2,ZP12#")	####9,876.54

OCONVS function

Use the OCONVS function to convert the elements of *dynamic.array* to a specified format for external output.

Syntax

```
OCONVS (dynamic.array, conversion)
CALL -OCONVS (return.array, dynamic.array, conversion)
CALL !OCONVS (return.array, dynamic.array, conversion)
```

The elements are converted to the external output format specified by *conversion* and returned in a dynamic array. *conversion* must evaluate to one or more conversion codes separated by value marks (ASCII 253).

If multiple codes are used, they are applied from left to right as follows: the leftmost conversion code is applied to the element, the next conversion code to the right is then applied to the result of the first conversion, and so on.

If *dynamic.array* evaluates to the null value, null is returned. If any element of *dynamic.array* is null, null is returned for that element. If *conversion* evaluates to the null value, the OCONVS function fails and the program terminates with a run-time error message.

If you use the subroutine syntax, the resulting dynamic array is returned as *return.array*.

The STATUS function reflects the result of the conversion:

Return value	Description
0	The conversion is successful.
1	An invalid element is passed to the OCONVS function; the original element is returned. If the invalid element is the null value, null is returned for that element.
2	The conversion code is invalid.

For information about converting elements in a dynamic array to an internal format, see the ICONVS function, on page 206.

ON statement

Use the ON statement to transfer program control to one of the internal subroutines named in the GOSUB clause or to one of the statements named in the GOTO clause.

Syntax

```
ON expression GOSUB statement.label [:] [,statement.label [:]… ]
ON expression GO[TO] statement.label [:] [,statement.label [:] … ]
```

Using the GOSUB clause

Use ON GOSUB to transfer program control to one of the internal subroutines named in the GOSUB clause. The value of *expression* in the ON clause determines which of the subroutines named in the GOSUB clause is to be executed.

During execution, *expression* is evaluated and rounded to an integer. If the value of *expression* is 1 or less than 1, the first subroutine named in the GOSUB clause is executed; if the value of *expression* is 2, the second subroutine is executed; and so on. If the value of *expression* is greater than the number of subroutines named in the GOSUB clause, the last subroutine is executed. If *expression* evaluates to the null value, the ON statement fails and the program terminates with a run-time error message.

statement.label can be any valid label defined in the program. If a nonexistent statement label is given, an error message is issued when the program is compiled. You must use commas to separate statement labels. You can use a colon with the statement labels to distinguish them from variable names.

A RETURN statement in the subroutine returns program flow to the statement following the ON GOSUB statement.

The ON GOSUB statement can be written on more than one line. A comma is required at the end of each line of the ON GOSUB statement except the last.

Using ON GOSUB in a PICK flavor account

If the value of *expression* is less than 1, the next statement is executed; if the value of *expression* is greater than the number of subroutines named in the GOSUB clause, execution continues with the next statement rather than the last subroutine. To get this characteristic in other flavors, use the ONGO.RANGE option of the $OPTIONS statement.

Using the GOTO clause

Use ON GOTO to transfer program control to one of the statements named in the GOTO clause. The value of *expression* in the ON clause determines which of the statements named in the GOTO clause is to be executed. During execution, *expression* is evaluated and rounded to an integer.

If the value of *expression* is 1 or less than 1, control is passed to the first statement label named in the GOTO clause; if the value of *expression* is 2, control is passed to the second statement label; and so on. If the value of *expression* is greater than the number of statements named in the GOTO clause, control is passed to the last statement label. If *expression* evaluates to the null value, the ON statement fails and the program terminates with a run-time error message.

statement.label can be any valid label defined in the program. If a nonexistent statement label is given, an error message is issued when the program is compiled. You must use commas to separate statement labels. You can use a colon with the statement labels to distinguish them from variable names.

Using ON GOTO in a PICK flavor account

If the value of *expression* is less than 1, control is passed to the next statement; if the value of *expression* is greater than the number of the statements named in the GOTO clause, execution continues with the next statement rather than the last statement label. To get this characteristic with other flavors, use the ONGO.RANGE option of the $OPTIONS statement.

Examples

Source lines	Program output
FOR X=1 TO 4 ON X GOSUB 10,20,30,40 PRINT 'RETURNED FROM SUBROUTINE' NEXT X STOP 10 PRINT 'AT LABEL 10' RETURN 20 PRINT 'AT LABEL 20' RETURN 30 PRINT 'AT LABEL 30' RETURN 40 PRINT 'AT LABEL 40' RETURN	AT LABEL 10 RETURNED FROM SUBROUTINE AT LABEL 20 RETURNED FROM SUBROUTINE AT LABEL 30 RETURNED FROM SUBROUTINE AT LABEL 40 RETURNED FROM SUBROUTINE
VAR=1234 Y=1 10* X=VAR[Y,1] IF X='' THEN STOP ON X GOTO 20,30,40 20* PRINT 'AT LABEL 20' Y=Y+1 GOTO 10 30* PRINT 'AT LABEL 30' Y=Y+1 GOTO 10 40* PRINT 'AT LABEL 40' Y=Y+1 GOTO 10	AT LABEL 20 AT LABEL 30 AT LABEL 40 AT LABEL 40

OPEN statement

Use the OPEN statement to open a UniVerse file for use by BASIC programs. All file references in a BASIC program must be preceded by either an OPEN statement or an OPENCHECK statement for that

file. You can open several UniVerse files at the same point in a program, but you must use a separate OPEN statement for each file.

Syntax

```
OPEN [dict,] filename [TO file.variable] [ON ERROR statements]
       {THEN statements [ELSE statements] | ELSE statements}
```

dict is an expression that evaluates to a string specifying whether to open the file dictionary or the data file. Use the string DICT to open the file dictionary, or use PDICT to open an associated Pick-style dictionary. Any other string opens the data file. By convention an empty string or the string DATA is used when you are opening the data file. If the *dict* expression is omitted, the data file is opened. If *dict* is the null value, the data file is opened.

filename is an expression that evaluates to the name of the file to be opened. If the file exists, the file is opened, and the THEN statements are executed; the ELSE statements are ignored. If no THEN statements are specified, program execution continues with the next statement. If the file cannot be accessed or does not exist, the ELSE statements are executed; any THEN statements are ignored. If *filename* evaluates to the null value, the OPEN statement fails and the program terminates with a run-time error message.

Use the TO clause to assign the opened file to *file.variable*. All statements that read, write to, delete, or clear that file must refer to it by the name of the file variable. If you do not assign the file to a file variable, an internal default file variable is used. File references that do not specify a file variable access the default file variable, which contains the most recently opened file. The file opened to the current default file variable is assigned to the system variable @STDFIL.

Default file variables are not local to the program from which they are executed. When a subroutine is called, the current default file variable is shared with the calling program.

When opening an SQL table, the OPEN statement enforces SQL security. The permissions granted to the program's effective user ID are loaded when the file is opened. If no permissions have been granted, the OPEN statement fails, and the ELSE statements are executed.

All writes to an SQL table opened with the OPEN statement are subject to SQL integrity checking unless the OPENCHK configurable parameter has been set to FALSE. Use the OPENCHECK statement instead of the OPEN statement to enable automatic integrity checking for all writes to a file, regardless of whether the OPENCHK configurable parameter is true or false.

Use the INMAT function after an OPEN statement to determine the modulo of the file.

The ON ERROR clause

The ON ERROR clause is optional in the OPEN statement. Its syntax is the same as that of the ELSE clause. The ON ERROR clause lets you specify an alternative for program termination when a fatal error is encountered while the OPEN statement is being processed.

If a fatal error occurs, and the ON ERROR clause was not specified, or was ignored (as in the case of an active transaction), the following occurs:

- An error message appears.
- Any uncommitted transactions begun within the current execution environment roll back.
- The current program terminates.
- Processing continues with the next statement of the previous execution environment, or the program returns to the UniVerse prompt.

A fatal error can occur if any of the following occur:

- A file is not open.

- *file.variable* is the null value.
- A distributed file contains a part file that cannot be accessed.

If the ON ERROR clause is used, the value returned by the `STATUS` function is the error number.

The STATUS function

The file type is returned if the file is opened successfully. If the file is not opened successfully, the following values may return:

Value	Description
-1	File name not found in the VOC file.
-2	A generic error that can occur for various reasons. Null file name or file. This error may also occur when you cannot open a file across UVNet.
-3	Operating system access error that occurs when you do not have permission to access a UniVerse file in a directory. For example, this may occur when trying to access a type 1 or type 30 file.
-4	Access error when you do not have operating system permissions or if DATA.30 is missing for a type 30 file.
-5	Read error detected by the operating system.
-6	Unable to lock file header.
-7	Invalid file revision or wrong byte-ordering for the platform.
-8	Invalid part file information.
-9	Invalid type 30 file information in a distributed file.
-10	A problem occurred while the file was being rolled forward during warmstart recovery. Therefore, the file is marked “inconsistent.”
-11	The file is a view, therefore it cannot be opened by a BASIC program.
-12	No SQL privileges to open the table.
-13	Index problem.
-14	Cannot open the NFS file.
-15	There is a problem with the OVER.30 file in a dynamic file.
-16	Modulo over limit.
-17	Freechain corruption.
-18	SICA corruption.
-19	External Database Access (EDA) setup error.
-20	Automatic Data Encryption (ADE) setup error.

Examples

```
OPEN "SUN.MEMBER" TO DATA ELSE STOP "CAN'T OPEN SUN.MEMBER"
 OPEN "FOOBAR" TO FOO ELSE STOP "CAN'T OPEN FOOBAR"
 PRINT "ALL FILES OPEN OK"
```

This is the program output:

```
CAN'T OPEN FOOBAR
```

The following example opens the same file as in the previous example. The OPEN statement includes an empty string for the *dict* argument.

```
OPEN "","SUN.MEMBER" TO DATA ELSE STOP "CAN'T OPEN SUN.MEMBER"
 OPEN "","FOO.BAR" TO FOO ELSE STOP "CAN'T OPEN FOOBAR"
 PRINT "ALL FILES OPEN OK"
```

OPENCHECK statement

Use the OPENCHECK statement to open an SQL table for use by BASIC programs, enforcing SQL integrity checking. All file references in a BASIC program must be preceded by either an OPENCHECK statement or an OPEN statement for that file.

The OPENCHECK statement works like the OPEN statement, except that SQL integrity checking is enabled if the file is an SQL table. All field integrity checks for an SQL table are stored in the security and integrity constraints area (SICA). The OPENCHECK statement loads the compiled form of these integrity checks into memory, associating them with the file variable. All writes to the file are subject to SQL integrity checking.

Syntax

```
OPENCHECK [dict,] filename [TO file.variable]
        {THEN statements [ELSE statements] | ELSE statements}
```

The STATUS function

The file type is returned if the file is opened successfully. If the file is not opened successfully, the following values may return:

Value	Description
-1	File name not found in the VOC file.
-2	Null file name or file. This error may also occur when you cannot open a file across UVNet.
-3	Operating system access error that occurs when you do not have permission to access a UniVerse file in a directory. For example, this may occur when trying to access a type 1 or type 30 file.
-4	Access error when you do not have operating system permissions or if DATA.30 is missing for a type 30 file.
-5	Read error detected by the operating system.
-6	Unable to lock file header.
-7	Invalid file revision or wrong byte-ordering for the platform.
-8	Invalid part file information.
-9	Invalid type 30 file information in a distributed file.
-10	A problem occurred while the file was being rolled forward during warmstart recovery. Therefore, the file is marked “inconsistent.”
-11	The file is a view, therefore it cannot be opened by a BASIC program.
-12	No SQL privileges to open the table.
-13	A generic error that can occur for various reasons. Index problem.

Value	Description
-14	Cannot open the NFS file.

OPENDEV statement

Use the OPENDEV statement to open a device for sequential processing. OPENDEV also sets a record lock on the opened device or file.

See the READSEQ statement, on page 310 and WRITESEQ statement, on page 454 for more details on sequential processing.

Syntax

```
OPENDEV device TO file.variable [LOCKED statements]
        {THEN statements [ELSE statements] | ELSE statements}
```

device is an expression that evaluates to the record ID of a device definition record in the &DEVICE& file. If *device* evaluates to the null value, the OPENDEV statement fails and the program terminates with a run-time error message. For more information, see the following section.

The TO clause assigns a *file.variable* to the device being opened. All statements used to read to or write from that device must refer to it by the assigned *file.variable*.

If the device exists and is not locked, the device is opened and any THEN statements are executed; the ELSE statements are ignored. If no THEN statements are specified, program execution continues with the next statement.

If the device is locked, the LOCKED statements are executed; THEN statements and ELSE statements are ignored.

If the device does not exist or cannot be opened, the ELSE statements are executed; any THEN statements are ignored. The device must have the proper access permissions for you to open it.

If NLS is enabled, you can use OPENDEV to open a device that uses a map defined in the &DEVICE& file. If there is no map defined in the &DEVICE& file, the default *mapname* is the name in the NLSDEFDEVMAP parameter in the `uvconfig` file. For more information about maps, see the *UniVerse NLS Guide*.

Devices on Windows platforms

On Windows NT systems, you may need to change to block size defined for a device in the &DEVICE& file before you can use OPENDEV to reference the device. On some devices there are limits to the type of sequential processing that is available once you open the device. The following table summarizes the limits:

Device type	Block size	Processing available
4 mm DAT drive	No change needed.	No limits.
8 mm DAT drive	No change needed.	No limits.
1/4-inch cartridge drive, 60 MB or 150 MB	Specify the block size as 512 bytes or a multiple of 512 bytes.	Use READBLK and WRITEBLK to read or write data in blocks of 512 bytes. Use SEEK only to move the file pointer to the beginning or the end of the file. You can use WEOF to write an end-of-file (EOF) mark only at the beginning of the data or after a write.

Device type	Block size	Processing available
1/4-inch 525 cartridge drive	No change needed.	No limits.
Diskette drive	Specify the block size as 512 bytes or a multiple of 512 bytes.	Use SEEK only to move the file pointer to the beginning of the file. Do not use WEOF.

The LOCKED clause

The LOCKED clause is optional, but recommended.

The LOCKED clause handles a condition caused by a conflicting lock (set by another user) that prevents the OPENDEV statement from processing. The LOCKED clause is executed if one of the following conflicting locks exists:

- Exclusive file lock
- Intent file lock
- Shared file lock
- Update record lock
- Shared record lock

If the OPENDEV statement does not include a LOCKED clause, and a conflicting lock exists, the program pauses until the lock is released.

Example

The following example opens TTY30 for sequential input and output operations:

```
OPENDEV 'TTY30' TO TERM THEN PRINT 'TTY30 OPENED'
 ELSE ABORT
```

This is the program output:

```
TTY30 OPENED
```

OPENPATH statement

The OPENPATH statement is similar to the OPEN statement, except that the *pathname* of the file is specified. This file is opened without reference to the VOC file. The file must be a hashed UniVerse file or a directory (UniVerse types 1 and 19).

Syntax

```
OPENPATH pathname [TO file.variable] [ON ERROR statements]
        {THEN statements [ELSE statements] | ELSE statements}
```

pathname specifies the relative or absolute path name of the file to be opened. If the file exists, it is opened and the THEN statements are executed; the ELSE statements are ignored. If *pathname* evaluates to the null value, the OPENPATH statement fails and the program terminates with a run-time error message.

If the file cannot be accessed or does not exist, the ELSE statements are executed; any THEN statements are ignored.

Use the TO clause to assign the file to a *file.variable*. All statements used to read, write, delete, or clear that file must refer to it by the assigned *file.variable* name. If you do not assign the file to a *file.variable*, an internal default file variable is used. File references that do not specify *file.variable* access the most recently opened default file. The file opened to the default file variable is assigned to the system variable @STDFIL.

The ON ERROR clause

The ON ERROR clause is optional in the OPENPATH statement. Its syntax is the same as that of the ELSE clause. The ON ERROR clause lets you specify an alternative for program termination when a fatal error is encountered during processing of the OPENPATH statement.

If a fatal error occurs, and the ON ERROR clause was not specified, or was ignored (as in the case of an active transaction), the following occurs:

- An error message appears.
- Any uncommitted transactions begun within the current execution environment roll back.
- The current program terminates.
- Processing continues with the next statement of the previous execution environment, or the program returns to the UniVerse prompt.

A fatal error can occur if any of the following occur:

- A file is not open.
- *file.variable* is the null value.
- A distributed file contains a part file that cannot be accessed.

If the ON ERROR clause is used, the value returned by the `STATUS` function is the error number.

The STATUS function

You can use the `STATUS` function after an OPENPATH statement to find the cause of a file open failure (that is, for an OPENPATH statement in which the ELSE clause is used). The following values can be returned if the OPENPATH statement is unsuccessful:

Value	Description
-1	File name not found in the VOC file.
-2	A generic error that can occur for various reasons. Null file name or file. This error may also occur when you cannot open a file across UVNet.
-3	Operating system access error that occurs when you do not have permission to access a UniVerse file in a directory. For example, this may occur when trying to access a type 1 or type 30 file.
-4	Access error when you do not have operating system permissions or if DATA.30 is missing for a type 30 file.
-5	Read error detected by the operating system.
-6	Unable to lock file header.
-7	Invalid file revision or wrong byte-ordering for the platform.
-8	Invalid part file information.
-9	Invalid type 30 file information in a distributed file.
-10	A problem occurred while the file was being rolled forward during warmstart recovery. Therefore, the file is marked “inconsistent.”
-11	The file is a view, therefore it cannot be opened by a BASIC program.

Value	Description
-12	No SQL privileges to open the table.
-13	Index problem.
-14	Cannot open the NFS file.

Example

The following example opens the file SUN.MEMBER. The path name specifies the file.

```
OPENPATH '/user/members/SUN.MEMBER' ELSE ABORT
```

OPENSEQ statement

Use the OPENSEQ statement to open a file for sequential processing. All sequential file references in a BASIC program must be preceded by an OPENSEQ or OPENDEV statement for that file. Although you can open several files for sequential processing at the same point in the program, you must issue a separate OPENSEQ statement for each.

See the READSEQ statement, on page 310 and WRITESEQ statement, on page 454 for more details on sequential processing.

Syntax

```
OPENSEQ filename, record.ID TO file.variable [USING dynamic.array]
        [ON ERROR statements] [LOCKED statements]
        {THEN statements [ELSE statements] | ELSE statements}

OPENSEQ pathname TO file.variable [USING dynamic.array]
        [ON ERROR statements] [LOCKED statements]
        {THEN statements [ELSE statements] | ELSE statements}
```

Note: Performing multiple OPENSEQ operations on the same file results in creating only one update record lock. This single lock can be released by a CLOSESEQ statement or RELEASE statement.

The first syntax is used to open a record in a type 1 or type 19 file.

The second syntax specifies a path name to open a UNIX or DOS file. The file can be a disk file, a pipe, or a special device.

filename specifies the name of the type 1 or type 19 file containing the record to be opened.

record.ID specifies the record in the file to be opened. If the record exists and is not locked, the file is opened and the THEN statements are executed; the ELSE statements are ignored. If no THEN statements are specified, program execution continues with the next statement. If the record or the file itself cannot be accessed or does not exist, the ELSE statements are executed; any THEN statements are ignored.

pathname is an explicit path name for the file, pipe, or device to be opened. If the file exists and is not locked, it is opened and the THEN statements are executed; the ELSE statements are ignored. If the path name does not exist, the ELSE statements are executed; any THEN statements are ignored.

If the file does not exist, the OPENSEQ statement fails. The file can also be explicitly created with the CREATE statement.

OPENSEQ sets an update record lock on the specified record or file. This lock is reset by a CLOSESEQ statement. This prevents any other program from changing the record while you are processing it.

If *filename*, *record.ID*, or *pathname* evaluate to the null value, the OPENSEQ statement fails and the program terminates with a run-time error message.

The TO clause is required. It assigns the record, file, or device to *file.variable*. All statements used to sequentially read, write, delete, or clear that file must refer to it by the assigned file variable name.

If NLS is enabled, you can use the OPENSEQ *filename, record.ID* statement to open a type 1 or type 19 file that uses a map defined in the `.uvnlsmap` file in the directory containing the type 1 or type 19 file. If there is no *.uvnlsmap* file in the directory, the default *mapname* is the name in the NLSDEFDIRMAP parameter in the `uvconfig` file.

Use the OPENSEQ *pathname* statement to open a UNIX pipe, file, or a file specified by a device that uses a map defined in the `.uvnlsmap` file in the directory holding *pathname*. If there is no `.uvnlsmap` file in the directory, the default *mapname* is the name in the NLSDEFSEQMAP parameter in the `uvconfig` file, or you can use the `SET.SEQ.MAP` command to assign a map.

For more information about maps, see the *UniVerse NLS Guide*.

File buffering

Normally UniVerse uses buffering for sequential input and output operations. Use the NOBUF statement after an OPENSEQ statement to turn off buffering and cause all writes to the file to be performed immediately. For more information about file buffering, see the NOBUF statement, on page 269.

The USING clause

You can optionally include the USING clause to control whether the opened file is included in the rotating file pool. The USING clause supplements OPENSEQ processing with a dynamic array whose structure emulates an &DEVICE& file record. Field 17 of the dynamic array controls inclusion in the rotating file pool with the following values:

- Y removes the opened file.
- N includes the opened file.The

ON ERROR clause

The ON ERROR clause is optional in the OPENSEQ statement. Its syntax is the same as that of the ELSE clause. The ON ERROR clause lets you specify an alternative for program termination when a fatal error is encountered while the OPENSEQ statement is being processed.

If a fatal error occurs, and the ON ERROR clause was not specified, or was ignored (as in the case of an active transaction), the following occurs:

- An error message appears.
- Any uncommitted transactions begun within the current execution environment roll back.
- The current program terminates.
- Processing continues with the next statement of the previous execution environment, or the program returns to the UniVerse prompt.

A fatal error can occur if any of the following occur:

- A file is not open.
- *file.variable* is the null value.
- A distributed file contains a part file that cannot be accessed.

If the ON ERROR clause is used, the value returned by the STATUS function is the error number.

The LOCKED clause is optional, but recommended. Its syntax is the same as that of the ELSE clause. The LOCKED clause handles a condition caused by a conflicting lock (set by another user) that prevents the OPENSEQ statement from processing. The LOCKED clause is executed if one of the following conflicting locks exists:

- Exclusive file lock
- Intent file lock
- Shared file lock
- Update record lock
- Shared record lock

If the OPENSEQ statement does not include a LOCKED clause, and a conflicting lock exists, the program pauses until the lock is released.

Use the `STATUS` function after an OPENSEQ statement to determine whether the file was successfully opened.

The STATUS function

The file type is returned if the file is opened successfully. If the file is not opened successfully, the following values may return:

Value	Description
-1	File name not found in the VOC file.
-2	A generic error that can occur for various reasons. Null file name or file. This error may also occur when you cannot open a file across UVNet.
-3	Operating system access error that occurs when you do not have privileges to access a UniVerse file in a directory. For example, this may occur when trying to access a type 1 or type 30 file.
-4	Access error when you do not have operating system permissions or if DATA.30 is missing for a type 30 file.
-5	Read error detected by the operating system.
-6	Unable to lock file header.
-7	Invalid file revision or wrong byte-ordering for the platform.
-8	Invalid part file information.
-9	Invalid type 30 file information in a distributed file.
-10	A problem occurred while the file was being rolled forward during warmstart recovery. Therefore, the file is marked “inconsistent.”
-11	The file is a view, therefore it cannot be opened by a BASIC program.
-12	No SQL privileges to open the table.
-13	Index problem.
-14	Cannot open the NFS file.

Examples

The following example reads RECORD1 from the nonhashed file FILE.E:

```
OPENSEQ 'FILE.E', 'RECORD1' TO FILE THEN
    PRINT "'FILE.E' OPENED FOR PROCESSING"
```

```
    END ELSE ABORT
  READSEQ A FROM FILE THEN PRINT A ELSE STOP
```

The next example writes the record read from FILE.E to the file */usr/depta/file1*:

```
 OPENSEQ '/usr/depta/file1' TO OUTPUT THEN
    PRINT "usr/depta/file1 OPENED FOR PROCESSING"
  END ELSE ABORT
  WRITESEQ A ON OUTPUT ELSE PRINT "CANNOT WRITE TO OUTPUT"
    .
    .
    .
  CLOSESEQ FILE
  CLOSESEQ OUTPUT
  END
```

This is the program output:

```
 FILE.E OPENED FOR PROCESSING
  HI THERE
    .
    .
    .
  /usr/depta/file1 OPENED FOR PROCESSING
```

The next example includes the USING clause to remove an opened file from the rotating file pool:

```
 DEVREC = "1"@FM
  FOR I = 2 TO 16
     DEVREC = DEVREC:I:@FM
  NEXT I
  DEVREC=DEVREC:'Y'
  *
  OPENSEQ 'SEQTEST', 'TESTDATA' TO TESTFILE USING DEVREC
  THEN PRINT "OPENED 'TESTDATA' OK...."
  ELSE PRINT "COULD NOT OPEN TESTDATA"
  CLOSESEQ TESTFILE
```

This is the program output:

```
OPENED 'TESTDATA' OK
```

openSecureSocket function

Use the `openSecureSocket`() function to open a secure socket connection in a specified mode and return the status.

This function behaves exactly the same as the `openSocket`() function, except that it returns the handle to a socket that transfers data in a secured mode (SSL/TLS).

All parameters (with the exception of *context*) have the exact meaning as the `openSocket`() parameters. *Context* must be a valid security context handle.

Once the socket is opened, any change in the associated security context will not affect the established connection.

Syntax

openSecureSocket(*name_or_IP, port, mode, timeout, socket_handle, context*)

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
name_or_IP	DNS name (x.com) or IP address of a server.
port	Port number. If the port number is specified as a value <= 0, CallHTTP defaults to a port number of 40001.
mode	0: using current mode 1: blocking mode (default) 2: non-blocking mode
timeout	The timeout value, expressed in milliseconds. If you specify mode as 0, timeout will be ignored.
socket_handle	A handle to the open socket.
context	A handle to the security context.

Return codes

The following table describes the status of each return code.

Return code	Description
0	Success.
99	UniVerse failed to obtain a license for an interactive PHANTOM process.
1-41	See Socket function error return codes, on page 584.
101	Invalid security context handle.
102	SSL/TLS handshake failure (unspecified, peer is not SSL aware).
103	Requires client authentication but does not have a certificate in the security context.
104	Unable to authenticate server.

openSocket function

Use the `openSocket()` function to open a socket connection in a specified mode and return the status.

Syntax

`openSocket``(`*`name_or_IP, port, mode, timeout, socket_handle`*`)`

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
name_or_IP	DNS name (x.com) or IP address of a server.
port	Port number. If the port number is specified as a value <= 0, CallHTTP defaults to a port number of 40001.

Parameter	Description
mode	0: using current mode 1: blocking mode (default) 2: non-blocking mode
timeout	The timeout value, expressed in milliseconds. If you specify mode as 0, timeout will be ignored.
socket_handle	A handle to the open socket.

Return codes

The following table describes the status of each return code.

Return code	Description
0	Success.
99	UniVerse failed to obtain a license for an interactive PHANTOM process.
Non-zero	See Socket function error return codes, on page 584.

Return status

The following table describes the return status of each mode.

Mode	Return status
Non-blocking	The function will return immediately regardless of whether or not the socket is successfully opened. The return code indicates if the operation is successful. The *timeout* value is ignored.
Blocking	If a positive *timeout* is specified, the function will either return with a valid socket handle or will time out after the specified *timeout* period. If the *timeout* value is 0, the function will block until either the socket is successfully opened, the underlying TCP/IP connection times out or some other error prevents the socket from opening.

OpenXMLData function

After you prepare the XML document, open it using the `OpenXMLData` function.

Syntax

```
Status=OpenXMLData(xml_handle,xml_data_extraction_rule,
xml_data_handle)
```

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
xml_handle	The XML handle generated by the `PrepareXML`() function.
xml_data_extraction_rule	The path to the XML extraction rule file.

Parameter	Description
xml_data_handle	The XML data file handle. The following are the possible return values: XML.SUCCESS: Success. XML.ERROR: Failed XML.INVALID.HANDLE: Invalid XML handle

Example

The following example illustrates use of the `OpenXMLData` function:

```
status = OpenXMLData("STUDENT_XML", "&XML&/MYSTUDENT.ext",STUDENT_XML_DATA)
If status = XML.ERROR THEN
STOP "Error when opening the XML document. "
END
IF status = XML.INVALID.HANDLE THEN
STOP "Error: Invalid parameter passed."
END
```

ORS function

Use the `ORS` function to create a dynamic array of the logical OR of corresponding elements of two dynamic arrays.

Each element of the new dynamic array is the logical OR of the corresponding elements of *array1* and *array2*. If an element of one dynamic array has no corresponding element in the other dynamic array, a false is assumed for the missing element.

If both corresponding elements of *array1* and *array2* are the null value, null is returned for those elements. If one element is the null value and the other is 0 or an empty string, null is returned. If one element is the null value and the other is any value other than 0 or an empty string, a true is returned.

If you use the subroutine syntax, the resulting dynamic array is returned as *return.array*.

Syntax

ORS (*array1, array2*)

CALL -**ORS** (*return.array, array1, array2*)

CALL !**ORS** (*return.array, array1, array2*)

Example

```
A="A":@SM:0:@VM:4:@SM:1
 B=0:@SM:1-1:@VM:2
 PRINT ORS(A,B)
```

This is the program output:

```
1S0V1S1
```

PAGE statement

Use the PAGE statement to print headings, footings, and page advances at the appropriate places on the specified output device.

You can specify headings and footings before execution of the PAGE statement (see the HEADING statement, on page 198 and FOOTING statement, on page 175). If there is no heading or footing, PAGE clears the screen.

Syntax

```
PAGE [ON print.channel] [page#]
```

The ON clause specifies the logical print channel to use for output. *print.channel* is an expression that evaluates to a number from –1 through 255. If you do not use the ON clause, logical print channel 0 is used, which prints to the user's terminal if PRINTER OFF is set (see the PRINTER statement, on page 294). Logical print channel –1 prints the data on the screen, regardless of whether a PRINTER ON statement has been executed.

page# is an expression that specifies the next page number. If a heading or footing is in effect when the page number is specified, the heading or footing on the current page contains a page number equal to one less than the value of page#.

If either *print.channel* or *page#* evaluates to the null value, the PAGE statement fails and the program terminates with a run-time error message.

Example

In the following example the current value of X provides the next page number:

```
PAGE ON 5 X
```

PERFORM statement

Use the PERFORM statement to execute a UniVerse sentence, paragraph, menu, or command from within the BASIC program, then return execution to the statement following the PERFORM statement. The commands are executed in the same environment as the BASIC program that called them; that is, unnamed common variables, @variables, and in-line prompts retain their values, and select lists and the DATA stack remain active. If these values change, the new values are passed back to the calling program.

Syntax

```
PERFORM command
```

You can specify multiple commands in the PERFORM statement in the same way you specify them in the body of a UniVerse paragraph. Each command or line must be separated by a field mark (ASCII CHAR(254)).

If *command* evaluates to the null value, the PERFORM statement fails and the program terminates with a run-time error message.

You cannot use the PERFORM statement within a transaction to execute most UniVerse commands and SQL statements. However, you can use PERFORM to execute the following UniVerse commands and SQL statements within a transaction:

- CHECK.SUM
- COUNT
- DELETE (SQL)
- DISPLAY
- ESEARCH

- GET.LIST
- INSERT
- LIST
- LIST.LABEL
- LIST.ITEM
- RUN
- SAVE.LIST
- SEARCH
- SELECT (RetrieVe)
- SELECT (SQL)
- SORT
- SORT.LABEL
- SORT.ITEM
- SSELECT
- STAT
- SUM
- UPDATE

REALITY flavor

In a REALITY flavor account PERFORM can take all the clauses of the EXECUTE statement. To get these PERFORM characteristics in other flavor accounts, use the PERF.EQ.EXEC option of the $OPTIONS statement.

Example

In the following example multiple commands are separated by field marks:

```
PERFORM 'RUN BP SUB'
 FM=CHAR(254)
 COMMAND = 'SSELECT EM':FM
 COMMAND := 'RUN BP PAY':FM
 COMMAND := 'DATA 01/10/85'
PERFORM COMMAND
 A = 'SORT EM '
 A := 'WITH PAY.CODE EQ'
 A := '10 AND WITH DEPT'
 A := 'EQ 45'
 PERFORM A
```

PRECISION statement

Use the PRECISION statement to control the maximum number of decimal places that are output when the system converts a numeric value from internal binary format to an ASCII character string value.

Syntax

```
PRECISION expression
```

expression specifies a number from 0 through 14. Any fractional digits in the result of such a conversion that exceed the precision setting are rounded off.

If you do not include a PRECISION statement, a default precision of 4 is assumed. Precisions are stacked so that a BASIC program can change its precision and call a subroutine whose precision is the default unless the subroutine executes a PRECISION statement. When the subroutine returns to the calling program, the calling program has the same precision it had when it called the subroutine.

Trailing fractional zeros are dropped during output. Therefore, when an internal number is converted to an ASCII string, the result might appear to have fewer decimal places than the precision setting allows. However, regardless of the precision setting, the calculation always reflects the maximum accuracy of which the computer is capable (that is, slightly more than 17 total digits, including integers).

If *expression* evaluates to the null value, the PRECISION statement fails and the program terminates with a run-time error message.

Example

```
A = 12.123456789
 PRECISION 8
 PRINT A
 PRECISION 4
 PRINT A
```

This is the program output:

```
12.12345679
 12.1235
```

PrepareXML function

The `PrepareXML` function allocates memory for the XML document, opens the document, determines the file structure of the document, and returns the file structure.

Syntax

```
Status=PrepareXML(xml_file,xml_handle)
```

For `PrepareXML` to complete successfully, you should set the library directory environment variable, which may not be the same name on all systems. For example, the environment variable is called LD_LIBRARY_PATH on Solaris systems, SHLIB_PATH on HP systems, and so on. If this environment variable is not properly set, UniVerse may produce errors such as the following:

ld.so.1: uvsh: fatal: lib*xxxx:* can't open file: errno=2

xxxx may be some unrecognizable combination of letters and numbers. To correct this, set up your environment according to the vendor's instructions

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
xml_file	The path to the file where the XML document resides.
xml_handle	The return value. The return value is the UniVerse BASIC variable for *xml_handle*. Status is one of the following return values: XML.SUCCESS: Success. XML.ERROR: Failed

Example

The following example illustrates use of the `PrepareXML` function:

```
STATUS = PrepareXML("&XML&/MYSTUDENT.XML",STUDENT_XML)
IF STATUS=XML.ERROR THEN
STATUS = XMLError(errmsg)
PRINT "error message ":errmsg
STOP "Error when preparing XML document "
END
```

PRINT statement

Use the PRINT statement to send data to the screen, a line printer, or another print file.

Syntax

PRINT [ON *print.channel*] [*print.list*]

The ON clause specifies the logical print channel to use for output. *print.channel* is an expression that evaluates to a number from –1 through 255. If you do not use the ON clause, logical print channel 0 is used, which prints to the user's terminal if PRINTER OFF is set (see the PRINTER statement, on page 294). If *print.channel* evaluates to the null value, the PRINT statement fails and the program terminates with a run-time error message. Logical print channel –1 prints the data on the screen, regardless of whether a PRINTER ON statement has been executed.

You can specify a HEADING statement, FOOTING statement, $PAGE statement, and PRINTER CLOSE statements for each logical print channel. The contents of the print files are printed in order by logical print channel number.

print.list can contain any BASIC expression. The elements of the list can be numeric or character strings, variables, constants, or literal strings; the null value, however, cannot be printed. The list can consist of a single expression or a series of expressions separated by commas (,) or colons (:) for output formatting. If no *print.list* is designated, a blank line is printed.

Expressions separated by commas are printed at preset tab positions. The default tabstop setting is 10 characters. Calculations for tab characters are based on character length rather than display length. For information about changing the default setting, see the TABSTOP statement, on page 399. Use multiple commas together for multiple tabulations between expressions.

Expressions separated by colons are concatenated. That is, the expression following the colon is printed immediately after the expression preceding the colon. To print a list without a LINEFEED and RETURN, end *print.list* with a colon (:).

If NLS is enabled, calculations for the PRINT statement are based on character length rather than display length. If *print.channel* has a map associated with it, data is mapped before it is output to the device. For more information about maps, see the *UniVerse NLS Guide*.

Examples

```
A=25;B=30
 C="ABCDE"
 PRINT A+B
 PRINT
 PRINT "ALPHA ":C
 PRINT "DATE ":PRINT "10/11/93"
 *
 PRINT ON 1 "FILE 1"
 * The string "FILE 1" is printed on print file 1.
```

This is the program output:

```
55
 ALPHA ABCDE
 DATE 10/11/93
```

The following example clears the screen:

```
PRINT @(-1)
```

The following example prints the letter X at location column 10, row 5:

```
PRINT @(10,5):'X'
```

PRINTER statement

Use the PRINTER statement to direct output either to the screen or to a printer. By default, all output is sent to the screen unless a PRINTER ON is executed or the P option to the RUN command is used. See the `SETPTR` command for more details about redirecting output.

Syntax

`PRINTER` { ON | OFF | RESET }

`PRINTER CLOSE` [ON *print.channel*]

PRINTER ON sends output to the system line printer via print channel 0. The output is stored in a buffer until a PRINTER CLOSE statement is executed or the program terminates; the output is then printed (see the PRINTER CLOSE statement).

PRINTER OFF sends output to the screen via print channel 0. When the program is executed, the data is immediately printed on the screen.

The PRINTER ON or PRINTER OFF statement must precede the PRINT statement that starts the print file.

Use the PRINTER RESET statement to reset the printing options. PRINTER RESET removes the header and footer, resets the page count to 1, resets the line count to 1, and restarts page waiting.

> **Note:** Use TPRINT statement to set a delay before printing. See also the TPARM function, on page 417 statement.

The PRINTER CLOSE statement

Use the PRINTER CLOSE statement to print all output data stored in the printer buffer.

You can specify print channel –1 through 255 with the ON clause. If you omit the ON clause from a PRINTER CLOSE statement, print channel 0 is closed. Only data directed to the printer specified by the ON clause is printed. Therefore, there must be a corresponding PRINTER CLOSE ON *print.channel* for each ON clause specified in a PRINT statement. All print channels are closed when the program stops. Logical print channel –1 prints the data on the screen, regardless of whether a PRINTER ON statement has been executed.

If *print.channel* evaluates to the null value, the PRINTER CLOSE statement fails and the program terminates with a run-time error message.

In PICK, IN2, and REALITY flavor accounts, the PRINTER CLOSE statement closes all print channels.

Example

```
PRINTER ON
 PRINT "OUTPUT IS PRINTED ON PRINT FILE 0"
 PRINTER OFF
 PRINT "OUTPUT IS PRINTED ON THE TERMINAL"
 *
 PRINT ON 1 "OUTPUT WILL BE PRINTED ON PRINT FILE 1"
 PRINT ON 2 "OUTPUT WILL BE PRINTED ON PRINT FILE 2"
```

This is the program output:

```
OUTPUT IS PRINTED ON THE TERMINAL
```

PRINTERR statement

Use the PRINTERR statement to print a formatted error message on the bottom line of the terminal. The message is cleared by the next INPUT @ statement or is overwritten by the next PRINTERR or INPUTERR statement. PRINTERR clears the type-ahead buffer.

Syntax

PRINTERR [*error.message*]

error.message is an expression that evaluates to the error message text. The elements of the expression can be numeric or character strings, variables, constants, or literal strings. The null value cannot be an element because it cannot be output. The expression can be a single expression or a series of expressions separated by commas (,) or colons (:) for output formatting. If no error message is designated, a blank line is printed. If *error.message* evaluates to the null value, the default message is printed:

```
Message ID is NULL: undefined error
```

Expressions separated by commas are printed at preset tab positions. The default tabstop setting is 10 characters. For information about changing the default setting, see the TABSTOP statement, on page 399. Use multiple commas together to cause multiple tabulations between expressions.

Expressions separated by colons are concatenated: that is, the expression following the colon is printed immediately after the expression preceding the colon.

See also the INPUTERR statement, on page 221.

REALITY flavor

In a REALITY flavor account the PRINTERR statement prints a formatted error message from the ERRMSG file on the bottom line of the terminal. REALITY syntax is:

```
PRINTERR [dynamic.array] [FROM file.variable]
```

dynamic.array must contain a record ID and any arguments to the message, with each element separated from the next by a field mark. If *dynamic.array* does not specify an existing record ID, a warning message states that no error message can be found.

If *dynamic.array* evaluates to the null value, the default error message is printed:

```
Message ID is NULL: undefined error
```

The FROM clause lets you read the error message from an open file. If *file.variable* evaluates to the null value, the PRINTERR statement fails and the program terminates with a run-time error message.

This statement is similar to the STOP statement on a Pick system except that it does not terminate the program upon execution. You can use it wherever you can use a STOP or ABORT statement.

To use the REALITY version of the PRINTERR statement in PICK, IN2, INFORMATION, and IDEAL flavor accounts, use the USE.ERRMSG option of the $OPTIONS statement.

UniVerse provides a standard Pick ERRMSG file. You can construct a local ERRMSG file using the following syntax in the records. Each field must start with one of these codes, as shown in the following table:

Code	Action
A[(*n*)]	Display next argument left-justified; *n* specifies field length.
D	Display system date.
E [*string*]	Display record ID of message in brackets; *string* displayed after ID.
H [*string*]	Display *string.*
L [(*n*)]	Output newline; *n* specifies number of newlines.
R [(*n*)]	Display next argument right-justified; *n* specifies field length.
S [(*n*)]	Output *n* blank spaces from beginning of line.
T	Display system time.

PROCREAD statement

Use the PROCREAD statement to assign the contents of the primary input buffer to a variable. Your BASIC program must be called by a proc. If your program was not called from a proc, the ELSE statements are executed; otherwise the THEN statements are executed.

If *variable* evaluates to the null value, the PROCREAD statement fails and the program terminates with a run-time error message.

Syntax

```
PROCREAD variable
      {THEN statements [ELSE statements] | ELSE statements}
```

PROCWRITE statement

Use the PROCWRITE statement to write *string* to the primary input buffer. Your program must be called by a proc.

If *string* evaluates to the null value, the PROCWRITE statement fails and the program terminates with a run-time error message.

Syntax

PROCWRITE *string*

PROGRAM statement

Use the PROGRAM statement to identify a program. The PROGRAM statement is optional; if you use it, it must be the first noncomment line in the program.

name can be specified for documentation purposes; it need not be the same as the actual program name.

Syntax

PROG[RAM] [*name*]

Example

```
PROGRAM BYSTATE
```

PROMPT statement

Use the PROMPT statement to specify the character to be displayed on the screen when user input is required. If no PROMPT statement is issued, the default prompt character is the question mark (?).

Syntax

PROMPT *character*

If *character* evaluates to more than one character, only the first character is significant; all others are ignored.

The prompt character becomes *character* when the PROMPT statement is executed. Although the value of *character* can change throughout the program, the prompt character remains the same until a new PROMPT statement is issued or the program ends.

Generally, data the user enters in response to the prompt appears on the screen. If the source of the input is something other than the keyboard (for example, a DATA statement), the data is displayed on the screen after the prompt character. Use PROMPT " " to prevent any prompt from being displayed. PROMPT " " also suppresses the display of input from DATA statements.

If *character* evaluates to the null value, no prompt appears.

Examples

Source Lines	Program Output
A[(*n*)]	Display next argument left-justified; *n* specifies field length.
D	Display system date.
E [*string*]	Display record ID of message in brackets; *string* displayed after ID.
H [*string*]	Display *string*.
L [(*n*)]	Output newline; *n* specifies number of newlines.
R [(*n*)]	Display next argument right-justified; *n* specifies field length.
S [(*n*)]	Output *n* blank spaces from beginning of line.

Source Lines	Program Output
T	Display system time.

protocolLogging function

The `protocolLogging` function starts or stops logging.

Syntax

protocolLogging(*log_file, log_action, log_level*)

log_file is the name of the file to which the logs will be recorded. The default log file name is httplog and is created under the current directory.

log_action is either ON or OFF. The default is OFF.

log_level is the detail level of logging. Valid values are 0–10. See the table below for information about each log level.

The following table describes each log level.

Log level	Description
0	No logging.
1	Socket open/read/write/close action (no real data) HTTP request: hostinfo(URL)
2	Level 1 logging plus socket data statistics (size, and so forth).
3	Level 2 logging plus all data actually transferred.
4-10	More detailed status data to assist debugging.

Return codes

The following table describes the status of each return code.

Return code	Status
0	Success.
1	Failed to start logging.

PWR function

Use the `PWR` function to return the value of *expression* raised to the power specified by *power*.

Syntax

PWR (*expression, power*)

The `PWR` function operates like exponentiation (that is, PWR(X,Y) is the same as X**Y).

A negative value cannot be raised to a noninteger power. If it is, the result of the function is PWR(-X,Y) and an error message is displayed.

If either *expression* or *power* is the null value, null is returned.

On overflow or underflow, a warning is printed and 0 is returned.

Example

```
A=3
 B=PWR(5,A)
 PRINT "B= ",B
```

This is the program output:

```
B= 125
```

PyCall function

The `PyCall` function calls a Python callable object.

Syntax

pyresult = **PyCall**(*PyCallableObject*[,*arg1*, *arg2*, ...])

Parameters

The following table describes the parameters for this function.

Parameter	Description
pyresult	A standard U2 BASIC variable or a PYOBJECT variable.
pycallableobject	A PYOBJECT variable pointing to a Python object that is callable, such as a function object, class object, or method object.
arg1,arg2,…	The arguments to the callable Python object that can be evaluated to a string, a number, or a PYOBJECT.

PyCallFunction function

The `PyCallFunction` function calls a Python function on a Python module.

Syntax

pyresult = **PyCallFunction**(*moduleName*, *functionName*[, *arg1*, *arg2*, ...])

Parameters

The following table describes the parameters for this function.

Parameter	Description
pyresult	A standard U2 BASIC variable or a PYOBJECT variable.
moduleName	The name of the module where the function is defined.
functionName	The name of the function to be called.
arg1,arg2	The arguments to the function object that can be evaluated to a string, a number, or a PYOBJECT.

PyCallMethod function

The `PyCallMethod` function calls a method on a Python object.

Syntax

```
pyresult = PyCallMethod(pyobject, methodName [,arg1, arg2, ...]
```

Parameters

The following table describes the parameters for this function.

Parameter	Description
pyresult	A standard U2 BASIC variable or a PYOBJECT variable.
pyobject	A PYOBJECT variable pointing to a Python object
methodName	The name of the method to be called. Must be defined on the class of the object.
arg1,arg2	The arguments to the method that can be evaluated to a string, a number, or a PYOBJECT.

PyGetAttr function

The `PyGetAttr` function gets the value of an attribute of a Python object.

Syntax

```
pyresult = PyGetAttr(pyobject, attrName)
```

Parameters

The following table describes the parameters for this function.

Parameter	Description
pyresult	A standard U2 BASIC variable or a PYOBJECT variable.
pyobject	A PYOBJECT variable pointing to a Python object.
attrName	The name of the attribute whose value is to be retrieved.

PyImport function

The `PyImport` function imports a Python module.

Syntax

```
pyresult = PyImport(moduleName)
```

Parameters

The following table describes the parameters for this function.

Parameter	Description
pyresult	A PYOBJECT variable pointing to the Python module object.
moduleName	The name of the module to be imported.

PySetAttr function

The `PySetAttr` function sets the value of an attribute of a Python object.

Syntax

pyresult = **PySetAttr**(*pyobject*, *attrName*, *value*)

Parameters

The following table describes the parameters for this function.

Parameter	Description
pyresult	An integer value, -1: failure.
pyobject	A PYOBJECT variable pointing to a Python object.
attrName	The name of the attribute whose value to be set.
value	A value expression that can be evaluated to a string, a number, or a PYOBJECT.

QUOTE function

Use the `QUOTE` function to enclose an expression in double quotation marks. If *expression* evaluates to the null value, null is returned (without quotation marks).

Syntax

QUOTE (*expression*)

Example

```
PRINT QUOTE(12 + 5) : " IS THE ANSWER."
 END
```

This is the program output:

```
"17" IS THE ANSWER.
```

RAISE function

Use the `RAISE` function to return a value equal to *expression*, except that system delimiters in *expression* are converted to the next higher-level delimiter: value marks are changed to field marks, subvalue marks are changed to value marks, and so on. If *expression* evaluates to the null value, null is returned.

Syntax

RAISE (*expression*)

The conversions are:

IM	CHAR(255)	to	IM	CHAR(255)

FM	CHAR(254)	to	IM	CHAR(255)
VM	CHAR(253)	to	FM	CHAR(254)
SM	CHAR(252)	to	VM	CHAR(253)
TM	CHAR(251)	to	SM	CHAR(252)
	CHAR(250)			CHAR(251)
	CHAR(249)			CHAR(250)
	CHAR(248)			CHAR(249)

PIOPEN flavor

In PIOPEN flavor, the delimiters that can be raised are CHAR(254) through CHAR(251). All other characters are left unchanged. You can obtain PIOPEN flavor for the `RAISE` function by:

- Compiling your program in a PIOPEN flavor account
- Specifying the $OPTIONS INFO.MARKS statement

Examples

In the following examples an item mark is shown by I, a field mark is shown by F, a value mark is shown by V, and a subvalue mark is shown by S.

The following example sets A to DDIEEI123I777:

```
A= RAISE('DD':FM'EE':FM:123:FM:777)
```

The next example sets B to 1I2F3I4V5:

```
B= RAISE(1:IM:2:VM:3:FM:4:SM:5)
```

The next example sets C to 999S888:

```
C= RAISE(999:TM:888)
```

RANDOMIZE statement

Use the RANDOMIZE statement with an expression to make the `RND` function generate the same sequence of random numbers each time the program is run. *expression* must be a positive integer or zero. If no expression is supplied, or if *expression* evaluates to the null value, the internal time of day is used (the null value is ignored). In these cases the sequence is different each time the program is run.

Syntax

RANDOMIZE [(*expression*)]

Example

```
RANDOMIZE (0)
 FOR N=1 TO 10
       PRINT RND(4):'  ':
 NEXT N
 PRINT
 *
 RANDOMIZE (0)
 FOR N=1 TO 10
    PRINT RND(4):'  ':
 NEXT
```

```
PRINT
*
RANDOMIZE (3)
FOR N=1 TO 10
   PRINT RND(4):'  ':
NEXT N
PRINT
```

This is the program output:

```
0 2 1 2 0 2 1 2 1 1
 0 2 1 2 0 2 1 2 1 1
 2 0 1 1 2 1 0 1 2 3
```

READ statements

Use READ statements to assign the contents of a record from a UniVerse file to *dynamic.array*.

Syntax

```
READ  dynamic.array  FROM [file.variable,] record.ID [ON ERROR  statements]
          { THEN  statements [ELSE  statements] | ELSE  statements }

{ READL | READU } dynamic.array FROM [file.variable ,] record.ID
               [ON ERROR  statements] [LOCKED  statements]
           { THEN  statements [ELSE  statements] | ELSE  statements }

READV dynamic.array FROM [file.variable ,] record.ID ,  field#
            [ON ERROR  statements]
          { THEN  statements [ELSE  statements] | ELSE  statements }

{ READVL | READVU } dynamic.array  FROM [file.variable ,] record.ID , field#
                [ON ERROR  statements] [LOCKED  statements]
           { THEN  statements [ELSE  statements] | ELSE  statements }
```

Use this statement...	To do this...
READ	Read a record.
READL	Acquire a shared record lock and read a record.
READU	Acquire an update record lock and read a record.
READV	Read a field.
READVL	Acquire a shared record lock and read a field.
READVU	Acquire an update record lock and read a field.

file.variable specifies an open file. If *file.variable* is not specified, the default file is assumed (for more information on default files, see the OPEN statement, on page 276). If the file is neither accessible nor open, the program terminates with a run-time error message.

If *record.ID* exists on the specified file, *dynamic.array* is set to the contents of the record, and the THEN statements are executed; any ELSE statements are ignored. If no THEN statements are specified, program execution continues with the next statement. If *record.ID* does not exist, *dynamic.array* is set to an empty string, and the ELSE statements are executed; any THEN statements are ignored.

If *file.variable*, *record.ID*, or *field#* evaluate to the null value, the READ statement fails and the program terminates with a run-time error message.

Tables

If the file is a table, the effective user of the program must have SQL SELECT privilege to read records in the file. For information about the effective user of a program, see the AUTHORIZATION statement, on page 71.

Distributed files

If the file is a distributed file, use the `STATUS` function after a READ statement to determine the results of the operation, as follows:

Value	Description
-1	The partitioning algorithm does not evaluate to an integer.
-2	The part number is invalid.

NLS mode

If NLS is enabled, READ and other BASIC statements that perform I/O operations map external data to the UniVerse internal character set using the appropriate map for the input file.

If the file contains unmappable characters, the ELSE statements are executed.

The results of the READ statements depend on all of the following:

- The inclusion of the ON ERROR clause
- The setting of the NLSREADELSE parameter in the `uvconfig` file
- The location of the unmappable character

The values returned by the `STATUS` function are as follows:

Value	Description
3	The unmappable character is in the record ID.
4	The unmappable character is in the record's data.

Note: 4 is returned only if the NLSREADELSE parameter is set to 1. If NLSREADELSE is 0, no value is returned, data is lost, and you see a run-time error message.

For more information about maps, see the *UniVerse NLS Guide*.

The ON ERROR clause

The ON ERROR clause is optional in the READ statement. Its syntax is the same as that of the ELSE clause. The ON ERROR clause lets you specify an alternative for program termination when a fatal error is encountered during processing of the READ statement.

If a fatal error occurs, and the ON ERROR clause was not specified, or was ignored (as in the case of an active transaction), the following occurs:

- An error message appears.
- Any uncommitted transactions begun within the current execution environment roll back.

- The current program terminates.
- Processing continues with the next statement of the previous execution environment, or the program returns to the UniVerse prompt.

A fatal error can occur if any of the following occur:

- A file is not open.
- *file.variable* is the null value.
- A distributed file contains a part file that cannot be accessed.

If the ON ERROR clause is used, the value returned by the `STATUS` function is the error number.

The LOCKED clause

You can use the LOCKED clause only with the READL, READU, READVL, and READVU statements. Its syntax is the same as that of the ELSE clause.

The LOCKED clause handles a condition caused by a conflicting lock (set by another user) that prevents the READ statement from being processed. The LOCKED clause is executed if one of the following conflicting locks exists:

In this statement...	This requested lock...	Conflicts with...
READL READVL	Shared record lock	Exclusive file lock Update record lock
READU READVU	Update record lock	Exclusive file lock Intent file lock Shared file lock Update record lock Shared record lock

If a READ statement does not include a LOCKED clause, and a conflicting lock exists, the program will timeout after 60 minutes or until the lock is released.

If a LOCKED clause is used, the value returned by the `STATUS` function is the terminal number of the user who owns the conflicting lock.

Releasing Locks

A shared record lock can be released with a CLOSE statement, RELEASE statement, or STOP statement. An update record lock can be released with a CLOSE statement, DELETE statements, MATWRITE statements, RELEASE statement, STOP, WRITE statements, or WRITEV statement.

Locks acquired or promoted within a transaction are not released when the previous statements are processed.

All record locks are released when you return to the UniVerse prompt.

READL and READU statements

Use the READL syntax to acquire a shared record lock and then read the record. This allows other programs to read the record with no lock or a shared record lock.

Use the READU statement to acquire an update record lock and then read the record. The update record lock prevents other users from updating the record until the user who owns it releases it.

An update record lock can only be acquired when no shared record lock exists. It can be promoted from a shared record lock owned by the user requesting the update record lock if no shared record locks exist.

To prevent more than one program or user from modifying the same record at the same time, use READU instead of READ.

READV, READVL, and READVU statements

Use the READV statement to assign the contents of a field in a UniVerse file record to *dynamic.array*.

Use the READVL statement to acquire a shared record lock and then read a field from the record. The READVL statement conforms to all the specifications of the READL and READV statements.

Use the READVU statement to acquire an update record lock and then read a field from the record. The READVU statement conforms to all the specifications of the READU and READV statements.

You can specify *field#* only with the READV, READVL, and READVU statements. It specifies the index number of the field to be read from the record. You can use a *field#* of 0 to determine whether the record exists. If the field does not exist, *dynamic.array* is assigned the value of an empty string.

PICK, IN2, and REALITY flavors

In PICK, IN2, and REALITY flavor accounts, if *record.ID* or *field#* does not exist, *dynamic.array* retains its value and is not set to an empty string. The ELSE statements are executed; any THEN statements are ignored. To specify PICK, IN2, and REALITY flavor READ statements in an INFORMATION or IDEAL flavor account, use the READ.RETAIN option of the $OPTIONS statement.

Examples

```
OPEN '','SUN.MEMBER' TO FILE ELSE STOP
 FOR ID=5000 TO 6000
    READ MEMBER FROM FILE, ID THEN PRINT ID ELSE NULL
 NEXT ID
OPEN '','SUN.SPORT' ELSE STOP 'CANT OPEN "SUN.SPORT"'
 READ ID FROM "853333" ELSE
    PRINT 'CANT READ ID "853333" ON FILE "SUN.SPORT"'
 END
X="6100"
 READ PERSON FROM FILE,X THEN PRINT PERSON<1> ELSE
    PRINT "PERSON ":X:" NOT ON FILE"
 END
```

The next example locks the record N in the file SUN.MEMBER, reads field 3 (STREET) from it, and prints the value of the field:

```
OPEN '','SUN.MEMBER' TO FILE ELSE STOP
 FOR N=5000 TO 6000
    READVU STREET FROM FILE,N,3 THEN PRINT STREET ELSE NULL
    RELEASE
  NEXT
OPEN "DICT","MYFILE" TO DICT.FILE ELSE STOP
 OPEN "","MYFILE" ELSE STOP ; *USING DEFAULT FILE VARIABLE
 READU ID.ITEM FROM DICT.FILE,"@ID" ELSE
    PRINT "NO @ID"
    STOP
 END
```

This is the program output:

```
5205
 5390
 CANT READ ID "853333" ON FILE "SUN.SPORT"
 MASTERS
 4646 TREMAIN DRIVE
 670 MAIN STREET
```

READBLK statement

Use the READBLK statement to read a block of data of a specified length from a file opened for sequential processing and assign it to a variable. The READBLK statement reads a block of data beginning at the current position in the file and continuing for *blocksize* bytes and assigns it to *variable*. The current position is reset to just beyond the last byte read.

Syntax

```
READBLK variable FROM file.variable, blocksize
        THEN statements [ELSE statements] | ELSE statements }
```

file.variable specifies a file previously opened for sequential processing.

If the data can be read from the file, the THEN statements are executed; any ELSE statements are ignored. If the file is not readable or if the end of file is encountered, the ELSE statements are executed and the THEN statements are ignored. If the ELSE statements are executed, *variable* is set to an empty string.

If either *file.variable* or *blocksize* evaluates to the null value, the READBLK statement fails and the program terminates with a run-time error message.

Note: A newline in UNIX files is one byte long, whereas in Windows NT it is two bytes long. This means that for a file with newlines, the same READBLK statement may return a different set of data depending on the operating system the file is stored under.

In the event of a timeout, READBLK returns no bytes from the buffer, and the entire I/O operation must be retried.

The difference between the READSEQ statement and the READBLK statement is that the READBLK statement reads a block of data of a specified length, whereas the READSEQ statement reads a single line of data.

On Windows NT systems, if you use READBLK to read data from a 1/4-inch cartridge drive (60 or 150 MB) that you open with the OPENDEV statement, on page 280, you must use a block size of 512 bytes or a multiple of 512 bytes.

For more information about sequential file processing, see the OPENSEQ statement, on page 283, READSEQ statement, on page 310, and WRITESEQ statement, on page 454.

If NLS is enabled and *file.variable* has a map associated with it, the data is mapped accordingly. For more information about maps, see the *UniVerse NLS Guide*.

Example

```
OPENSEQ 'FILE.E', 'RECORD4' TO FILE ELSE ABORT
 READBLK VAR1 FROM FILE, 15 THEN PRINT VAR1
```

```
 PRINT
 READBLK VAR2 FROM FILE, 15 THEN PRINT VAR2
```

This is the program output:

```
FIRST LINE
 SECO

 ND LINE
 THIRD L
```

READL statement

Use the READL statement to acquire a shared record lock and perform the READ statement.

For details, see the READ statements, on page 303.

READLIST statement

Use the READLIST statement to read the remainder of an active select list into a dynamic array.

Syntax

```
READLIST dynamic.array [FROM list.number]
        { THEN statements [ELSE statements] | ELSE statements }
```

list.number is an expression that evaluates to the number of the select list to be read. It can be from 0 through 10. If you do not use the FROM clause, select list 0 is used.

READLIST reads all elements in the active select list. If READ statements are used on the select list before the READLIST statement, only the elements not read by the READNEXT statement are stored in *dynamic.array*. READLIST empties the select list.

If one or more elements are read from *list.number*, the THEN statements are executed. If there are no more elements in the select list or if a select list is not active, the ELSE statements are executed; any THEN statements are ignored.

If *list.number* evaluates to the null value, the READLIST statement fails and the program terminates with run-time error message.

In IDEAL and INFORMATION flavor accounts, use the VAR.SELECT option of the $OPTIONS statement to get READLIST to behave as it does in PICK flavor accounts.

PICK, REALITY, and IN2 flavors

In PICK, REALITY, and IN2 flavor accounts, the READLIST statement has the following syntax:

```
READLIST dynamic.array FROM listname [SETTING variable]
     {THEN statements [ELSE statements] | ELSE statements}
```

In these flavors the READLIST statement reads a saved select list from the &SAVEDLISTS& file without activating a select list. In PICK and IN2 flavor accounts, READLIST lets you access a saved select list without changing the currently active select list if there is one.

The select list saved in *listname* in the &SAVEDLISTS& file is put in *dynamic.array*. The elements of the list are separated by field marks.

listname can be of the form

record.ID

or

record.IDaccount.name

record.ID specifies the record ID of the list in &SAVEDLISTS&, and *account.name* specifies the name of another UniVerse account in which to look for the &SAVEDLISTS& file.

The SETTING clause assigns the count of the elements in the list to *variable*.

If the list is retrieved successfully (the list must not be empty), the THEN statements are executed; if not, the ELSE statements are executed. If *listname* evaluates to the null value, the READLIST statement fails and the program terminates with a run-time error message.

In PICK, REALITY, and IN2 flavor accounts, use the -VAR.SELECT option of the $OPTIONS statement to get READLIST to behave as it does in IDEAL flavor accounts.

READNEXT statement

Use the READNEXT statement to assign the next record ID from an active select list to *dynamic.array*.

Syntax

```
READNEXT dynamic.array [,value [,subvalue]] [FROM list]
        {THEN statements [ELSE statements] | ELSE statements}
```

list specifies the select list. If none is specified, select list 0 is used. *list* can be a number from 0 through 10 indicating a numbered select list, or the name of a select list variable.

The BASIC SELECT statements, on page 340 or the UniVerse `GET.LIST`, `FORM.LIST`, `SELECT`, or `SSELECT` commands create an active select list; these commands build the list of record IDs. The READNEXT statement reads the next record ID on the list specified in the FROM clause and assigns it to the *dynamic.array*.

When the select list is exhausted, *dynamic.array* is set to an empty string, and the ELSE statements are executed; any THEN statements are ignored.

If *list* evaluates to the null value, the READNEXT statement fails and the program terminates with a run-time error message.

A READNEXT statement with *value* and *subvalue* specified accesses an exploded select list. The record ID is stored in *dynamic.array*, the value number in *value*, and the subvalue number in *subvalue*. If only *dynamic.array* is specified, it is set to a multivalued field consisting of the record ID, value number, and subvalue number, separated by value marks.

INFORMATION flavor

In INFORMATION flavor accounts READNEXT returns an exploded select list. Use the RNEXT.EXPL option of the $OPTIONS statement to return exploded select lists in other flavors.

Example

```
OPEN '','SUN.MEMBER' ELSE STOP "CAN'T OPEN FILE"
 SELECT TO 1
 10: READNEXT MEM FROM 1 THEN PRINT MEM ELSE GOTO 15:
 GOTO 10:
```

```
 *
 15: PRINT
OPEN '','SUN.SPORT' TO FILE ELSE STOP
 SELECT FILE
 COUNT=0
 20*
 READNEXT ID ELSE
 PRINT 'COUNT= ',COUNT
 STOP
 END
 COUNT=COUNT+1
 GOTO 20
```

This is the program output:

```
4108
 6100
 3452
 5390
 7100
 4500
 2430
 2342
 6783
 5205
 4439
 6203
 7505
 4309
 1111
 COUNT= 14
```

READSEQ statement

Use the READSEQ statement to read a line of data from a file opened for sequential processing. Sequential processing lets you process data one line at a time. UniVerse keeps a pointer at the current position in the file. The $OPTIONS statement sets this pointer to the first byte of the file, and it is advanced by READSEQ, READBLK statement, WRITESEQ statement, and WRITEBLK statement.

Syntax

```
READSEQ variable FROM file.variable [ON ERROR statements]
        {THEN statements [ELSE statements] | ELSE statements}
```

Each READSEQ statement reads data from the current position in the file up to a newline and assigns it to *variable*. The pointer is then set to the position following the newline. The newline is discarded.

file.variable specifies a file previously opened for sequential processing. The FROM clause is required. If the file is neither accessible nor open, or if *file.variable* evaluates to the null value, the READSEQ statement fails and the program terminates with a run-time error message.

If data is read from the file, the THEN statements are executed, and the ELSE statements are ignored. If the file is not readable, or the end of file is encountered, the ELSE statements are executed; any THEN statements are ignored.

In the event of a timeout, READSEQ returns no bytes from the buffer, and the entire I/O operation must be retried.

READSEQ affects the STATUS function in the following way:

Value	Description
0	The read is successful.
1	The end of file is encountered.
2	A timeout ended the read.
-1	The file is not open for a read.

If NLS is enabled, the READSEQ and other BASIC statements that perform I/O operations always map external data to the UniVerse internal character set using the appropriate map for the input file if the file has a map associated with it. For more information about maps, see the *UniVerse NLS Guide*.

The ON ERROR clause

The ON ERROR clause is optional in the READSEQ statement. Its syntax is the same as that of the ELSE clause. The ON ERROR clause lets you specify an alternative for program termination when a fatal error is encountered during processing of the READSEQ statement.

If a fatal error occurs, and the ON ERROR clause was not specified, or was ignored (as in the case of an active transaction), the following occurs:

- An error message appears.
- Any uncommitted transactions begun within the current execution environment roll back.
- The current program terminates.
- Processing continues with the next statement of the previous execution environment, or the program returns to the UniVerse prompt.

A fatal error can occur if any of the following occur:

- A file is not open.
- *file.variable* is the null value.
- A distributed file contains a part file that cannot be accessed.

If the ON ERROR clause is used, the value returned by the STATUS function is the error number.

Example

```
OPENSEQ 'FILE.E', 'RECORD4' TO FILE ELSE ABORT
 FOR N=1 TO 3
    READSEQ A FROM FILE THEN PRINT A
 NEXT N
 CLOSESEQ FILE
```

This is the program output:

```
FIRST LINE
 SECOND LINE
 THIRD LINE
```

readSocket function

Use the readSocket() function to read data in the socket buffer up to *max_read_size* characters.

Syntax

```
readSocket(socket_handle, socket_data, max_read_size, time_out, mode,
actual_read_size)
```

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
socket_handle	A handle to the open socket.
socket_data	The data to be read from the socket.
max_read_size	The maximum number of characters to return. If this is 0, then the entire buffer should be returned.
time_out	The time (in milliseconds) before a return in blocking mode. This is ignored for non-blocking read.
mode	0: using current mode 1: blocking mode (default) 2: non-blocking mode
actual_read_size	The number of characters actually read. -1 if error.

Return status

The following table describes the return status of each mode.

Mode	Return status
Non-blocking	The function will return immediately if there is no data in the socket. If the *max_read_size* parameter is greater than the socket buffer then just the socket buffer will be returned.
Blocking	If there is no data in the socket, the function will block until data is put into the socket on the other end. It will return up to the *max_read_size* character setting.

Return codes

The following table describes the status of each return code.

Return code	Status
0	Success.
1-41	See Socket function error return codes, on page 584.
107	Encryption error.
108	Decryption error.

READT statement

Use the READT statement to read the next tape record from a magnetic tape unit and assign its contents to a variable.

Syntax

```
READT [UNIT (mtu)] variable
      {THEN statements [ELSE statements] | ELSE statements}
```

The UNIT clause specifies the number of the tape drive unit. Tape unit 0 is used if no unit is specified.

mtu is an expression that evaluates to a code made up of three decimal digits, as shown in the following table:

Code	Available Options
m (mode)	0 = No conversion 1 = EBCDIC conversion 2 = Invert high bit 3 = Invert high bit and EBCDIC conversion
t (tracks)	0 = 9 tracks. Only 9-track tapes are supported.
u (unit number)	0 through 7

The *mtu* expression is read from right to left. Therefore, if *mtu* evaluates to a one-digit code, it represents the tape unit number. If *mtu* evaluates to a two-digit code, the rightmost digit represents the unit number and the digit to its left is the track number; and so on.

If either *mtu* or *variable* evaluates to the null value, the READT statement fails and the program terminates with a run-time error message.

Each tape record is read and processed completely before the next record is read. The program waits for the completion of data transfer from the tape before continuing.

If the next tape record exists, *variable* is set to the contents of the record, and the THEN statements are executed. If no THEN statements are specified, program execution continues with the next statement.

Before a READT statement is executed, a tape drive unit must be attached (assigned) to the user. Use the `ASSIGN` command to assign a tape unit to a user. If no tape unit is attached or if the unit specification is incorrect, the ELSE statements are executed and the value assigned to *variable* is empty. Any THEN statements are ignored.

The largest tape record that the READT statement can read is system-dependent. If a tape record is larger than the system maximum, only the bytes up to the maximum are assigned to *variable*.

The STATUS function returns 1 if READT takes the ELSE clause, otherwise it returns 0.

If NLS is enabled, the READT and other BASIC statements that perform I/O operations always map external data to the UniVerse internal character set using the appropriate map for the input file if the file has a map associated with it. For more information about maps, see the *UniVerse NLS Guide*.

PIOPEN flavor

If you have a program that specifies the syntax UNIT *ndmtu*, the *nd* elements are ignored by the compiler and no errors are reported.

Examples

The following example reads a tape record from tape drive 0:

```
READT RECORD ELSE PRINT "COULD NOT READ FROM TAPE"
```

The next example reads a record from tape drive 3, doing an EBCDIC conversion in the process:

```
READT UNIT(103) RECORD ELSE PRINT "COULD NOT READ"
```

READU statement

Use the READU statement to set an update record lock and perform the READ statement.

For details, see the READ statements, on page 303.

In 8.3.3.2, a new LOCK.WAIT clause was introduced into the SQL environment defaulting to 3600 seconds (60 Minutes), which caused the BASIC READU statement to follow that parameter. As a result, if a program uses the READU statement with no LOCKED clause and waits for 60 minutes attempting to gain that lock, because it is already locked somewhere else, the program will proceed with the ELSE clause of the READU statement. This will cause problems for most developers because the ELSE clause will normally be followed by REC = "" which could result in unwanted results in the rest of the program / application in that any subsequent WRITES in that program would be allowed.

The two methods to change this behavior are from UniVerse BASIC or from TCL via SQL. From BASIC you can `ASSIGN` a value to SYSTEM(1999) denoting the number of seconds to wait on a READU. For example, to set the wait time to 2 hours, you would use:

```
ASSIGN 7200 TO SYSTEM(1999)
```

You can also specify an indefinite wait time using:

```
ASSIGN 0 TO SYSTEM(1999)
```

Note: You cannot inquire on the current contents of SYSTEM(1999). It will always report 0 no matter what has previously been assigned.

From TCL, you can increase the LOCK.WAIT time parameter as follows:

```
SET.SQL LOCK.WAIT n
```

where *n* is a number of seconds. *n* must be > 0.

Both of these methods can be done via an account LOGIN paragraph through an account change or using the UV.LOGIN paragraph to set system wide. This parameter will stay set for the duration of the UniVerse session.

Note: UniVerse also has a write timeout of 20 Minutes on any locked record and at present there is no method to override this.

READV statement

Use the READV statement to read the contents of a specified field of a record in a UniVerse file.

For details, see the READ statements, on page 303.

READVL statement

Use the READVL statement to set a shared record lock and perform the READV statement.

For details, see the READ statements, on page 303.

READVU statement

Use the READVU statement to set an update record lock and read the contents of a specified field of a record in a UniVerse file.

For details, see the READ statements, on page 303.

ReadXMLData function

After you open an XML document, read the document using the `ReadXMLData` function. UniVerse BASIC returns the XML data as a dynamic array.

Syntax

```
Status=ReadXMLData(xml_data_handle, rec)
```

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
xml_data_handle	A variable that holds the XML data handle created by the `OpenXMLData` function.
rec	A mark-delimited dynamic array containing the extracted data. Status if one of the following: XML.SUCCESS: Success. XML.ERROR: Failed XML.INVALID.HANDLE2: Invalid *xml_data_handle* XML.EOF: End of data

After you read the XML document, you can execute any UniVerse BASIC statement or function against the data.

Example

The following example illustrates use of the `ReadXMLData` function:

```
MOREDATA=1
LOOP WHILE (MOREDATA=1)
status = ReadXMLData(STUDENT_XML,rec)
IF status = XML.ERROR THEN
STOP "Error when preparing the XML document. "
END ELSE IF status = XML.EOF THEN
PRINT "No more data"
MOREDATA = 0
END ELSE
PRINT "rec = ":rec
END
REPEAT
```

REAL function

Use the REAL function to convert *number* into a floating-point number without loss of accuracy. If *number* evaluates to the null value, null is returned.

Syntax

```
REAL (number)
```

RECORDLOCK statements

Use RECORDLOCK statements to acquire a record lock on a record without reading the record.

Syntax

```
RECORDLOCKL file.variable , record.ID [ON ERROR statements]
        [LOCKED statements]

RECORDLOCKU file.variable , record.ID [ON ERROR statements]
       [LOCKED statements]
```

Use this statement...	To acquire this lock without reading the record...
RECORDLOCKL	Shared record lock
RECORDLOCKU	Update record lock

file.variable is a file variable from a previous OPEN statement.

record.ID is an expression that evaluates to the record ID of the record that is to be locked.

The RECORDLOCKL statement

The RECORDLOCKL statement lets other users lock the record using RECORDLOCK or any other statement that sets a shared record lock, but cannot gain exclusive control over the record with FILELOCK statement, or any statement that sets an update record lock.

The RECORDLOCKU statement

The RECORDLOCKU statement prevents other users from accessing the record using a FILELOCK statement or any statement that sets either a shared record lock or an update record lock. You can reread a record after you have locked it; you are not affected by your own locks.

The ON ERROR clause

The ON ERROR clause is optional in RECORDLOCK statements. The ON ERROR clause lets you specify an alternative for program termination when a fatal error is encountered while a RECORDLOCK statement is being processed.

If a fatal error occurs, and the ON ERROR clause was not specified, or was ignored (as in the case of an active transaction), the following occurs:

- An error message appears.
- Any uncommitted transactions begun within the current execution environment roll back.

- The current program terminates.
- Processing continues with the next statement of the previous execution environment, or the program returns to the UniVerse prompt.

A fatal error can occur if any of the following occur:

- A file is not open.
- *file.variable* is the null value.
- A distributed file contains a part file that cannot be accessed.

If the ON ERROR clause is used, the value returned by the STATUS function is the error number.

The LOCKED clause

The LOCKED clause is optional, but recommended.

The LOCKED clause handles a condition caused by a conflicting lock (set by another user) that prevents the RECORDLOCK statement from processing. The LOCKED clause is executed if one of the following conflicting locks exists:

In this statement...	This requested lock...	Conflicts with these locks...
RECORDLOCKL	Shared record lock	Exclusive file lock Update record lock
RECORDLOCKU	Update record lock	Exclusive file lock Intent file lock Shared file lock Update record lock Shared record lock

If the RECORDLOCK statement does not include a LOCKED clause, and a conflicting lock exists, the program pauses until the lock is released.

If a LOCKED clause is used, the value returned by the `STATUS` function is the terminal number of the user who owns the conflicting lock.

Releasing locks

A shared record lock can be released with a CLOSE statement, RELEASE statement, or STOP statement. An update record lock can be released with a CLOSE statement, DELETE statements, MATWRITE statements, RELEASE statement, STOP statement, WRITE statements, or WRITEV statement.

Locks acquired or promoted within a transaction are not released when the previous statements are processed.

All record locks are released when you return to the UniVerse prompt.

Example

In the following example, the file EMPLOYEES is opened. Record 23694 is locked. If the record was already locked, the program terminates, and an appropriate message is displayed. The RECORDLOCKL statement allows other users to read the record with READL or lock it with another RECORDLOCKL, but prevents any other user from gaining exclusive control over the record.

```
OPEN '','EMPLOYEES' TO EMPLOYEES ELSE STOP 'Cannot open file'
```

```
RECORDLOCKL EMPLOYEES,'23694'
    LOCKED STOP 'Record previously locked by user ':STATUS()
```

RECORDLOCKED function

Use the RECORDLOCKED function to return the status of a record lock.

Syntax

RECORDLOCKED (*file.variable , record.ID*)

file.variable is a file variable from a previous OPEN statement.

record.ID is an expression that evaluates to the record ID of the record that is to be checked.

An insert file of equate names is provided to let you use mnemonics (see the following table). The insert file is called RECORDLOCKED.INS.IBAS, and is located in the INCLUDE directory in the UV account directory. In PIOPEN flavor accounts, the VOC file has a file pointer called SYSCOM. SYSCOM references the INCLUDE directory in the UV account directory.

To use the insert file, specify $INCLUDE SYSCOM RECORDLOCKED.INS.IBAS when you compile the program.

Equate name	Value	Meaning
LOCK$MY.FILELOCK	3	This user has a FILELOCK.
LOCK$MY.READU	2	This user has an update record lock.
LOCK$MY.READL	1	This user has a shared record lock.
LOCK$NO.LOCK	0	The record is not locked.
LOCK$OTHER.READL	–1	Another user has a shared record lock.
LOCK$OTHER.READU	–2	Another user has an update record lock.
LOCK $OTHER.FILELOCK	–3	Another user has a FILELOCK.

If you have locked the file, the RECORDLOCKED function indicates only that you have the file lock for that record. It does not indicate any update record or shared record lock that you also have on the record.

Value returned by the STATUS function

Possible values returned by the STATUS function, and their meanings, are as follows:

Return value	Description
> 0	A positive value is the terminal number of the owner of the lock (or the first terminal number encountered, if more than one user has locked records in the specified file).
< 0	A negative value is –1 times the terminal number of the remote user who has locked the record or file.

Examples

The following program checks to see if there is an update record lock or FILELOCK held by the current user on the record. If the locks are not held by the user, the ELSE clause reminds the user that an

update record lock or FILELOCK is required on the record. This example using the SYSCOM file pointer, only works in PI/open flavor accounts.

```
$INCLUDE SYSCOM RECORDLOCKED.INS.IBAS
 OPEN '','EMPLOYEES' TO EMPLOYEES
    ELSE STOP 'CANNOT OPEN FILE'
    .
    .
    .
 IF RECORDLOCKED(EMPLOYEES,RECORD.ID) >= LOCK$MY.READU THEN
    GOSUB PROCESS.THIS.RECORD:
 ELSE PRINT 'Cannot process record : ':RECORD.ID :', READU or FILELOCK required.'
```

The next program checks to see if the record lock is held by another user and prints a message where the `STATUS` function gives the terminal number of the user who holds the record lock:

```
$INCLUDE SYSCOM RECORDLOCKED.INS.IBAS
 OPEN '','EMPLOYEES' TO EMPLOYEES
 ELSE STOP 'CANNOT OPEN FILE'
 .
 .
 .
 IF RECORDLOCKED(EMPLOYEES,RECORD.ID) < LOCK$NO.LOCK THEN
    PRINT 'Record locked by user' : STATUS()
 END
```

RELEASE statement

Use the RELEASE statement to unlock, or release, locks set by a FILELOCK statement, MATREADL statement, MATREADU statement, READL statement, READU statement, READVL statement, READVU statement, and OPENSEQ statement. These statements lock designated records to prevent concurrent updating by other users. If you do not explicitly release a lock that you have set, it is unlocked automatically when the program terminates.

Syntax

RELEASE [*file.variable* [,*record.ID*]] [ON ERROR *statements*]

file.variable specifies an open file. If *file.variable* is not specified and a record ID is specified, the default file is assumed (for more information on default files, see the OPEN statement, on page 276). If the file is neither accessible nor open, the program terminates with a run-time error message.

record.ID specifies the lock to be released. If it is not specified, all locks in the specified file (that is, either *file.variable* or the default file) are released. If either *file.variable* or *record.ID* evaluates to the null value, the RELEASE statement fails and the program terminates with a run-time error message.

When no options are specified, all locks in all files set by any FILELOCK, READL, READU, READVL, READVU, WRITEU, WRITEVU, MATREADL, MATREADU, MATWRITEU, or OPENSEQ statements during the current login session are released.

A RELEASE statement within a transaction is ignored.

The ON ERROR Clause

The ON ERROR clause is optional in the RELEASE statement. The ON ERROR clause lets you specify an alternative for program termination when a fatal error is encountered during processing of the RELEASE statement.

If a fatal error occurs, and the ON ERROR clause was not specified, or was ignored (as in the case of an active transaction), the following occurs:

- An error message appears.
- Any uncommitted transactions begun within the current execution environment roll back.
- The current program terminates.
- Processing continues with the next statement of the previous execution environment, or the program returns to the UniVerse prompt.

A fatal error can occur if any of the following occur:

- A file is not open.
- *file.variable* is the null value.
- A distributed file contains a part file that cannot be accessed.

If the ON ERROR clause is used, the value returned by the STATUS function is the error number.

Examples

The following example releases all locks set in all files by the current program:

```
RELEASE
```

The next example releases all locks set in the NAMES file:

```
RELEASE NAMES
```

The next example releases the lock set on the record QTY in the PARTS file:

```
RELEASE PARTS, "QTY"
```

ReleaseXML function

Release the XML dynamic array after closing it using the `ReleaseXML` function. `ReleaseXML` destroys the internal DOM tree and releases the associated memory.

Syntax

`ReleaseXML``(`*`XMLhandle`*`)`

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
XMLhandle	The XML handle created by the `PrepareXML`() function.

REM function

Use the `REM` function to calculate the remainder after integer division is performed on the dividend expression by the divisor expression.

Syntax

`REM` `(`*`dividend, divisor`*`)`

The REM function calculates the remainder using the following formula:

REM (X, Y) = X - (INT (X / Y) * Y)

dividend and *divisor* can evaluate to any numeric value, except that *divisor* cannot be 0. If *divisor* is 0, a division by 0 warning message is printed, and 0 is returned. If either *dividend* or *divisor* evaluates to the null value, null is returned.

The REM function works like the MOD function, on page 263.

Example

```
X=85; Y=3
 PRINT 'REM (X,Y)= ',REM (X,Y)
```

This is the program output:

```
REM (X,Y)= 1
```

REM statement

Use the REM statement to insert a comment in a BASIC program. Comments explain or document various parts of a program. They are part of the source code only and are nonexecutable. They do not affect the size of the object code.

A comment must be a separate BASIC statement, and can appear anywhere in a program. A comment must be one of the following comment designators:

REM * ! $*

Any text that appears between a comment designator and the end of a physical line is treated as part of the comment. If a comment does not fit on one physical line, it can be continued on the next physical line only by starting the new line with a comment designator. If a comment appears at the end of a physical line containing an executable statement, you must treat it as if it were a new statement and put a semicolon (;) after the executable statement, before the comment designator.

Syntax

REM [*comment.text*]

Example

```
PRINT "HI THERE"; REM This part is a comment.
 REM This is also a comment and does not print.
 REM
 IF 5<6 THEN PRINT "YES"; REM A comment; PRINT "PRINT ME"
 REM BASIC thinks PRINT "PRINT ME" is also part
 REM of the comment.
 IF 5<6 THEN
    PRINT "YES"; REM Now it doesn't.
    PRINT "PRINT ME"
 END
```

This is the program output:

```
HI THERE
 YES
 YES
```

```
PRINT ME
```

REMOVE function

Use the REMOVE function to successively extract and return dynamic array elements that are separated by system delimiters, and to indicate which system delimiter was found. When a system delimiter is encountered, the value of the extracted element is returned. The REMOVE function is more efficient than the EXTRACT function for extracting successive fields, values, and so on, for multivalue list processing.

Syntax

REMOVE (*dynamic.array, variable*)

dynamic.array is the dynamic array from which to extract elements.

variable is set to a code corresponding to the system delimiter which terminates the extracted element. The contents of *variable* indicate which system delimiter was found, as follows:

Value	Description
0	End of string
1	Item mark ASCII CHAR(255)
2	Field mark ASCII CHAR(254)
3	Value mark ASCII CHAR(253)
4	Subvalue mark ASCII CHAR(252)
5	Text mark ASCII CHAR(251)
6	ASCII CHAR(250) **Note:** Not available in the PIOPEN flavor
7	ASCII CHAR(249) **Note:** Not available in the PIOPEN flavor
8	ASCII CHAR(248) **Note:** Not available in the PIOPEN flavor

The REMOVE function extracts one element each time it is executed, beginning with the first element in *dynamic.array*. The operation can be repeated until all elements of *dynamic.array* are extracted. The REMOVE function does not change the dynamic array.

As each successive element is extracted from *dynamic.array*, a pointer associated with *dynamic.array* is set to the beginning of the next element to be extracted. Thus the pointer is advanced every time the REMOVE function is executed.

The pointer is reset to the beginning of *dynamic.array* whenever *dynamic.array* is reassigned. Therefore, *dynamic.array* should not be assigned a new value until all elements have been extracted (that is, until *variable* is 0).

If *dynamic.array* evaluates to the null value, null is returned and *variable* is set to 0 (end of string). If an element in *dynamic.array* is the null value, null is returned for that element, and *variable* is set to the appropriate delimiter code.

Unlike the EXTRACT function, the REMOVE function maintains a pointer into the dynamic array. (The EXTRACT function always starts processing at the beginning of the dynamic array, counting field marks, value marks, and subvalue marks until it finds the correct element to extract.)

See the REMOVE statement, on page 323 for the statement equivalent of this function.

Examples

The first example sets the variable FIRST to the string MIKE and the variable X to 2 (field mark). The second example executes the REMOVE function and PRINT statement until all the elements have been extracted, at which point A = 0. Printed lines are 12, 4, 5, 7654, and 00.

Source lines	Program output
FM=CHAR(254) NAME='MIKE':FM:'JOHN':FM X=REMOVE(NAME,FIRST) PRINT 'FIRST = ':FIRST, 'X = ':X	FIRST = 2 X = MIKE
VM=CHAR(253) A = 1 Z=12:VM:4:VM:5:VM:7654:VM:00 FOR X=1 TO 20 UNTIL A=0 A = REMOVE(Z,Y) PRINT 'Y = ':Y, 'A = ':A NEXT X	Y = 3 A = 12 Y = 3 A = 4 Y = 3 A = 5 Y = 3 A = 7654 Y = 0 A = 0

REMOVE statement

Use the REMOVE statement to successively extract dynamic array elements that are separated by system delimiters. When a system delimiter is encountered, the extracted element is assigned to *element*. The REMOVE statement is more efficient than the EXTRACT function for extracting successive fields, values, and so on, for multivalue list processing.

Syntax

REMOVE *element* FROM *dynamic.array* SETTING *variable*

dynamic.array is the dynamic array from which to extract elements.

variable is set to a code value corresponding to the system delimiter terminating the element just extracted. The delimiter code settings assigned to *variable* are as follows:

Value	Description
0	End of string
1	Item mark ASCII CHAR(255)
2	Field mark ASCII CHAR(254)
3	Value mark ASCII CHAR(253)
4	Subvalue mark ASCII CHAR(252)
5	Text mark ASCII CHAR(251)
6	ASCII CHAR(250) **Note:** Not supported in the PIOPEN flavor

Value	Description
7	ASCII CHAR(249) **Note:** Not supported in the PIOPEN flavor
8	ASCII CHAR(248) **Note:** Not supported in the PIOPEN flavor

The REMOVE statement extracts one element each time it is executed, beginning with the first element in *dynamic.array*. The operation can be repeated until all elements of *dynamic.array* are extracted. The REMOVE statement does not change the dynamic array.

As each element is extracted from *dynamic.array* to *element*, a pointer associated with *dynamic.array* is set to the beginning of the next element to be extracted. Thus, the pointer is advanced every time the REMOVE statement is executed.

The pointer is reset to the beginning of *dynamic.array* whenever *dynamic.array* is reassigned. Therefore, *dynamic.array* should not be assigned a new value until all elements have been extracted (that is, until *variable* = 0).

If an element in *dynamic.array* is the null value, null is returned for that element.

Unlike the `EXTRACT` function, the REMOVE statement maintains a pointer into the dynamic array. (The `EXTRACT` function always starts processing at the beginning of the dynamic array, counting field marks, value marks, and subvalue marks until it finds the correct element to extract.)

See the REMOVE function, on page 322 for the function equivalent of this statement.

Examples

The first example sets the variable FIRST to the string MIKE and the variable X to 2 (field mark).
The second example executes the REMOVE and PRINT statements until all the elements have been extracted, at which point A = 0. Printed lines are 12, 4, 5, 7654, and 00.

Source lines	Program output
FM=CHAR(254) NAME='MIKE':FM:'JOHN':FM REMOVE FIRST FROM NAME SETTING X PRINT 'X= ':X, 'FIRST= ':FIRST	X= 2 FIRST= MIKE
VM=CHAR(253) A=1 Z=12:VM:4:VM:5:VM:7654:VM:00 FOR X=1 TO 20 UNTIL A=0 REMOVE Y FROM Z SETTING A PRINT 'Y= ':Y, 'A= ':A NEXT X	Y= 12 A= 3 Y= 4 A= 3 Y= 5 A= 3 Y= 7654 A= 3 Y= 0 A= 0

REPEAT statement

The REPEAT statement is a loop-controlling statement.

For syntax details, see the LOOP statement, on page 248.

REPLACE function

Use the `REPLACE` function to return a copy of a dynamic array with the specified field, value, or subvalue replaced with new data.

Syntax

```
REPLACE (expression, field#, value#, subvalue# { , | ; } replacement)
REPLACE (expression [,field# [,value#]] ; replacement)
variable < field# [ ,value# [,subvalue#]] >
```

expression specifies a dynamic array.

The expressions *field#*, *value#*, and *subvalue#* specify the type and position of the element to be replaced. These expressions are called delimiter expressions.

replacement specifies the value that the element is given.

The *value#* and *subvalue#* are optional. However, if either *subvalue#* or both *value#* and *subvalue#* are omitted, a semicolon (;) must precede *replacement*, as shown in the second syntax.

You can use angle brackets to replace data in dynamic arrays. Angle brackets to the left of an assignment operator change the specified data in the dynamic array according to the assignment operator. Angle brackets to the right of an assignment operator indicate that an `EXTRACT` function is to be performed (for examples, see the EXTRACT function, on page 153).

variable specifies the dynamic array containing the data to be changed.

The three possible results of delimiter expressions are described as case 1, case 2, and case 3.

Case	Description
Case 1:	Both *value#* and *subvalue#* are omitted or are specified as 0. A field is replaced by the value of *replacement*. ▪ If *field#* is positive and less than or equal to the number of fields in the dynamic array, the field specified by *field#* is replaced by the value of *replacement*. ▪ If *field#* is negative, a new field is created by appending a field mark and the value of *replacement* to the last field in the dynamic array. ▪ If *field#* is positive and greater than the number of fields in the dynamic array, a new field is created by appending the proper number of field marks, followed by the value of *replacement*; thus, the value of *field#* is the number of the new field.

Case	Description
Case 2:	*subvalue#* is omitted or is specified as 0, and *value#* is nonzero. A value in the specified field is replaced with the value of *replacement*. ▪ If *value#* is positive and less than or equal to the number of values in the field, the value specified by the *value#* is replaced by the value of *replacement*. ▪ If *value#* is negative, a new value is created by appending a value mark and the value of *replacement* to the last value in the field. ▪ If *value#* is positive and greater than the number of values in the field, a value is created by appending the proper number of value marks, followed by the value of *replacement*, to the last value in the field; thus, the value of *value#* is the number of the new value in the specified field.
Case 3:	*field#*, *value#*, and *subvalue#* are all specified and are nonzero. A subvalue in the specified value of the specified field is replaced with the value of *replacement*. ▪ If *subvalue#* is positive and less than or equal to the number of subvalues in the value, the subvalue specified by the *subvalue#* is replaced by the value of *replacement*. ▪ If *subvalue#* is negative, a new subvalue is created by appending a subvalue mark and the subvalue of *replacement* to the last subvalue in the value. ▪ If the *subvalue#* is positive and greater than the number of subvalues in the value, a new subvalue is created by appending the proper number of subvalue marks followed by the value of *replacement* to the last subvalue in the value; thus, the value of the expression *subvalue#* is the number of the new subvalue in the specified value.

In IDEAL, PICK, PIOPEN, and REALITY flavor accounts, if *replacement* is an empty string and an attempt is made to append the new element to the end of the dynamic array, field, or value, the dynamic array, field, or value is left unchanged; additional delimiters are not appended. Use the EXTRA.DELIM option of the $OPTIONS statement to make the `REPLACE` function append a delimiter to the dynamic array, field, or value.

If *replacement* is the null value, the stored representation of null (CHAR(128)) is inserted into *dynamic.array*. If *dynamic.array* evaluates to the null value, it remains unchanged by the replacement. If the REPLACE statement references a subelement of an element whose value is the null value, the dynamic array is unchanged.

INFORMATION and IN2 flavors

In INFORMATION and IN2 flavor accounts, if *expression* is an empty string and the new element is appended to the end of the dynamic array, the end of a field, or the end of a value, a delimiter is appended to the dynamic array, field, or value. Use the -EXTRA.DELIM option of the $OPTIONS statement to make the `REPLACE` function work as it does in IDEAL, PICK, and REALITY flavor accounts.

Examples

In the following examples a field mark is shown by F, a value mark is shown by V, and a subvalue mark is shown by S.

The first example replaces field 1 with # and sets Q to `#FAVBVDSEFDFFF`:

```
R=@FM:"A":@VM:"B":@VM:"D":@SM:"E":@FM:"D":@FM:@FM:"F"
Q=R
Q=REPLACE(Q,1;"#")
```

The next example replaces the first subvalue of the third value in field 2 with # and sets Q to FAVBV#SEFDFFF:

```
Q=R
 Q<2,3,1>="#"
```

The next example replaces field 4 with # and sets Q to FAVBVDSEFDF#FF:

```
Q=R
 Q=REPLACE(Q,4,0,0;"#")
```

The next example replaces the first value in fields 1 through 4 with # and sets Q to #F#VBVDSEF#F#FF:

```
Q=R
 FOR X=1 TO 4
 Q=REPLACE(Q,X,1,0;"#")
 NEXT
```

The next example appends a value mark and # to the last value in field 2 and sets Q to FAVBVDSEV#FDFFF:

```
Q=R
 Q=REPLACE(Q,2,-1;"#")
```

RETURN statement

Use the RETURN statement to terminate a subroutine and return control to the calling program or statement.

Syntax

RETURN [TO *statement.label*]

If the TO clause is not specified, the RETURN statement exits either an internal subroutine called by a GOSUB statement or an external subroutine called by a CALL statement. Control returns to the statement that immediately follows the CALL or GOSUB statement.

Use a RETURN statement to terminate an internal subroutine called with a GOSUB statement to ensure that the program proceeds in the proper sequence.

Use a RETURN statement or an END statement to terminate an external subroutine called with a CALL statement. When you exit an external subroutine called by CALL, all files opened by the subroutine are closed, except files that are open to common variables.

Use the TO clause to exit only an internal subroutine; control passes to the specified statement label. If you use the TO clause and *statement.label* does not exist, an error message appears when the program is compiled.

> **Note:** Using the TO clause can make program debugging and modification extremely difficult. Be careful when you use the RETURN TO statement, because all other GOSUBs or CALLs active at the time the GOSUB is executed remain active, and errors can result.

If the RETURN or RETURN TO statement does not have a place to return to, control is passed to the calling program or to the command language.

Example

In the following example, subroutine XYZ prints the message “THIS IS THE EXTERNAL SUBROUTINE” and returns to the main program:

```
20: GOSUB 80:
 25: PRINT "THIS LINE WILL NOT PRINT"
 30: PRINT "HI THERE"
 40: CALL XYZ


60: PRINT "BACK IN MAIN PROGRAM"
 70: STOP
 80: PRINT "THIS IS THE INTERNAL SUBROUTINE"
 90: RETURN TO 30:
 END
```

This is the program output:

```
THIS IS THE INTERNAL SUBROUTINE
 HI THERE
 THIS IS THE EXTERNAL SUBROUTINE
 BACK IN MAIN PROGRAM
```

RETURN (value) statement

Use the RETURN (*value*) statement to return a value from a user-written function.

expression evaluates to the value you want the user-written function to return. If you use a RETURN (*value*) statement in a user-written function and you do not specify *expression*, an empty string is returned by default.

You can use the RETURN (*value*) statement only in user-written functions. If you use one in a program or subroutine, an error message appears.

Syntax

RETURN (*expression*)

REUSE function

Use the `REUSE` function to specify that the value of the last field, value, or subvalue be reused in a dynamic array operation.

Syntax

REUSE (*expression*)

expression is either a dynamic array or an expression whose value is considered to be a dynamic array.

When a dynamic array operation processes two dynamic arrays in parallel, the operation is always performed on corresponding subvalues. This is true even for corresponding fields, each of which contains a single value. This single value is treated as the first and only subvalue in the first and only value in the field.

A dynamic array operation isolates the corresponding fields, values, and subvalues in a dynamic array. It then operates on them in the following order:

1. The subvalues in the values
2. The values in the fields
3. The fields of each dynamic array

A dynamic array operation without the REUSE function adds zeros or empty strings to the shorter array until the two arrays are equal. (The DIVS function, on page 134 is an exception. If a divisor element is absent, the divisor array is padded with ones, so that the dividend value is returned.)

The `REUSE` function reuses the last value in the shorter array until all elements in the longer array are exhausted or until the next higher delimiter is encountered.

After all subvalues in a pair of corresponding values are processed, the dynamic array operation isolates the next pair of corresponding values in the corresponding fields and repeats the procedure.

After all values in a pair of corresponding fields are processed, the dynamic array operation isolates the next pair of corresponding fields in the dynamic arrays and repeats the procedure.

If *expression* evaluates to the null value, the null value is replicated, and null is returned for each corresponding element.

Example

```
B = (1:@SM:6:@VM:10:@SM:11)
 A = ADDS(REUSE(5),B)
 PRINT "REUSE(5) + 1:@SM:6:@VM:10:@SM:11 = ": A
 *
 PRINT "REUSE(1:@SM:2) + REUSE(10:@VM:20:@SM:30) = ":
 PRINT ADDS(REUSE(1:@SM:2),REUSE(10:@VM:20:@SM:30))
 *
 PRINT "(4:@SM:7:@SM:8:@VM:10)*REUSE(10) = ":
 PRINT MULS((4:@SM:7:@SM:8:@VM:10 ),REUSE(10))
```

This is the program output:

```
REUSE(5) + 1:@SM:6:@VM:10:@SM:11 = 6S11V15S16
 REUSE(1:@SM:2) + REUSE(10:@VM:20:@SM:30) = 11S12V22S32
 (4:@SM:7:@SM:8:@VM:10)*REUSE(10) = 40S70S80V100
```

REVREMOVE statement

Use the REVREMOVE statement to successively extract dynamic array elements that are separated by system delimiters. The elements are extracted from right to left, in the opposite order from those extracted by the REMOVE statement. When a system delimiter is encountered, the extracted element is assigned to *element*.

Syntax

REVREMOVE *element* FROM *dynamic.array* SETTING *variable*

dynamic.array is an expression that evaluates to the dynamic array from which to extract elements.

variable is set to a code value corresponding to the system delimiter terminating the element just extracted. The delimiter code settings assigned to variable are as follows:

Value	Description
0	End of string
1	Item mark ASCII CHAR(255)

Value	Description
2	Field mark ASCII CHAR(254)
3	Value mark ASCII CHAR(253)
4	Subvalue mark ASCII CHAR(252)
5	Text mark ASCII CHAR(251)
6	ASCII CHAR(250)
7	ASCII CHAR(249)
8	ASCII CHAR(248)

The REVREMOVE statement extracts one element each time it is executed, beginning with the “remove pointer” of the *dynamic.array*. The operation can be repeated until all elements of *dynamic.array* are extracted. The REVREMOVE statement does not change the dynamic array.

As each element is extracted from *dynamic.array* to *element*, a pointer associated with *dynamic.array* moves back to the beginning of the element just extracted.

The pointer is reset to the beginning of *dynamic.array* whenever *dynamic.array* is reassigned. Therefore, *dynamic.array* should not be assigned a new value until all elements have been extracted (that is, until *variable* = 0).

If an element in *dynamic.array* is the null value, null is returned for that element.

Use REVREMOVE with the REMOVE statement. After a REMOVE statement, REVREMOVE returns the same string as the preceding REMOVE, setting the pointer to the delimiter preceding the extracted element. Thus, a subsequent REMOVE statement extracts the same element yet a third time.

If no REMOVE statement has been performed on *dynamic.array* or if the leftmost dynamic array element has been returned, *element* is set to the empty string and *variable* indicates end of string (that is, 0).

Example

```
DYN = "THIS":@FM:"HERE":@FM:"STRING"
 REMOVE VAR FROM DYN SETTING X
 PRINT VAR
 REVREMOVE NVAR FROM DYN SETTING X
 PRINT NVAR
 REMOVE CVAR FROM DYN SETTING X
 PRINT CVAR
```

The program output is:

```
THIS
 THIS
 THIS
```

REWIND statement

Use the REWIND statement to rewind a magnetic tape to the beginning-of-tape position.

The UNIT clause specifies the number of the tape drive unit. Tape unit 0 is used if no unit is specified. If the UNIT clause is used, *mtu* is an expression that evaluates to a code made up of three decimal digits. Although the *mtu* expression is a function of the UNIT clause, the REWIND statement uses only the third digit (the *u*). Its value must be in the range of 0 through 7. If *mtu* evaluates to the null value, the REWIND statement fails and the program terminates with a run-time error message.

Before a REWIND statement is executed, a tape drive unit must be attached to the user. Use the ASSIGN command to assign a tape unit to a user. If no tape unit is attached or if the unit specification is incorrect, the ELSE statements are executed.

The STATUS function returns 1 if REWIND takes the ELSE clause, otherwise it returns 0.

Syntax

```
REWIND [UNIT (mtu)]
        {THEN statements [ELSE statements] | ELSE statements}
```

PIOPEN flavor

If you have a program that specifies the syntax UNIT *ndmtu*, the *nd* elements are ignored by the compiler and no errors are reported.

Example

```
REWIND UNIT(002) ELSE PRINT "UNIT NOT ATTACHED"
```

RIGHT function

Use the RIGHT function to extract a substring comprising the last *n* characters of a string. It is equivalent to the following substring extraction operation:

string [*length*]

If you use this function, you need not calculate the string length.

If *string* evaluates to the null value, null is returned. If *n* evaluates to the null value, the RIGHT function fails and the program terminates with a run-time error message.

Syntax

```
RIGHT (string, n)
```

Example

```
PRINT RIGHT("ABCDEFGH",3)
```

This is the program output:

```
FGH
```

RND function

Use the RND function to generate any positive or negative random integer or 0.

expression evaluates to the total number of integers, including 0, from which the random number can be selected. That is, if *n* is the value of *expression*, the random number is generated from the numbers 0 through (*n* - 1).

If *expression* evaluates to a negative number, a random negative number is generated. If *expression* evaluates to 0, 0 is the random number. If *expression* evaluates to the null value, the RND function fails and the program terminates with a run-time error message.

See the RANDOMIZE statement, on page 302 for details on generating repeatable sequences of random numbers.

Syntax

RND (*expression*)

Example

```
A=20
 PRINT RND(A)
 PRINT RND(A)
 PRINT RND(A)
 PRINT RND(A)
```

This is the program output:

```
10
 3
 6
 10
```

ROLLBACK statement

Use the ROLLBACK statement to cancel all file I/O changes made during a transaction. The WORK keyword provides compatibility with SQL syntax conventions; it is ignored by the compiler.

A transaction includes all statements executed since the most recent BEGIN TRANSACTION statement. The ROLLBACK statement rolls back all changes made to files during the active transaction. If a subtransaction rolls back, none of the changes resulting from the active subtransaction affect the parent transaction. If the top-level transaction rolls back, none of the changes made are committed to disk, and the database remains unaffected by the transaction.

Use the ROLLBACK statement in a transaction without a COMMIT statement to review the results of a possible change. Doing so does not affect the parent transaction or the database. Executing a ROLLBACK statement ends the current transaction. After the transaction ends, execution continues with the statement following the next END TRANSACTION statement.

If no transaction is active, the ROLLBACK statement generates a run-time warning, and the ELSE statements are executed.

Syntax

ROLLBACK [WORK] [THEN *statements*] [ELSE *statements*]

Example

This example begins a transaction that applies locks to rec1 and rec2. If errors occur (such as a failed READU statement or a failed WRITE statements), the ROLLBACK statements ensure that no changes are written to the file.

```
BEGIN TRANSACTION
    READU data1 FROM file1,rec1 ELSE ROLLBACK
    READU data2 FROM file2,rec2 ELSE ROLLBACK
       .
       .
       .
```

```
    WRITE new.data1 ON file1,rec1 ELSE ROLLBACK
    WRITE new.data2 ON file2,rec2 ELSE ROLLBACK
    COMMIT WORK
END TRANSACTION
```

The update record lock on rec1 is not released on successful completion of the first WRITE statement.

RPC.CALL function

Use the `RPC.CALL` function to make requests of a connected server. The request is packaged and sent to the server using the C client RPC library. RPC.CALL returns the results of processing the remote request: 1 for success, 0 for failure.

Syntax

`RPC.CALL` (*`connection.ID, procedure, #args,`* `MAT` *`arg.list, #values,`* `MAT` *`return.list`*)

connection.ID is the handle of the open server connection on which to issue the RPC request. The RPC.CONNECT function, on page 334 gets the *connection.ID*.

procedure is a string identifying the operation requested of the server.

#args is the number of elements of *arg.list* to pass to the RPC server.

arg.list is a two-dimensional array (matrix) containing the input arguments to pass to the RPC server. The elements of this array represent ordered pairs of values. The first value is the number of the argument to the server operation, the second value is an argument-type declarator. (Data typing generalizes the RPC interface to work with servers that are data-type sensitive.)

#values is the number of values returned by the server.

return.list is a dimensioned array containing the results of the remote operation returned by `RPC.CALL`. Like *arg.list*, the results are ordered pairs of values.

`RPC.CALL` builds an RPC packet from *#args* and *arg.list*. Functions in the C client RPC library transmit the packet to the server and wait for the server to respond. When a response occurs, the RPC packet is separated into its elements and stored in the array *return.list*.

Use the `STATUS` function after an `RPC.CALL` function is executed to determine the result of the operation, as follows:

Value	Description
81001	Connection closed, reason unspecified.
81002	*connection.ID* does not correspond to a valid bound connection.
81004	Error occurred while trying to store an argument in the transmission packet.
81005	Procedure access denied because of a mismatch of RPC versions.
81008	Error occurred because of a bad parameter in *arg.list*.
81009	Unspecified RPC error.
81010	*#args* does not match expected argument count on remote machine.
81015	Timeout occurred while waiting for response from server.

Example

The following example looks for jobs owned by *fred*. The server connection was made using the RPC.CONNECT function.

```
args (1,1) = "fred"; args (1,2) = UNIRPC.STRING
 IF (RPC.CALL (server.handle, "COUNT.USERS", 1, MAT args,
       return.count, MAT res)) ELSE
    PRINT "COUNT.JOBS request failed, error code is: " STATUS()
    GOTO close.connection:
 END
```

RPC.CONNECT function

Use the RPC.CONNECT function to establish a connection to a server process. Once the host and server are identified, the local UVNet daemon tries to connect to the remote server. If the attempt succeeds, RPC.CONNECT returns a connection ID. If it fails, RPC.CONNECT returns 0. The connection ID is a nonzero integer used to refer to the server in subsequent calls to RPC.CALL function and RPC.DISCONNECT function.

Syntax

RPC.CONNECT (*host, server*)

> **Note:** Beginning at UniVerse 11.2, you must run SET.REMOTE.ID prior to using RPC.CONNECT. A client prior to UniVerse 11.1.14 will not be able to connect to UniVerse 11.1.14 or greater.

host is the name of the host where the server resides:

- UNIX This is defined in the local /etc/hosts file.
- Windows NT This is defined in the *system32*\drivers\etc\hosts file.

server is the name, as defined in the remote /etc/services file, of the RPC server class on the target host.

If *host* is not in the /etc/hosts file, or if *server* is not in the remote /etc/services file, the connection attempt fails.

Use the STATUS function after an RPC.CONNECT function is executed to determine the result of the operation, as follows:

Value	Description
81005	Connection failed because of a mismatch of RPC versions.
81007	Connection refused because the server cannot accept more clients.
81009	Unspecified RPC error.
81011	Host is not in the local /etc/hosts file.
81012	Remote unirpcd cannot start service because it could not fork the process.
81013	Cannot open the remote unirpcservices file.
81014	Service not found in the remote unirpcservices file.
81015	Connection attempt timed out.

Example

The following example connects to a remote server called MONITOR on HOST.A:

```
MAT args(1,2), res(1,2)
 server.handle = RPC.CONNECT ("HOST.A", "MONITOR")
 IF (server.handle = 0) THEN
    PRINT "Connection failed, error code is: ": STATUS()
    STOP
 END
```

RPC.DISCONNECT function

Use the `RPC.DISCONNECT` function to end an RPC session.

Syntax

`RPC.DISCONNECT` (*`connection.ID`*)

connection.ID is the RPC server connection you want to close.

`RPC.DISCONNECT` sends a request to end a connection to the server identified by *connection.ID*. When the server gets the request to disconnect, it performs any required termination processing. If the call is successful, `RPC.DISCONNECT` returns 1. If an error occurs, `RPC.DISCONNECT` returns 0.

Use the `STATUS` function after an RPC.DISCONNECT function is executed to determine the result of the operation, as follows:

Value	Description
81001	The connection was closed, reason unspecified.
81002	*connection.ID* does not correspond to a valid bound connection.
81009	Unspecified RPC error.

Example

The following example closes the connection to a remote server called MONITOR on HOST.A:

```
MAT args(1,2), res(1,2)
 server.handle = RPC.CONNECT ("HOST.A", "MONITOR")
 IF (server.handle = 0) THEN
    PRINT "Connection failed, error code is: ": STATUS()
    STOP
 END
    .
    .
    .
 close.connection:
 IF (RPC.DISCONNECT (server.handle)) ELSE
    PRINT "Bizarre disconnect error, result code is: " STATUS()
 END
```

saveSecurityContext function

The `saveSecurityContext`() function encrypts and saves a security context to a system security file. The file is maintained on a per account basis for UniData and UniVerse. The name is used as the

record ID to access the saved security information. Since the information is encrypted, you should not attempt to directly manipulate it.

You might want your application to save a security context to be used later. Multiple contexts can be created to suit different needs. For example, you might want to use different protocols to talk to different servers. These contexts can be saved and reused.

When creating a saved context, you must provide both a *name* and a *passPhrase* to be used to encrypt the contents of the context. The same *name* and *passPhrase* must be provided to load the saved context back. To ensure a high level of security, we recommend that the *passPhrase* be relatively long, yet easy to remember.

Syntax

saveSecurityContext(*context, name, passPhrase*)

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
context	The security context handle.
name	String containing the name of the saved context.
passPhrase	String containing the password to encrypt the context contents.

Return code status

The following table describes the status of each return code.

Return code	Status
0	Success.
1	Invalid security context handle.
2	Invalid parameters (empty name or passPhrase).
3	Context could not be saved.

SADD function

Use the SADD function to add two string numbers and return the result as a string number. You can use this function in any expression where a string or string number is valid, but not necessarily where a standard number is valid, because string numbers can exceed the range of numbers that standard arithmetic operators can handle.

Either string number can evaluate to any valid number or string number.

If either string number contains nonnumeric data, an error message is generated, and 0 replaces the nonnumeric data. If either string number evaluates to the null value, null is returned.

Syntax

SADD (*string.number.1, string.number.2*)

Example

```
A = 8888888888888888
 B = 77777777777777777
```

```
X = "8888888888888888"
Y = "7777777777777777"
PRINT A + B
PRINT SADD(X,Y)
```

This is the program output:

```
166666666666667000
 166666666666666665
```

SCMP function

Use the SCMP function to compare two string numbers and return one of the following three numbers: -1 (less than), 0 (equal), or 1 (greater than). If *string.number.1* is less than *string.number.2*, the result is -1. If they are equal, the result is 0. If *string.number.1* is greater than *string.number.2*, the result is 1. You can use this function in any expression where a string or string number is valid.

Either string number can be a valid number or string number. Computation is faster with string numbers.

If either string number contains nonnumeric data, an error message is generated and 0 is used instead of the nonnumeric data. If either string number evaluates to the empty string, null is returned.

Syntax

SCMP (*string.number.1, string.number.2*)

Example

```
X = "123456789"
 Y = "123456789"
 IF SCMP(X,Y) = 0 THEN PRINT "X is equal to Y"
    ELSE PRINT "X is not equal to Y"
 END
```

This is the program output:

X is equal to Y

SDIV function

Use the SDIV function to divide *string.number.1* by *string.number.2* and return the result as a string number. You can use this function in any expression where a string or string number is valid, but not necessarily where a standard number is valid, because string numbers can exceed the range of numbers which standard arithmetic operators can handle. Either string number can be a valid number or a string number.

precision specifies the number of places to the right of the decimal point. The default precision is 14.

If either string number contains nonnumeric data, an error message is generated and 0 is used for that number. If either string number evaluates to the null value, null is returned.

Syntax

SDIV (*string.number.1, string.number.2* [,*precision*])

Example

```
X = "1"
 Y = "3"
 Z = SDIV (X,Y)
 ZZ = SDIV (X,Y,20)
 PRINT Z
 PRINT ZZ
```

This is the program output:

```
0.33333333333333
 0.33333333333333333333
```

SEEK statement

Use the SEEK statement to move the file pointer by an offset specified in bytes, relative to the current position, the beginning of the file, or the end of the file.

file.variable specifies a file previously opened for sequential access.

offset is the number of bytes before or after the reference position. A negative offset results in the pointer being moved before the position specified by *relto*. If *offset* is not specified, 0 is assumed.

Note: On Windows NT systems, line endings in files are denoted by the character sequence RETURN + LINEFEED rather than the single LINEFEED used in UNIX files. The value of offset should take into account this extra byte on each line in Windows NT file systems.

The permissible values of *relto* and their meanings follow:

Value	Description
0	Relative to the beginning of the file
1	Relative to the current position
2	Relative to the end of the file

If *relto* is not specified, 0 is assumed.

If the pointer is moved, the THEN statements are executed and the ELSE statements are ignored. If the THEN statements are not specified, program execution continues with the next statement.

If the file cannot be accessed or does not exist, the ELSE statements are executed; any THEN statements are ignored.

If *file.variable*, *offset*, or *relto* evaluates to the null value, the SEEK statement fails and the program terminates with a run-time error message.

Note: On Windows NT systems, if you use the OPENDEV statement to open a 1/4-inch cartridge tape (60 MB or 150 MB) for sequential processing, you can move the file pointer only to the beginning or the end of the data. For diskette drives, you can move the file pointer only to the start of the data.

Seeking beyond the end of the file and then writing creates a gap, or hole, in the file. This hole occupies no physical space, and reads from this part of the file return as ASCII CHAR 0 (neither the number nor the character 0).

For more information about sequential file processing, see the OPENSEQ statement, on page 283, READSEQ statement, on page 310, and WRITESEQ statement, on page 454.

Syntax

```
SEEK file.variable [, offset [, relto]]
        {THEN statements [ELSE statements] | ELSE statements}
```

Example

The following example reads and prints the first line of RECORD4. Then the SEEK statement moves the pointer five bytes from the front of the file, then reads and prints the rest of the current line.

```
OPENSEQ 'FILE.E', 'RECORD4' TO FILE ELSE ABORT
 READSEQ B FROM FILE THEN PRINT B
 SEEK FILE,5, 0 THEN READSEQ A FROM FILE
 THEN PRINT A ELSE ABORT
```

This is the program output:

```
FIRST LINE
 LINE
```

SEEK(ARG.) statement

Use the SEEK(ARG.) statement to move the command line argument pointer to the next command line argument from left to right, or to a command line argument specified by *arg#*. The command line is delimited by blanks, and the first argument is assumed to be the first word after the program name. When a cataloged program is invoked, the argument list starts with the second word in the command line.

Syntax

```
SEEK(ARG. [,arg#] ) [THEN statements] [ELSE statements]
```

Blanks in quoted strings are not treated as delimiters. A quoted string is treated as a single argument.

arg# specifies the command line argument to move to. It must evaluate to a number. If *arg#* is not specified , the pointer moves to the next command line argument. SEEK(ARG.) works similarly to GET(ARG.) statement except that SEEK(ARG.) makes no assignments.

THEN and ELSE statements are both optional. The THEN clause is executed if the argument is found. The ELSE clause is executed if the argument is not found. The SEEK(ARG.) statement fails if *arg#* evaluates to a number greater than the number of command line arguments or if the last argument has been assigned and a SEEK(ARG.) with no *arg#* is used. To move to the beginning of the argument list, set *arg#* to 1.

If *arg#* evaluates to the null value, the SEEK(ARG.) statement fails and the program terminates with a run-time error message.

Example

If the command line is:

```
RUN BP PROG ARG1 ARG2 ARG3
```

and the program is:

```
A=5;B=2
 SEEK(ARG.)
 SEEK(ARG.,B)
 SEEK(ARG.)
 SEEK(ARG.,A-B)
 SEEK(ARG.,1)
```

the system pointer moves as follows:

```
ARG2
 ARG2
 ARG3
 ARG3
 ARG1
```

SELECT statements

Use a SELECT statement to create a numbered select list of record IDs from a UniVerse file or a dynamic array. A subsequent READNEXT statement can access this select list, removing one record ID at a time from the list. READNEXT instructions can begin processing the select list immediately.

Syntax

SELECT [*variable*] [TO *list.number*] [ON ERROR *statements*]

SELECTN [*variable*] [TO *list.number*] [ON ERROR *statements*]

SELECTV [*variable*] TO *list.variable* [ON ERROR *statements*]

variable can specify a dynamic array or a file variable. If it specifies a dynamic array, the record IDs must be separated by field marks (ASCII 254). If *variable* specifies a file variable, the file variable must have previously been opened. If *variable* is not specified, the default file is assumed (for more information on default files, see the OPEN statement, on page 276). If the file is neither accessible nor open, or if *variable* evaluates to the null value, the SELECT statement fails and the program terminates with a run-time error message.

If the file is an SQL table, the effective user of the program must have SQL SELECT privilege to read records in the file. For information about the effective user of a program, see the AUTHORIZATION statement, on page 71.

You must use a file lock with the SELECT statement when it is within a transaction running at isolation level 4 (serializable). This prevents phantom reads.

The TO clause specifies the select list that is to be used. *list.number* is an integer from 0 through 10. If no *list.number* is specified, select list 0 is used.

The record IDs of all the records in the file, in their stored order, form the list. Each record ID is one entry in the list.

The SELECT statement does not process the entire file at once. It selects record IDs group by group. The @SELECTED variable is set to the number of elements in the group currently being processed.

You often want a select list with the record IDs in an order different from their stored order or with a subset of the record IDs selected by some specific criteria. To do this, use the SELECT or SSELECT commands in a BASIC EXECUTE statement. Processing the list by READNEXT is the same, regardless of how the list is created.

Use the SELECTV statement to store the select list in a named list variable instead of to a numbered select list. *list.variable* is an expression that evaluates to a valid variable name. This is the default behavior of the SELECT statement in PICK, REALITY, and IN2 flavor accounts. You can also use the VAR.SELECT option of the $OPTIONS statement to make the SELECT statement act as it does in PICK, REALITY, and IN2 flavor accounts.

The ON ERROR clause

The ON ERROR clause is optional in the SELECT statement. The ON ERROR clause lets you specify an alternative for program termination when a fatal error is encountered during processing of the SELECT statement.

If a fatal error occurs, and the ON ERROR clause was not specified, or was ignored (as in the case of an active transaction), the following occurs:

- An error message appears.
- Any uncommitted transactions begun within the current execution environment roll back.
- The current program terminates.
- Processing continues with the next statement of the previous execution environment, or the program returns to the UniVerse prompt.

A fatal error can occur if any of the following occur:

- A file is not open.
- *file.variable* is the null value.
- A distributed file contains a part file that cannot be accessed.

If the ON ERROR clause is used, the value returned by the STATUS statement is the error number.

PICK, REALITY, and IN2 flavors

In a PICK, REALITY, or IN2 flavor account, the SELECT statement has the following syntax:

```
SELECT [V][variable] TO list.variable

SELECTN [variable] TO list.number
```

You can use either the SELECT or the SELECTV statement to create a select list and store it in a named list variable. The only useful thing you can do with a list variable is use a READNEXT statement to read the next element of the select list.

Use the SELECTN statement to store the select list in a numbered select list. *list.number* is an expression that evaluates to a number from 0 through 10. You can also use the -VAR.SELECT option of the $OPTIONS statement to make the SELECT statement act as it does in IDEAL and INFORMATION flavor accounts.

Example

The following example opens the file SUN.MEMBER to the file variable MEMBER.F, then creates an active select list of record IDs. The READNEXT statement assigns the first record ID in the select list to the variable @ID, then prints it. Next, the file SUN.SPORT is opened to the file variable SPORT.F, and a select list of its record IDs is stored as select list 1. The READNEXT statement assigns the first record ID in the select list to the variable A, then prints DONE.

```
OPEN '','SUN.MEMBER' TO MEMBER.F ELSE PRINT "NOT OPEN"
 SELECT
 READNEXT @ID THEN PRINT @ID
 *
 OPEN '','SUN.SPORT' TO SPORT.F ELSE PRINT "NOT OPEN"
```

```
SELECT TO 1
READNEXT A FROM 1 THEN PRINT "DONE" ELSE PRINT "NOT"
```

This is the program output:

```
4108
 DONE
```

SELECTE statement

Use the SELECTE statement to assign the contents of select list 0 to *list.variable*. *list.variable* is activated in place of select list 0 and can be read with the READNEXT statement.

Syntax

SELECTE TO *list.variable*

SELECTINDEX statement

Use the SELECTINDEX statement to create select lists from secondary indexes.

Syntax

SELECTINDEX *index* [, *alt.key*] FROM *file.variable* [TO *list.number*]

index is an expression that evaluates to the name of an indexed field in *file.variable*. *index* must be the name of the field that was used in the CREATE.INDEX command when the index was built.

alt.key is an expression that evaluates to a secondary index key. If *alt.key* is specified, a select list is created of the record IDs referenced by *alt.key*. If *alt.key* is not specified, a select list is created of the record IDs referenced by all of the index's keys.

file.variable specifies an open file.

list.number is an expression that evaluates to the select list number. It can be a number from 0 through 10. The default list number is 0.

Note: In PICK, REALITY, and IN2 flavors, *list.number* is a variable rather than a list number.

Note: If *index* is multivalued, each value is indexed even if the field contains duplicate values in the same record. Except in PIOPEN flavor accounts, such duplicate values are returned to *list.number*. To prevent the return of duplicate key values, use the PIOPEN.SELIDX option of the $OPTIONS statement.

If the field is not indexed, the select list is empty, and the value of the STATUS function is 1; otherwise the STATUS function is 0. If *index*, *alt.key*, or *file.variable* evaluates to the null value, the SELECTINDEX statement fails and the program terminates with a run-time error message.

PIOPEN flavor

In a PIOPEN flavor account, the SELECTINDEX statement eliminates duplicate key values when it creates a select list from *index*. To do this in other flavors, use the PIOPEN.SELIDX option of the $OPTIONS statement.

Example

In the following example, the first SELECTINDEX selects all data values to list 1. The second SELECTINDEX selects record IDs referenced by STOREDVAL to list 2.

```
OPEN "", "DB" TO FV ELSE STOP "OPEN FAILED"
 SELECTINDEX "F1" FROM FV TO 1
 EOV = 0
 LOOP
    SELECTINDEX "F1" FROM FV TO 1

UNTIL EOV DO
    SELECTINDEX "F1", STOREDVAL FROM FV TO 2
    EOK = 0
    LOOP
       READNEXT KEY FROM 2 ELSE EOK=1
    UNTIL EOK DO
       PRINT "KEY IS ":KEY:" STOREDVAL IS ":STOREDVAL
    REPEAT
 REPEAT
 END
```

SELECTINFO function

Use the SELECTINFO function to determine whether a select list is active, or to determine the number of items it contains.

list is an expression evaluating to the number of the select list for which you require information. The select list number must be in the range of 0 through 10.

key specifies the type of information you require. You can use equate names for the keys as follows:

Syntax

SELECTINFO (*list, key*)

Key	Description
IK$SLACTIVE	Returns 1 if the select list specified is active, and returns 0 if the select list specified is not active.
IK$SLCOUNT	Returns the number of items in the select list. 0 is returned if the select list is not active or is an empty select list.

Equate names

An insert file of equate names is provided for the SELECTINFO keys. To use the equate names, specify the directive $INCLUDE UNIVERSE.INCLUDE INFO_KEYS.INS.IBAS when you compile your program.

Example

In the following example, the insert file containing the equate name is inserted by the $INCLUDE statement. The conditional statement tests if select list 0 is active.

```
$INCLUDE SYSCOM INFO_KEYS.INS.IBAS
    IF SELECTINFO(0,IK$SLACTIVE)
                   THEN PRINT 'SELECT LIST ACTIVE'
                   ELSE PRINT 'SELECT LIST NOT ACTIVE'
```

```
END
```

SEND statement

Use the SEND statement to write a block of data to a device. The SEND statement can be used to write data to a device that has been opened for I/O using the OPENDEV statement or OPENSEQ statement.

Syntax

```
SEND output [:] TO device
      { THEN statements [ELSE statements] | ELSE statements }
```

output is an expression evaluating to a data string that will be written to *device*. If the optional colon is used after *output*, the terminating newline is not generated.

device is a valid file variable resulting from a successful OPENDEV or OPENSEQ statement. This is the handle to the I/O device that supplies the data stream for the operation of the SEND statement.

The SEND syntax requires that either a THEN or an ELSE clause, or both, be specified. If data is successfully sent, the SEND statement executes the THEN clause. If data cannot be sent, it executes the ELSE clause.

The data block specified by *output* is written to the device followed by a newline. Upon successful completion of the SEND operation, program control is passed to the THEN clause if specified. If an error occurs during the SEND operation, program control is passed to the ELSE clause if specified.

Example

The following code fragment shows how the SEND statement is used to write a series of messages on a connected device:

```
OPENDEV "TTY10" TO TTYLINE ELSE STOP "CANNOT OPEN TTY10"
 LOOP
       INPUT MESSAGE
    WHILE MESSAGE # "QUIT" DO
       SEND MESSAGE TO TTYLINE
       ELSE
          STOP "ERROR WRITING DATA TO TTY10"
       END
 REPEAT
```

SENTENCE function

Use the SENTENCE function to return the stored sentence that invoked the current process. Although the SENTENCE function uses no arguments, parentheses are required to identify it as a function. The SENTENCE function is a synonym for the @SENTENCE system variable.

A PERFORM statement in a program updates the system variable, @SENTENCE, with the command specified in the PERFORM statement.

Syntax

```
SENTENCE ()
```

Example

```
PRINT SENTENCE()
```

This is the program output:

```
RUN BP TESTPROGRAM
```

SEQ function

Use the SEQ function to convert an ASCII character to its numeric string equivalent.

Syntax

SEQ (*expression*)

expression evaluates to the ASCII character to be converted. If *expression* evaluates to the null value, null is returned.

The SEQ function is the inverse of the CHAR function, on page 87.

In NLS mode, use the UNISEQ function to return Unicode values in the range x0080 through x00F8.

Using the SEQ function to convert a character outside its range results in a run-time message, and the return of an empty string.

For more information about these ranges, see the *UniVerse NLS Guide*.

PICK, IN2, and REALITY flavors

In PICK, IN2, and REALITY flavors SEQ(" ") is 255 instead of 0. In IDEAL and INFORMATION flavor accounts, use the SEQ.255 option of the $OPTIONS statement to cause SEQ(" ") to be interpreted as 255.

Example

```
G="T"
 A=SEQ(G)
 PRINT A, A+1
 PRINT SEQ("G")
```

This is the program output:

```
84          85
 71
```

SEQS function

Use the SEQS function to convert a dynamic array of ASCII characters to their numeric string equivalents.

Syntax

SEQS (*dynamic.array*)

CALL -**SEQS** (*return.array, dynamic.array*)

```
CALL !SEQS (return.array, dynamic.array)
```

dynamic.array specifies the ASCII characters to be converted. If *dynamic.array* evaluates to the null value, null is returned. If any element of *dynamic.array* is the null value, null is returned for that element.

If you use the subroutine syntax, the resulting dynamic array is returned as *return.array*.

In NLS mode, you can use the UNISEQS function to return Unicode values in the range x0080 through x00F8.

Using the SEQS function to convert a character outside its range results in a run-time message, and the return of an empty string.

For more information about these ranges, see the *UniVerse NLS Guide*.

Example

```
G="T":@VM:"G"
 A=SEQS(G)
 PRINT A
 PRINT SEQS("G")
```

This is the program output:

```
84V71
 71
```

setAuthenticationDepth function

The setAuthenticationDepth() function sets how deeply UniData and UniVerse should verify before deciding that a certificate is not valid.

Syntax

setAuthenticationDepth(*context, depth, ServerOrClient*)

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
context	The security context handle.
depth	Numeric value for verification depth.
ServerOrClient	Flag: 1- Server (SSL_SERVER) 2- Client (SSL_CLIENT) Any other value is treated as a value of 1.

Return code status

The following table describes the status of each return code.

Return code	Status
0	Success.
1	Invalid security context handle.
2	Invalid depth (must be greater than or equal to 0).
3	Invalid value for *ServerOrClient* (must be 1 or 2)

This function can be used to set both server authentication and client certification, determined by the value in parameter *ServerOrClient*. The default depth for both is 1.

The *depth* is the maximum number of intermediate issuer certificate, or CA certificates which must be examined while verifying an incoming certificate. Specifically, a depth of 0 means that the certificate must be self-signed. A depth of 1 means that the incoming certificate can be either self-signed, or signed by a CA which is known to the *context*.

You should set this value according to your organization's Public Key Infrastructure setup. Usually it should not be more than 5, but it should be large enough to allow the whole certificate chain to be examined.

setCipherSuite function

The `setCipherSuite`() function allows you to identify which cipher suites should be supported for the specified context. It affects the cipher suites and public key algorithms supported during the SSL/TLS handshake and subsequent data exchanges.

When a context is created, its cipher suites will all be set to SSLv3 suites by default.

Syntax

`setCipherSuite``(`*`context,cipherSpecs`*`)`

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
context	The security context handle.
CipherSpecs	String containing cipher suite specification described above.

Return code status

The following table describes the status of each return code.

Return code	Status
0	Success.
1	Invalid security context handle.
2	Invalid cipher suite specification.

The *CipherSpecs* parameter is a string containing *cipher-spec* separated by colons. An SSL cipher specification in *cipher-spec* is composed of 4 major attributes as well as several, less significant attributes. These are defined below.

Some of this information on ciphers is excerpted from the mod_ssl open source package of the Apache web server.

- Key Exchange Algorithm - RSA or Diffie-Hellman variants.
- Authentication Algorithm - RSA, Diffie-Hellman, DSS or none.
- Cipher/Encryption Algorithm - AES, DES, Triple-DES, RC4, RC2 or none.
- MAC Digest Algorithm - MD5, SHA, SHA1, or the SHA2 family.

An SSL cipher can also be an export cipher and is either an SSLv2 or SSLv3/TLSv1 cipher (here TLSv1 is equivalent to SSLv3). To specify which ciphers to use, one can either specify all the ciphers, one at a time, or use aliases to specify the preference and order for the ciphers.

The following table describes each tag for the Key Exchange Algorithm.

Tag	Description
KRSA	RSA key exchange
kDHr	Diffie-Hellman key exchange with RSA key
kDHd	Diffie-Hellman key exchange with DSA key
kEDH	Ephemeral (temp.key) Diffie-Hellman key exchange (no cert)

The following table describes each tag for the Authentication Algorithm.

Tag	Description
aNULL	No authentication
aRSA	RSA authentication
aDSS	DSS authentication
aDH	Diffie-Hellman authentication

The following table describes each tag for the Cipher Encoding Algorithm.

Tag	Description
eNULL	No encoding
DES	DES encoding
3DES	Triple-DES encoding
RC4	RC4 encoding
RC2	RC2 encoding
AES	AES encoding

The following table describes each tag for the MAC Digest Algorithm.

Tag	Description
MD5	MD5 hash function
SHA2	SHA2 family of hash functions
SHA1	SHA1 hash function
SHA	SHA hash function

The following table describes each of the Aliases.

Alias	Description
SSLv2	All SSL version 2.0 ciphers
SSLv3	All SSL version 3.0 ciphers
TLSv1	All TLS version 1.0 ciphers

Alias	Description
EXP	All export ciphers
EXPORT40	All 40-bit export ciphers only
EXPORT56	All 56-bit export ciphers only
LOW	All low strength ciphers (no export, single DES)
MEDIUM	All ciphers with 128 bit encryption
HIGH	All ciphers using Triple-DES
RSA	All ciphers using RSA key exchange
DH	All ciphers using Diffie-Hellman key exchange
EDH	All ciphers using Ephemeral Diffie-Hellman key exchange
ADH	All ciphers using Anonymous Diffie-Hellman key exchange
DSS	All ciphers using DSS authentication
NULL	All cipher using no encryption

These can be put together to specify the order and ciphers you wish to use. To speed this up there are also aliases (SSLv2, SSLv3, TLSv1, EXP, LOW, MEDIUM, HIGH) for certain groups of ciphers. These tags can be joined together with prefixes to form the *cipher-spec*.

The following table describes the available prefixes.

Tag	Description
none	Add cipher to the list
+	Add ciphers to the list and pull them to the current location in the list
-	Remove the cipher from the list (it can be added again later)
!	Kill the cipher from the list completely (cannot be added again later)

A more practical way of looking at all of this is to use the `getCipherSuite`() function, which provides a nice way to successively create the correct *cipher-spec* string. The default setup for a *cipher-spec* string is shown in the following example:

```
"ALL:!ADH=RC4+RSA:+HIGH:+MEDIUM:+LOW:SSLV2:+EXP"
```

As shown in the example, you must first remove from consideration any ciphers that do not authenticate, for example, for SSL only the Anonymous Diffie-Hellman ciphers. Next, use ciphers using RC4 and RSA. Next include the high, medium, and then the low security ciphers. Finally pull all SSLv2 and export the ciphers to the end of the list.

```
    Example:
        SetCipherSuite(ctxHandle,"RSA:!EXP:!NULL:+HIGH:+MEDIUM:-LOW")
SetCipherSuite(ctxHandle,"SSLv3")
```

To see a full list of the available ciphers, open a command prompt and navigate to the UniData or UniVerse `bin` directory. Enter the following command:

```
openssl ciphers -v
```

setClientAuthentication function

The `setClientAuthentication`() function turns on or off client authentication for a server socket.

When *option* is set to on, during the initial SSL handshake, the server sends a client authentication request to the client. It also receives the client certificate and performs authentication according to the issuer's certificate (or certificate chain) set in the security context.

Syntax

setClientAuthentication(*context*,*option*)

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
context	The security context handle.
option	1 - ON (SSL_CLIENT_AUTH) 2 - OFF (SSL_NO_CLIENT_AUTH)

Return code status

The following table describes the status of each return code.

Return code	Status
0	Success.
1	Invalid security context handle.

setIpv

Use the `setIpv` function to set the default IPv connection for the whole system or for only Socket networks or UVNet. The function also returns the setting back for display.

Syntax

setIpv (*ipvexpr*[,*networkexpr*])

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
ipvexpr	The IPv expression. Enter one of the following values: IPV4 IPV6 IPVANY IPV4_IPV6 IPV6_IPV4
networkexpr	The optional network expression. If you want the `setIpv` function to only set connections for Socket or UVNet networks, use the "SOCKET" or "UVNET" flags, respectively.

setPrivateKey function

The `setPrivateKey`() function loads the private key into a security context so that it can be used by SSL functions. If the context already had a set private key, it will be replaced.

Syntax

`setPrivateKey`(*`key, format, keyLoc, passPhrase, validate, context, p12pass`*)

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
Key	A string containing either the key or path for a key file.
format	1 - PEM (Base64 encoded) format (SSL_FMT_PEM) 2 - DER (ASN.1 binary) format (SSL_FMT_DER) 3 - PKCS #12 format (SSL_FMT_P12)
KeyLoc	1 - key contained in key string (SSL_LOC_STRING) 2 - key is in a file specified by key (SSL_LOC_FILE)
passPhrase	String containing the *passPhrase* required for gaining access to the key. It can be empty if the key is not pass-phrase protected. **Warning:** This method is not recommended.
Validate	1 - Validate against matching public key (SSL_VALIDATE) 0 - Won't bother to validate (SSL_NO_VALIDATE)
context	The security context handle.
p12pass	Optional. Sets a password on the PKCS #12 file. This parameter should only be included if using a PKCS #12 certificate that has a password. Otherwise the parameter should be omitted.

Return code status

The following table describes the status of each return code.

Return code	Status
0	Success.
1	Invalid security context handle.
2	Invalid format
3	Invalid key type
4	Key file cannot be accessed (non-existent or wrong pass phrase)
5	Certificate cannot be accessed
6	Private key does not match public key in certificate
7	Private key cannot be interpreted
99	Other errors that prevent private key from being accepted by UniData or UniVerse.

SSL depends on public-key crypto algorithms to perform its functions. A pair of keys is needed for each communicating party to transfer data over SSL. The public key is usually contained in a certificate, signed by a CA, while the private key is kept secretly by the user.

A private key is used to digitally sign a message or encrypt a symmetric secret key to be used for data encryption.

The *Key* parameter contains either the key string itself or a path that specifies a file that contains the key. UniData and UniVerse only accept PKCS #8 style private keys.

The *Format* parameter specifies if the key is in binary format or Base64 encoded format. If the key is in a file, Base64 format also means that it must be in PEM format.

The *KeyLoc* parameter specifies if the key is provided in a file or in a dynamic array string.

If the key is previously encrypted, a correct passPhrase must be given to decrypt the key first. It is recommended that the private key be always in encrypted form. Note that if the private key is generated by the `generateKey`() function described in the generateKey function, on page 182, then it is always in PEM format and always encrypted by a pass phrase.

If the *validate* parameter is set, the private key is verified with the public key contained in the certificate specified for either the server or client. They must match for SSL to work. In some cases there is no need or it is impossible to check against a certificate. For example, the certificate might already be distributed to the other end and there is no need for a user application to authenticate itself. In that case, *validate* can be set to 0 (SSL_NO_VALIDATE).

If *validate* is required, the corresponding certificate should be added first by calling the `addCertificate`() function.

The direct form of this function might be preferred by some applications where a hard coded private key can be incorporated into the application, eliminating the need to access an external key file, which might be considered a security hazard.

Note: The private key is the single most important piece of secret information for a public-key-based crypto system. You must take every precaution to keep it secure. If the private key is compromised, there will be no data security. This is especially true for server private keys.

setRandomSeed function

The `setRandomSeed`() function generates a random seed file from a series of source files and sets that file as the default seed file for the supplied security context.

Syntax

`setRandomSeed``(`*`inFiles, outFile, length, context`*`)`

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
inFiles	A string containing source file names.
outFile	A string containing the generated seed file.
length	The number of bytes that should be generated (the default is 1024 if less than 1024 is specified).
context	The security context handle.

Return code status

The following table describes the status of each return code.

Return code	Status
0	Success.
1	Invalid parameter(s).
2	Random file generation error.
3	Random file set error.

The strength of cryptographic functions depends on the true randomness of the keys. This function generates and sets the random seed file used by many of the UniData and UniVerse cryptographic functions. By default, UniData and UniVerse use the `.rnd` file in your UniData or UniVerse application's current UDTHOME or UVHOME directory. You can override the default by calling this function.

Note: Your application on a U2 server might be running under a system directory such as `C:\WINDOWS\system32` or `/usr/ud82` (UniData) or `/usr/uv112` (UniVerse), which might not allow the file to be created. To avoid this situation, you should always specify a location that allows random files to be created.

The random seed file is specified by *outFile*, which is generated based on source files specified in *inFiles*. For Windows platforms, multiple files must be separated by a semi-colon (;). For UNIX platforms, multiple files must be separated by a colon (:).

The *length* parameter specifies how many bytes of seed data should be generated.

If no source is specified in the *inFiles* parameter, then the *outFile* parameter must already exist.

If context is not specified, the seed file will be used as a global seed file that applies to all cryptographic functions. However, a seed file setting in a particular security context will always override the global setting.

SET TRANSACTION ISOLATION LEVEL statement

Use the SET TRANSACTION ISOLATION LEVEL statement to set the default transaction isolation level you need for your program.

Syntax

SET TRANSACTION ISOLATION LEVEL *level*

Note: The isolation level you set with this statement remains in effect until another such statement is issued. This affects all activities in the session, including UniVerse commands and SQL transactions.

The SET TRANSACTION ISOLATION LEVEL statement cannot be executed while a transaction exists. Attempting to do so results in a run-time error message, program failure, and the rolling back of all uncommitted transactions started in the execution environment.

level has the following syntax:

{n | keyword | expression}

level is an expression that evaluates to 0 through 4, or one of the following keywords:

Integer	Keyword	Effect on This Transaction
0	NO.ISOLATION	Prevents lost updates. Lost updates are prevented if the ISOMODE configurable parameter is set to 1 or 2.
1	READ.UNCOMMITTED	Prevents lost updates.
2	READ.COMMITTED	Prevents lost updates and dirty reads.
3	REPEATABLE.READ	Prevents lost updates, dirty reads, and nonrepeatable reads.
4	SERIALIZABLE	Prevents lost updates, dirty reads, nonrepeatable reads, and phantom writes.

Examples

The following example sets the default isolation level to 3 then starts a transaction at isolation level 4. The isolation level is reset to 3 after the transaction finishes.

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE.READ
 PRINT "We are at isolation level 3."
 BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE
          PRINT "We are at isolation level 4."
          COMMIT WORK
 END TRANSACTION
 PRINT "We are at isolation level 3"
```

The next example uses an expression to set the transaction level:

```
PRINT "Enter desired transaction isolation level:":
 INPUT TL
    SET TRANSACTION LEVEL TL
    BEGIN TRANSACTION
       .
       .
       .
    END TRANSACTION
```

setHTTPDefault function

The `setHTTPDefault` function configures the default HTTP settings, including proxy server and port, buffer size, authentication credential, HTTP version, and request header values. These settings are used with every HTTP request that follows.

Syntax

`setHTTPDefault`(*`option, value`*)

If you require all outgoing network traffic to go through a proxy server, you should call `setHTTPDefault`() with values containing the proxy server name or IP address, as well as the port (if other than the default of 80).

option is a string containing an option name. See the table below for the options currently defined.

value is a string containing the appropriate option value.

The following table describes the available options for `setHTTPDefault`.

Option	Description
PROXY_NAME	Name or IP address of the proxy server.
PROXY_PORT	The port number to be used on the proxy server. This only needs to be specified if the port is other than the default of 80.
VERSION	The version of HTTP to be used. The default version is 1.0, but it can be set to 1.1 for web servers that understand the newer protocol. The string should be “1.0” or “1.1.”
BUFSIZE	The size of the buffer for HTTP data transfer between UniVerse and the web server. The default is 4096, however, the buffer size can be increased to improve performance. It should be entered as an integer greater than or equal to 4096.
AUTHENTICATE	The user name and password to gain access. The string should be “username:password.” Default Basic authentication can also be set. If a request is denied (HTTP status 401/407), UniVerse BASIC will search for the default credential to automatically resubmit the request.
HEADERS	The header to be sent with the HTTP request. If *default_headers* contains an empty string, any current user-specified default header will be cleared. Currently, the only default header UniVerse BASIC sets automatically is “User-Agent UniVerse 9.6.” If you do not want to send out this header, you should overwrite it with `setHTTPDefault`(). Per RFC 2616, for “net politeness” an HTTP client should always send out this header. UniVerse BASIC also sends a date/time stamp with every HTTP request. According to RFC 2616, the stamp represents time in Universal Time (UT) format. A header should be entered as a dynamic array in the form of <HeaderName>@VM<HeaderValue>@Fm<HeaderName>@VM<HeaderValue>.

Return codes

The following table describes the status of each return code.

Return code	Status
0	Success.
1	Invalid option.
2	Invalid Value.

Note: All defaults set by `setHTTPDefault`() stay in effect until the end of the current UniVerse session. If you do not want the setting to affect subsequent programs, you need to clear it before exiting the current program. If the you want to set the “Authorization” or “Proxy-Authorization” header as defaults, see the description under `setRequestHeader`(). To clear the default settings, pass an empty string with PROXY_NAME, AUTHENTICATE and HEADERS, and 0 for PROXY_PORT and BUFSIZE.

setRequestHeader function

The `setRequestHeader` function enables you to set additional headers for a request.

request_handle is the handle to the request returned by createRequest().

header_name is the name of the header.

header_value is the value of the header.

Syntax

setRequestHeader(*request_handle, header_name, header_value*)

Return codes

The following table describes the status of each return code.

Return code	Status
0	Success.
1	Invalid request handle.
2	Invalid header (Incompatible with method).
3	Invalid header value.

Note: Since a user-defined header or header value can be transferred, it is difficult to check the validity of parameters passed to the function. UniVerse BASIC currently will not perform syntax checking on the parameters, although it will reject setting a response header to a request. Refer to RFC 2616 for valid request headers. The header set by this function will overwrite settings by `setHTTPDefault`().
The header set by this function will overwrite settings by `setHTTPDefault`().

Example

The following example changes the default Content-Type of the HTTP header from “Content-Type: application/x-www-form-urlencoded” to “Content-Type: text/xml; charset=utf-8.”

```
ret=setRequestHeader(REQUEST.HANDLE, "Content-Type",
 "text/xml; charset=utf-8")
```

SETLOCALE function

In NLS mode, use the `SETLOCALE` function to enable or disable a locale for a specified category or change its setting.

Syntax

SETLOCALE (*category, value*)

category is one of the following tokens that are defined in the UVNLSLOC.H file:

Category	Description
UVLC$ALL	Sets or disables all categories as specified in *value*. *value* is the name of a locale, OFF, or DEFAULT. *value* can also be a dynamic array whose elements correspond to the categories.
UVLC$TIME	Sets or disables the Time category. *value* is the name of a locale, OFF, or DEFAULT.

Category	Description
UVLC$NUMERIC	Sets or disables the Numeric category. *value* is the name of a locale, OFF, or DEFAULT. **Note:** Programs must be compiled in the locale in which the numeric constraints were intended. For example, if LOCALE CH-GERMAN is enabled in a US-ENGLISH locale, incorrect results are returned. The US-ENGLISH locale must be compiled with LOCALE US-ENGLISH.
UVLC$MONETARY	Sets or disables the Monetary category. *value* is the name of a locale, OFF, or DEFAULT.
UVLC$CTYPE	Sets or disables the Ctype category. *value* is the name of a locale, OFF, or DEFAULT.
UVLC$COLLATE	Sets or disables the Collate category. *value* is the name of a locale, OFF, or DEFAULT.
UVLC$SAVE	Saves the current locale state, overwriting any previous saved locale. *value* is ignored.
UVLC$RESTORE	Restores the saved locale state. *value* is ignored.

value specifies either a dynamic array whose elements are separated by field marks or the string OFF. An array can have one or five elements:

- If the array has one element, all categories are set or unset to that value.
- If the array has five elements, it specifies the following values in this order: TIME, NUMERIC, MONETARY, CTYPE, and COLLATE.

The MD, MR, and ML conversions require both Numeric and Monetary categories to be set in order for locale information to be used.

The STATUS function returns 0 if SETLOCALE is successful, or one of the following error tokens if it fails:

Error token	Description
LCE$NO.LOCALES	UniVerse locales are disabled.
LCE$BAD.LOCALE	*value* is not the name of a locale that is currently loaded, or the string OFF.
LCE$BAD.CATEGORY	You specified an invalid category.
LCE$NULL.LOCALE	*value* has more than one field and a category is missing.

The error tokens are defined in the UVNLSLOC.H file.

For more information about locales, see the *UniVerse NLS Guide*.

Examples

The following example sets all the categories in the locale to FR-FRENCH:

```
status = SETLOCALE(UVLC$ALL,"FR-FRENCH")
```

The next example saves the current locale. This is the equivalent of executing the `SAVE.LOCALE` command.

```
status = SETLOCALE(UVLC$SAVE,"")
```

The next example sets the Monetary category to DE-GERMAN:

```
status = SETLOCALE(UVLC$MONETARY,"DE-GERMAN")
```

The next example disables the Monetary category. UniVerse behaves as though there were no locales for the Monetary category only.

```
status = SETLOCALE(UVLC$MONETARY,"OFF")
```

The next example completely disables locale support for all categories:

```
status = SETLOCALE(UVLC$ALL,"OFF")
```

The next example restores the locale setting saved earlier:

```
status = SETLOCALE(UVLC$RESTORE,"")
```

SETREM statement

Use the SETREM statement to set the remove pointer in *dynamic.array* to the position specified by *position*.

Syntax

SETREM *position* ON *dynamic.array*

position is an expression that evaluates to the number of bytes you want to move the pointer forward. If it is larger than the length of *dynamic.array*, the length of *dynamic.array* is used. If it is less than 0, 0 is used.

dynamic.array must be a variable that evaluates to a string. If it does not evaluate to a string, an improper data type warning is issued.

If the pointer does not point to the first character after a system delimiter, subsequent REMOVE statement and REVREMOVE statement act as follows:

- A REMOVE statement returns a substring, starting from the pointer and ending at the next delimiter.
- A REVREMOVE statement returns a substring, starting from the previous delimiter and ending at the pointer.

If NLS is enabled and you use a multibyte character set, use GETREM function to ensure that *position* is at the start of a character. For more information about locales, see the *UniVerse NLS Guide*.

Example

```
DYN = "THIS":@FM:"HERE":@FM:"STRING"
 REMOVE VAR FROM DYN SETTING X
 A = GETREM(DYN)
 REMOVE VAR FROM DYN SETTING X
 PRINT VAR
 SETREM A ON DYN
 REMOVE VAR FROM DYN SETTING X
 PRINT VAR
```

The program output is:

```
HERE
 HERE
```

setSocketMap function

The `setSocketMap`() function sets the default NLS map for either server or client sockets. If you call `openSocket`() or `acceptConnection`() prior to calling `setSocketMap`(), UniVerse uses the default map defined in `uvconfig`.

Syntax

`setSocketMap`(*`mapname`*)

setSocketOptions function

The `setSocketOptions`() function sets the current value for a socket option associated with a socket of any type.

Syntax

`setSocketOptions`(*`socket_handle, options`*)

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
socket_handle	The socket handle from `openSocket`(), `acceptSocket`(), or `initServerSocket`().
options	Dynamic Array containing information about the socket options and their current settings. The dynamic array is configured as: *optName1*<VM>*optValue1a*[<VM>*optValue1b*]<FM> *optName2*<VM>*optValue2a*[<VM>*optValue2b*]<FM> *optName3*... Where *optName* is specified by the caller and must be an option name string listed below. For all options other than LINGER, the first *optValue* specifies whether the option is ON or OFF and must be one of two possible values: “1” for ON or “2” for OFF. The second *optValue* is optional and can hold additional data for a specific option. For the LINGER option, the first value will be zero for OFF and non-zero for ON. The second *optValue* is the timeout value, which is the number of time units to wait before closing the socket. The timeout value's unit type (seconds, milliseconds, and so forth) is dependent on the implementation of the `SELECT`() function on your operating system.

The following table describes the available options (case-sensitive) for `setSocketOptions`.

Option	Description
DEBUG	Enable/disable recording of debug information.
REUSEADDR	Enable/disable the reuse of a location address (default)
KEEPALIVE	Enable/disable keeping connections alive.
DONTROUTE	Enable/disable routing bypass for outgoing messages.
LINGER	Linger on close if data is present.

Option	Description
BROADCAST	Enable/disable permission to transmit broadcast messages.
OOBINLINE	Enable/disable reception of out-of-band data in band.
SNDBUF	Set buffer size for output (the default value depends on operating-system type).
RCVBUF	Set buffer size for input (the default value depends on operating-system type).

Return codes

The following table describes the status of each return code.

Return code	Description
0	Success.
Non-zero	See Socket function error return codes, on page 584.

showSecurityContext function

The `showSecurityContext`() function dumps the SSL configuration parameters of a security context into a readable format.

The security context handle must have been returned by a successful execution of `createSecurityContext`() or `loadSecurityContext`().

The configuration information includes: protocol, version, certificate, cipher suite used by this connection and other properties.

> **Warning:** For security reasons, the *privateKey* installed into the context is not displayed. Once installed, there is no way for you to extract it.

Syntax

`showSecurityContext``(`*`context, config`*`)`

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
context	The security context handle.
config	A dynamic array containing the security context data.

Return code status

The following table describes the status of each return code.

Return code	Status
0	Success.
1	Invalid security context handle.
2	Configuration data could not be obtained.

SIGNATURE function

The SIGNATURE() function generates a digital signature or verifies a signature using the supplied key. Digital signature is generally created over a piece of data or document by some cryptographic algorithm and used to prove the authenticity and integrity of the data or document, for example, the recipient of the data with a valid digital signature has reason to believe that the data is from a trusted sender and its contents are not modified.

The *algorithm* parameter specifies the digest algorithm used to construct the signature. There are four actions that can be specified: **RSA-Sign**, **RSA-Verify**, **DSA-Sign**, and **DSA-Verify**. Note that if DSA is chosen, only SHA1 can be specified in *algorithm*.

The data to be signed or verified against a signature can be supplied either directly in *data*, or read from a file whose names is in *data*.

For signing action, a private key should be specified. For verification, a public key is usually expected. However, a private key is also accepted for verification purposes. *Key* can be either in PEM or DER format. If a private key is password protected, the password must be supplied with *pass*.

For verification, *key* can also contain a certificate or name of a certificate file. A signature is expected in *sigIn*.

For signing action, the generated signature is put into *result*.

Syntax

```
SIGNATURE(algorithm, action, data, dataLoc, key, keyLoc, keyFmt, pass,
sigIn, result, p12pass)
```

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
algorithm	A string containing the digest algorithm name (uppercase or lowercase). UniVerse 11.2.4+ supports the following algorithms: ▪ MD4 ▪ MD5 ▪ SHA ▪ SHA1 ▪ SHA224 ▪ SHA256 ▪ SHA384 ▪ SHA512 Versions prior to 11.2.4 support MD2, MDC2, and RMD160. These algorithms are no longer supported in later versions.
action	1 - RSA-Sign (SSL_RSA_SIGN) 2 - RSA-Verify (SSL_RSA_VERIFY) 3 - DSA-Sign (SSL_DSA_SIGN) 4 - DSA-Verify (SSL_DSA_VERIFY)
data	Data or the name of the file containing the data to be signed or verified.

Parameter	Description
dataLoc	1 - Data in a string (SSL_LOC_STRING) 2 - Data in a file (SSL_LOC_FILE)
key	The key or the name of the file containing the key to be used to sign or verify. In the case of verification, key can be a certificate string or a file.
keyLoc	1 - Key is in a string (SSL_LOC_STRING) 2 - Key is in a file (SSL_LOC_FILE) 3 - Key is in a certificate for verification. (Currently, no constant is defined)
keyFmt	1 - PEM (SSL_FMT_PEM) 2 - DER (SSL_FMT_DER) 3 - PKCS #12 (SSL_FMT_P12)
pass	A string containing the pass phrase for the private key.
sigIn	A string containing a digital signature.
result	A generated signature or a file to store the signature.
p12pass	Optional. Sets a password on the PKCS #12 file. This parameter should only be included if using a PKCS #12 certificate that has a password. Otherwise the parameter should be omitted.

Return code status

The following table describes the status of each return code.

Return code	Status
0	Success.
1	Unsupported digest algorithm.
2	The data cannot be read.
3	Message digest cannot be obtained.
4	Invalid parameters.
5	Key cannot be read or is in the wrong format / algorithm.
6	Incorrect password.
7	Signature cannot be generated.
8	Signature cannot be verified.

SIN function

Use the `SIN` function to return the trigonometric sine of an expression. *expression* represents the angle expressed in degrees. Numbers greater than 1E17 produce a warning message, and 0 is returned. If *expression* evaluates to the null value, null is returned.

Syntax

```
SIN (expression)
```

Example

```
PRINT SIN(45)
```

This is the program output:

```
0.7071
```

SINH function

Use the `SINH` function to return the hyperbolic sine of *expression*. *expression* must be numeric and represents the angle expressed in degrees. If *expression* evaluates to the null value, null is returned.

Syntax

SINH (*expression*)

Example

```
PRINT "SINH(2) = ":SINH(2)
```

This is the program output:

```
SINH(2) = 3.6269
```

SLEEP statement

Use the SLEEP statement to suspend execution of a BASIC program, pausing for a specified number of seconds.

seconds is an expression evaluating to the number of seconds for the pause. If *seconds* is not specified, a value of 1 is used. If *seconds* evaluates to the null value, it is ignored and 1 is used.

Syntax

SLEEP [*seconds*]

Example

In the following example the program pauses for three seconds before executing the statement after the SLEEP statement. The EXECUTE statement clears the screen.

```
PRINT "STUDY THE FOLLOWING SENTENCE CLOSELY:"
 PRINT
 PRINT
 PRINT "There are many books in the"
 PRINT "the library."
 SLEEP 3
 EXECUTE 'CS'
 PRINT "DID YOU SEE THE MISTAKE?"
```

This is the program output:

```
STUDY THE FOLLOWING SENTENCE CLOSELY:


 There are many books in the
 the library.
 DID YOU SEE THE MISTAKE?
```

SMUL function

Use the SMUL function to multiply two string numbers and return the result as a string number. You can use this function in any expression where a string or string number is valid, but not necessarily where a standard number is valid, because string numbers can exceed the range of numbers that standard arithmetic operators can handle.

Syntax

SMUL (*string.number.1, string.number.2*)

Either string number can be any valid number or string number.

If either string number contains nonnumeric data, an error message is generated and 0 is used for that number. If either string number evaluates to the null value, null is returned.

Example

```
X = "5436"
 Y = "234"
 Z = SMUL (X,Y)
 PRINT Z
```

This is the program output:

```
1272024
```

SOAPCreateRequest function

The SOAPCreateRequest function creates a SOAP request and returns a handle to the request.

Syntax

SOAPCreateRequest(*URL, soapAction, Request*)

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
URL	A string containing the URL where the web service is located. UniVerse sends the SOAP request to this URL. For information about the format of the URL, see URL format, on page 364. [IN]
soapAction	A string UniVerse uses as the SOAPAction HTTP header for this SOAP request. [IN]
Request	The returned handle to the SOAP request. You can use this handle can be used in subsequent calls to the SOAP API for UniVerse BASIC. [OUT]

URL format

The URL you specify must follow the syntax defined in RFS 1738. The general format is:

http://<*host*>:<*port*>/*path*>?<*searchpart*>

The following table describes each parameter of the syntax.

Parameter	Description
host	Either a name string or an IP address of the host system.
port	The port number to which you want to connect. If you do not specify *port*, UniVerse defaults to 80. Omit the preceding colon if you do not specify this parameter.
path	Defines the file you want to retrieve on the web server. If you do not specify *path*, UniVerse defaults to the home page.
searchpart	Use *searchpart* to send additional information to a web server.

Note: If the URL you define contains a *searchpart*, you must define it in its encoded format. For example, a space is converted to +, and other nonalphanumeric characters are converted to %HH format. You do not need to specify the host and path parameters in their encoded formats. UniVerse BASIC encodes these parameters prior to communicating with the web server.

Return codes

The following table describes the status of each return code.

Return code	Status
0	Success.
1	Invalid URL (Syntactically).
2	Invalid HTTP method (indicates the POST method is not supported by the HTTP server).

You can also use the UniVerse BASIC `STATUS()` function to obtain the return status from the function.

Example

The following code segment illustrates the `SOAPCreateRequest` function:

```
* Create the Request
Ret = SoapCreateRequest(URL, SoapAction, SoapReq)
IF Ret <> 0 THEN
  STOP "Error in SoapCreateRequest: " : Ret
END
.
.
```

SOAPCreateSecureRequest function

The `SOAPCreateSecureRequest` function creates a secure SOAP request and returns a handle to the request.

Syntax

`SOAPCreateSecureRequest`(*`URL, soapAction, Request, security_context`*)

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
URL	A string containing the URL where the web service is located. UniVerse sends the SOAP request to this URL. For information about the format of the URL, see SOAPCreateRequest function, on page 364. [IN]
soapAction	A string UniVerse uses as the SOAPAction HTTP header for this SOAP request. [IN]
Request	The returned handle to the SOAP request. You can use this handle can be used in subsequent calls to the SOAP API for UniVerse BASIC. [OUT]
security_context	A handle to the security context.

URL format

The URL you specify must follow the syntax defined in RFS 1738. The general format is:

http://<*host*>:<*port*>/*path*>?<*searchpart*>

The following table describes each parameter of the syntax.

Parameter	Description
host	Either a name string or an IP address of the host system.
port	The port number to which you want to connect. If you do not specify *port*, UniVerse defaults to 80. Omit the preceding colon if you do not specify this parameter.
path	Defines the file you want to retrieve on the web server. If you do not specify *path*, UniVerse defaults to the home page.
searchpart	Use *searchpart* to send additional information to a web server.

Note: If the URL you define contains a *searchpart*, you must define it in its encoded format. For example, a space is converted to +, and other nonalphanumeric characters are converted to %HH format. You do not need to specify the host and path parameters in their encoded formats. UniVerse BASIC encodes these parameters prior to communicating with the web server.

Return codes

The following table describes the status of each return code.

Return code	Status
0	Success.
1	Invalid URL (Syntactically).
2	Invalid HTTP method (indicates the POST method is not supported by the HTTP server).
101	Invalid security context handle.

You can also use the UniVerse BASIC `STATUS()` function to obtain the return status from the function.

Example

The following code segment illustrates the `SOAPCreateSecureRequest` function:

```
* Create the Request
Ret = SoapCreateSecureRequest(URL, SoapAction, SoapReq, SecurityContext)
IF Ret <> 0 THEN
  STOP "Error in SoapCreateSecureRequest: " : Ret
```

```
END
.
.
```

SOAPGetDefault function

The `SOAPGetDefault` function retrieves default SOAP settings, such as the SOAP version.

Syntax

`SOAPGetDefault``(`*`option, value`*`)`

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
option	A string containing an option name. UniVerse currently only supports the VERSION option. [IN]
value	A string returning the option value. [OUT]

Return codes

The following table describes the status of each return code.

Return code	Status
0	Success.
1	Invalid option (currently, UniVerse only supports the VERSION option).

You can also use the UniVerse BASIC `STATUS`() function to obtain the return status from the function.

SOAPGetFault function

If the `SOAPSubmitRequest` function receives a SOAP Fault, the `SOAPGetFault` function parses the response data from `SOAPSubmitRequest` into a dynamic array of SOAP Fault components.

Syntax

`SOAPGetFault``(`*`respData, soapFault`*`)`

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
respData	Response data from `SOAPSubmitRequest` after receiving a SOAP fault. [IN]

Parameter	Description
soapFault	Dynamic array consisting of Fault Code, Fault String, and optional Fault Detail, for example: <*faultcode*>@AM<*faultstring*>@AM<*faultdetail*>@AM<*faultactor*> Fault code values are XML-qualified names, consisting of: ▪ VersionMismatch ▪ MustUnderstand ▪ DTDNotSupported ▪ DataEncoding Unknown ▪ Sender ▪ Receiver

Return codes

The following table describes the status of each return code.

Return code	Status
0	Success.
1	Invalid response data, possibly not a valid XML document.
2	SOAP Fault not found in response data.

You can also use the UniVerse BASIC `STATUS()` function to obtain the return status from the function.

SOAPGetResponseHeader function

The `SOAPGetResponseHeader` function gets a specific response header after issuing a SOAP request.

Syntax

`SOAPGetResponseHeader``(`*`Request, headerName, headerValue`*`)`

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
Request	Handle to the request created with SOAPCreateRequest function. [IN]
headerName	The header name whose value is being queried. [IN]
headerValue	The header value, if present in the response, or empty string if not (in which case the return status of the function is 2). [OUT]

Return codes

The following table describes the status of each return code.

Return code	Status
0	Success.
1	Invalid request handle.

Return code	Status
2	Header not found in set of response headers.

You can also use the UniVerse BASIC `STATUS()` function to obtain the return status from the function.

SOAPSetRequestBody function

The `SOAPSetRequestBody` function sets up a SOAP request body directly, as opposed to having it constructed via the `SOAPSetParameters` function. With this function, you can also attach multiple body blocks to the SOAP request.

Each SOAP request should include at least one body block.

Syntax

SOAPSetRequestBody(*Request, value*)

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
Request	Handle to the request created with SOAPCreateRequest function. [IN]
value	A dynamic array containing SOAP body blocks, for example: <*body block*>@AM<*body block*>... [IN]

Return codes

The following table describes the status of each return code.

Return code	Description
0	Success.
1	Invalid request handle.

You can also use the UniVerse BASIC `STATUS()` function to obtain the return status from the function.

SOAPSetRequestContent function

The `SOAPSetRequestContent` function sets the entire SOAP request's content from an input string or from a file.

Syntax

SOAPSetRequestContent(*Request, reqDoc, docTypeFlag*)

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
Request	Handle to the request created with SOAPCreateRequest function. [IN]
reqDoc	The input document to use as the SOAP request content. [IN]

Parameter	Description
docTypeFlag	A flag indicating whether *reqDoc* is a string holding the actual content, or the path to a file holding the content. ▪ 0 – *reqDoc* is a file holding the request content. ▪ 1 – *reqDoc* is a string holding the request content. [IN]

Return codes

The following table describes the status of each return code.

Return code	Status
0	Success.
1	Invalid request handle.
2	Unable to open the file named by *reqDoc*.
3	Unable to read the file named by *reqDoc*.

You can also use the UniVerse BASIC `STATUS()` function to obtain the return status from the function.

SOAPSetRequestHeader function

The `SOAPSetRequestHeader` function sets up a SOAP request header. By default, there is no SOAP header.

Syntax

`SOAPSetRequestHeader``(`*`Request, value`*`)`

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
Request	Handle to the request created with SOAPCreateRequest function. [IN]
value	A dynamic array containing SOAP header blocks, for example: <*header block*>@AM<*header block*>...[IN]

Return codes

The following table describes the status of each return code.

Return code	Description
0	Success.
1	Invalid request handle.

You can also use the UniVerse BASIC `STATUS()` function to obtain the return status from the function.

SOAPRequestWrite function

The `SOAPRequestWrite` function outputs the SOAP request, in XML format, to a string or to a file.

Syntax

`SOAPRequestWrite``(`*`Request, reqDoc, docTypeFlag`*`)`

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
Request	Handle to the request created with SOAPCreateRequest function. [IN]
reqDoc	Depending on *docTypeFlag*, either an output string containing the SOAP request content, or a path to a file where the SOAP request content will be written. [OUT]
docTypeFlag	A flag indicating whether *reqDoc* is an output string that is to hold the request content, or a path to a file where the SOAP request content will be written. ▪ 0 – reqDoc is a file where the request content will be written upon successful completion. ▪ 1 – reqDoc is a string that will hold the request upon successful completion. [IN]

Return codes

The following table describes the status of each return code.

Return code	Status
0	Success.
1	Invalid request handle.
2	Unable to open the file named by *reqDoc*.
3	Unable to write to the file named by *reqDoc*.

You can also use the UniVerse BASIC `STATUS`() function to obtain the return status from the function.

SOAPSetDefault function

Use the `SOAPSetDefault` function to define default SOAP settings, such as the SOAP version. By default, the SOAP version is 1.1, although you can specify version 1.2.

Syntax

`SOAPSetDefault``(`*`option, value`*`)`

For SOAP version 1.1, the namespace prefixes "env" and "enc" are associated with the SOAP namespace names http://schemas.xmlsoap.org/soap/envelope/ and http://schemas.xmlsoap.org/soap/encoding/ respectively. The namespace prefixed "xsi" and "xsd" are associated with the namespace names http://www.w3.org/1999/XMLSchema-instance and http://www.w3.org/1999/XMLSchema respectively.

The SOAP version can be set to 1.2 to support the newer SOAP 1.2 protocol. The namespace prefixes "env" and "enc" are associated with the SOAP namespace names "http://www.w3.org/2001/12/soap-envelope" and "http://www.w3.org/2001/12/soap-encoding" respectively. The namespace prefixes "xsd" and "xsi" will be associated with the namespace names "http://www.w3.org/2001/XMLSchema" and "http://www.w3.org/2001/XMLSchema-instance" respectively.

Note: All defaults set by `SOAPSetDefault` remain in effect until the end of the current UniVerse session. If you do not want the setting to affect subsequent programs, clear it before exiting the current program.

Along with `SOAPSetDefault`, you can use the `CallHTTP` function `setHTTPDefault` to set HTTP-specific settings or headers, if the HTTP default settings are not sufficient.

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
option	A string containing an option name. UniVerse currently only supports the “VERSION” option. [IN]
value	A string containing the appropriate option value. For the VERSION option, the string should be 1.0, 1.1, or 1.2. [IN]

Return codes

The following table describes the status of each return code.

Return code	Status
0	Success.
1	Invalid option (currently, UniVerse only supports VERSION).
2	Invalid value. If you do not specify a value, UniVerse uses the default of 1.1.

You can also use the UniVerse BASIC `STATUS`() function to obtain the return status from the function.

SOAPSetParameters function

The `SOAPSetParameters` function sets up the SOAP request body, specifying a remote method to call along with the method's parameter list.

Syntax

SOAPSetParameters(*Request, URI, serviceName, paramArray*)

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
Request	Handle to the request created with SOAPCreateRequest function. [IN]
namespace	A string is used as the namespace URI for the SOAP call. [IN]
serviceName	The name of the SOAP service. [IN]

Parameter	Description
paramArray	A dynamic array containing the method parameters for the SOAP call. Each method parameter consists of the following values: ▪ *A* parameter name ▪ *A* parameter value ▪ *A* parameter type (if type is omitted, *xsd:string* will be used. name, value, and type are separated by @VM. Additional parameters are separated by @AM, as shown in the following example: <*param1Name*>@VM<*param1Value*>@VM<*param1Type*>@AM <*param2Name*>@VM<*param2Value*>@VM<*param2Type*>...[IN]

Return codes

The following table describes the status of each return code.

Return code	Description
0	Success.
1	Invalid request handle.

You can also use the UniVerse BASIC `STATUS()` function to obtain the return status from the function.

Example

As an example, the following inputs:

Input	Description
serviceName	“getStockQuote”
namespace	“http://host/#StockQuoteService”
paramArray	“symbol”:@VM:”U2”:@VM:”xsd:string”

set the SOAP body as follows:

```
<SOAP-ENV:Body>
   <ns1:getStockQuote
     xmlns:ns1="http://host/#StockQuoteService">
         <symbol xsi:type="xsd:string">U2</symbol>
   </ns1:getQuote>
<SOAP-ENV:Body>
```

The following code example illustrates the `SOAPSetParameters` function:

* Set up the Request Body

```
Ret = SoapSetParameters(SoapReq, NameSpace, Method, MethodParms)
IF Ret <> 0 THEN
  STOP "Error in SoapSetParameters: " : Ret
END
```

SOAPSubmitRequest function

The `SOAPSubmitRequest` function submits a request and gets the response.

Internally, SOAPSubmitRequest utilizes CallHTTP's submitRequest() function to send the SOAP message. The *soapStatus* variable holds the status from the underlying CallHTTP function. If an error occurs on the SOAP server while processing the request, *soapStatus* will indicate an HTTP 500 "Internal Server Error", and *respData* will be a SOAP Fault message indicating the server-side processing error.

Syntax

SOAPSubmitRequest(*Request, timeout, respHeaders, respData, soapStatus*)

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
Request	Handle to the request created with SOAPCreateRequest function. [IN]
timeout	Timeout, in milliseconds, to wait for a response. [IN]
respHeaders	Dynamic array of HTTP response headers and their associated values. [OUT]
respData	The SOAP response message. [OUT]
soapStatus	Dynamic array containing status code and explanatory text. [OUT]

Return codes

The following table describes the status of each return code.

Return code	Status
0	Success.
1	Invalid request handle.
2	Request timed out.
3	Network error occurred.
4	Other error occurred.
99	UniVerse failed to obtain a license for an interactive PHANTOM process.

You can also use the UniVerse BASIC STATUS() function to obtain the return status from the function.

Example

The following code sample illustrates the SOAPSubmitRequest function:

```
* Submit the Request
Ret = SoapSubmitRequest(SoapReq, Timeout, RespHeaders, RespData, SoapStatus)
IF Ret <> 0 THEN
STOP "Error in SoapSubmitRequest: " : Ret
END
PRINT "Response status : " : SoapStatus
PRINT "Response headers: " : RespHeaders
PRINT "Response data : " : RespData
.
.
.
```

SOUNDEX function

The SOUNDEX function evaluates *expression* and returns the most significant letter in the input string followed by a phonetic code. Non-alphabetic characters are ignored. If *expression* evaluates to the null value, null is returned.

This function uses the soundex algorithm (the same as the one used by the SAID keyword in RetrieVe) to analyze the input string. The soundex algorithm returns the first letter of the alphabetic string followed by a one- to three-digit phonetic code.

Syntax

SOUNDEX (*expression*)

Example

Source lines	Program output
DATA "MCDONALD", "MACDONALD", "MACDOUGALL"	?MCDONALD
FOR I=1 TO 3	M235
INPUT CUSTOMER	?MACDONALD
PHONETIC.CODE=SOUNDEX(CUSTOMER)	M235
PRINT PHONETIC.CODE	?MACDOUGALL
NEXT	M232

SPACE function

Use the SPACE function to return a string composed of blank spaces. *expression* specifies the number of spaces in the string. If *expression* evaluates to the null value, the SPACE function fails and the program terminates with a run-time error message.

There is no limit to the number of blank spaces that can be generated.

Syntax

SPACE (*expression*)

Example

```
PRINT "HI":SPACE(20):"THERE"
 *
 *
 VAR=SPACE(5)
 PRINT "TODAY IS":VAR:OCONV(DATE(),"D")
```

This is the program output:

```
HI                  THERE
 TODAY IS       18 JUN 1992
```

SPACES function

Use the SPACES function to return a dynamic array with elements composed of blank spaces. *dynamic.array* specifies the number of spaces in each element. If *dynamic.array* or any element of *dynamic.array* evaluates to the null value, the SPACES function fails and the program terminates with a run-time error message.

There is no limit to the number of blank spaces that can be generated except available memory.

If you use the subroutine syntax, the resulting dynamic array is returned as *return.array*.

Syntax

SPACES (*dynamic.array*)

CALL -**SPACES** (*return.array, dynamic.array*)

CALL !**SPACES** (*return.array, dynamic.array*)

SPLICE function

Use the SPLICE function to create a dynamic array of the element-by-element concatenation of two dynamic arrays, separating concatenated elements by the value of *expression*.

Syntax

SPLICE (*array1, expression, array2*)

CALL -**SPLICE** (*return.array, array1, expression, array2*)

CALL !**SPLICE** (*return.array, array1, expression, array2*)

Each element of *array1* is concatenated with *expression* and with the corresponding element of *array2*. The result is returned in the corresponding element of a new dynamic array. If an element of one dynamic array has no corresponding element in the other dynamic array, the element is returned properly concatenated with *expression*. If either element of a corresponding pair is the null value, null is returned for that element. If *expression* evaluates to the null value, null is returned for the entire dynamic array.

If you use the subroutine syntax, the resulting dynamic array is returned as *return.array*.

Example

```
A="A":@VM:"B":@SM:"C"
 B="D":@SM:"E":@VM:"F"
 C='-'
 PRINT SPLICE(A,C,B)
```

This is the program output:

```
A-DS-EVB-FSC-
```

SQRT function

Use the SQRT function to return the square root of *expression*. *expression* must evaluate to a numeric value that is greater than or equal to 0. If *expression* evaluates to a negative value, the result of the

function is SQRT(-*n*) and an error message is printed. If *expression* evaluates to the null value, null is returned.

Syntax

SQRT (*expression*)

Example

```
A=SQRT(144)
 PRINT A
 *
 PRINT "SQRT(45) IS ":SQRT(45)
```

This is the program output:

```
12
 SQRT(45) IS 6.7082
```

SQUOTE function

Use the SQUOTE function to enclose an expression in single quotation marks. If *expression* evaluates to the null value, null is returned, without quotation marks.

Syntax

SQUOTE (*expression*)

CALL !**SQUOTE** (*quoted.expression, expression*)

quoted.expression is the quoted string.

expression is the input string.

Example

```
PRINT SQUOTE(12 + 5) : " IS THE ANSWER."
 END
```

This is the program output:

```
'17' IS THE ANSWER.
```

SSELECT statement

Use an SSELECT statement to create:

- A numbered select list of record IDs in sorted order from a UniVerse file
- A numbered select list of record IDs from a dynamic array. A select list of record IDs from a dynamic array is not in sorted order.

You can then access this select list by a subsequent READNEXT statement which removes one record ID at a time from the list.

Syntax

SSELECT [*variable*] [TO *list.number*] [ON ERROR *statements*]

SSELECTN [*variable*] [TO *list.number*] [ON ERROR *statements*]

SSELECTV [*variable*] TO *list.variable* [ON ERROR *statements*]

variable can specify a dynamic array or a file variable. If it specifies a dynamic array, the record IDs must be separated by field marks (ASCII 254). If *variable* specifies a file variable, the file variable must have previously been opened. If *variable* is not specified, the default file is assumed (for more information on default files, see the OPEN statement). If the file is neither accessible nor open, or if *variable* evaluates to the null value, the SSELECT statement fails and the program terminates with a run-time error message.

Note: The select list created by the SSELECT command is only sorted when you supply a file variable as an argument to the command. If you supply a dynamic array, UniVerse returns the information in the dynamic array as a select list sorted in the same order as the dynamic array.

If the file is an SQL table, the effective user of the program must have SQL SELECT privilege to read records in the file. For information about the effective user of a program, see the AUTHORIZATION statement, on page 71.

You must use a file lock with the SSELECT statement when it is within a transaction running at isolation level 4 (serializable). This prevents phantom reads.

The TO clause specifies the select list that is to be used. *list.number* is an integer from 0 through 10. If no *list.number* is specified, select list 0 is used.

The record IDs of all the records in the file form the list. The record IDs are listed in ascending order. Each record ID is one entry in the list.

You often want a select list with the record IDs in an order different from their stored order or with a subset of the record IDs selected by some specific criteria. To do this, use the SELECT statements or SSELECT commands in a BASIC EXECUTE statement. Processing the list by READNEXT statement is the same, regardless of how the list is created.

Use the SSELECTV statement to store the select list in a named list variable instead of to a numbered select list. *list.variable* is an expression that evaluates to a valid variable name. This is the default behavior of the SSELECT statement in PICK, REALITY, and IN2 flavor accounts. You can also use the VAR.SELECT option of the $OPTIONS statement to make the SSELECT statement act as it does in PICK, REALITY, and IN2 flavor accounts.

In NLS mode when locales are enabled, the SSELECT statements use the Collate convention of the current locale to determine the collating order. For more information about locales, see the *UniVerse NLS Guide*.

The ON ERROR clause

The ON ERROR clause is optional in SSELECT statements. The ON ERROR clause lets you specify an alternative for program termination when a fatal error is encountered during processing of a SSELECT statement.

If a fatal error occurs, and the ON ERROR clause was not specified, or was ignored (as in the case of an active transaction), the following occurs:

- An error message appears.
- Any uncommitted transactions begun within the current execution environment roll back.

- The current program terminates.
- Processing continues with the next statement of the previous execution environment, or the program returns to the UniVerse prompt.

A fatal error can occur if any of the following occur:

- A file is not open.
- *file.variable* is the null value.
- A distributed file contains a part file that cannot be accessed.

If the ON ERROR clause is used, the value returned by the STATUS function is the error number.

PICK, REALITY, and IN2 flavors

In a PICK, REALITY, or IN2 flavor account, the SSELECT statement has the following syntax:

```
SSELECT[V] [variable] TO list.variable
SSELECTN [variable] TO list.number
```

You can use either the SSELECT or the SSELECTV statement to create a select list and store it in a named list variable. The only useful thing you can do with a list variable is use a READNEXT statement to read the next element of the select list.

Use the SSELECTN statement to store the select list in a numbered select list. *list.number* is an expression that evaluates to a number from 0 through 10. You can also use the -VAR.SELECT option of the $OPTIONS statement to make the SSELECT statement act as it does in IDEAL and INFORMATION flavor accounts.

Example

The following example opens the file SUN.MEMBER to the file variable MEMBER.F, then creates an active sorted select list of record IDs. The READNEXT statement assigns the first record ID in the select list to the variable @ID, then prints it. Next, the file SUN.SPORT is opened to the file variable SPORT.F, and a sorted select list of its record IDs is stored as select list 1. The READNEXT statement assigns the first record ID in the select list to the variable A, then prints DONE.

```
OPEN '','SUN.MEMBER' ELSE PRINT "NOT OPEN"
 SSELECT
 READNEXT @ID THEN PRINT @ID
 *
 OPEN '','SUN.SPORT' ELSE PRINT "NOT OPEN"
 SSELECT TO 1
 READNEXT A FROM 1 THEN PRINT "DONE" ELSE PRINT "NOT"
```

This is the program output:

```
0001
 DONE
```

SSUB function

Use the SSUB function to subtract *string.number.2* from *string.number.1* and return the result as a string number. You can use this function in any expression where a string or string number is valid, but not necessarily where a standard number is valid, because string numbers can exceed the range of numbers that standard arithmetic operators can handle.

Either string number can be any valid number or string number.

If either string number contains nonnumeric data, an error message is generated, and 0 replaces the nonnumeric data. If either string number evaluates to the null value, null is returned.

Syntax

SSUB (*string.number.1, string.number.2*)

Example

```
X = "123456"
 Y = "225"
 Z = SSUB (X,Y)
 PRINT Z
```

This is the program output:

```
123231
```

STATUS function

Use the STATUS function to determine the results of the operations performed by certain statements and functions.

The parentheses must be used with the STATUS function to distinguish it from potential user-named variables called STATUS. However, no arguments are required with the STATUS function.

Syntax

STATUS ()

The following sections describe STATUS function values.

After a BSCAN statement, on page 78:

Value	Description
0	The scan proceeded beyond the leftmost or rightmost leaf node. *ID.variable* and *rec.variable* are set to empty strings.
1	The scan returned an existing record ID, or a record ID that matches *record*.
2	The scan returned a record ID that does not match *record*. *ID.variable* is either the next or the previous record ID in the B-tree, depending on the direction of the scan.
3	The file is not a B-tree (type 25) file, or, if the USING clause is used, the file has no active secondary indexes.
4	*indexname* does not exist.
5	*seq* does not evaluate to A or D.
6	The index specified by *indexname* needs to be built, or is currently being built concurrently.
10	An internal error was detected.

After a DELETE statement:

After DELETE statements with an ON ERROR clause, the value returned is the error number. In some instances, the error number returned corresponds to a record in the SYS.MESSAGE file. Record IDs in the SYS.MESSAGE file are numeric and consist of six digits. The error message number returned does

not contain leading zeros. If the number returned is less than six digits, it should be prefixed with zeros before reading from the SYS.MESSAGE file.

After a FILEINFO function:

After a successful execution of the FILEINFO function, STATUS returns 0. If the function fails to execute, `STATUS` returns a nonzero value. For complete information, see the `FILEINFO` function.

After a FILELOCK statement:

After a FILELOCK statement with a LOCKED clause, the value returned is the terminal number of the user who has a conflicting lock.

After an FMT function:

Value	Description
0	The conversion is successful.
1	The string expression passed as an argument is invalid. If NLS is enabled: the data supplied cannot be converted.
2	The conversion code passed as an argument to the function is invalid.

After a GET or GETX statement:

Value	Description
0	The timeout limit expired.
Any nonzero value	A device input error occurred.

After an ICONV or OCONV function:

Value	Description
0	The conversion is successful.
1	The string expression passed as an argument to the function is not convertible using the conversion code passed. An empty string is returned as the value of the function.
2	The conversion code passed as an argument to the function is invalid. An empty string is returned as the value of the function.
3	Successful conversion of a possibly invalid date.

After an INPUT @ statement:

A 0 is returned if the statement was completed by a Return. The trap number is returned if the statement was completed by one of the trapped keys (see the INPUT @ and KEYTRAP statement, on page 234).

After a MATWRITE, WRITE, WRITEU, WRITEV, or WRITEVU statement:

Value	Description
0	The record was locked before the operation.
3	In NLS mode, the unmappable character is in the record ID.
4	In NLS mode, the unmappable character is in the record's data.
-2	The record was unlocked before the operation.
-3	The record failed an SQL integrity check.
-4	The record failed a trigger program.
-6	Failed to write to a published file while the subsystem was shut down.

After an OPEN, OPENCHECK, OPENPATH, or OPENSEQ statement:

The file type is returned if the file is opened successfully. If the file is not opened successfully, the following values may return:

Value	Description
-1	File name not found in the VOC file.
-2	A generic error that can occur for various reasons. Null file name or file. This error may also occur when you cannot open a file across UVNet.
-3	Operating system access error that occurs when you do not have permission to access a UniVerse file in a directory. For example, this may occur when trying to access a type 1 or type 30 file.
-4	Access error when you do not have operating system permissions or if DATA.30 is missing for a type 30 file.
-5	Read error detected by the operating system.
-6	Unable to lock file header.
-7	Invalid file revision or wrong byte-ordering for the platform.
-8	Invalid part file information.
-9	Invalid type 30 file information in a distributed file.
-10	A problem occurred while the file was being rolled forward during warmstart recovery. Therefore, the file is marked “inconsistent.”
-11	The file is a view, therefore it cannot be opened by a BASIC program.
-12	No SQL privileges to open the table.
-13	Index problem.
-14	Cannot open the NFS file.
-15	There is a problem with the OVER.30 file in a dynamic file.
-16	Modulo over limit.
-17	Freechain corruption.
-18	SICA corruption.
-19	External Database Access (EDA) setup error.
-20	Automatic Data Encryption (ADE) setup error.

After a READ statement:

If the file is a distributed file, the STATUS function returns the following:

Value	Description
-1	The partitioning algorithm does not evaluate to an integer.
-2	The part number is invalid.

After a READBLK statement:

Value	Description
0	The read is successful.
1	The end of file is encountered, or the number of bytes passed in was less than or equal to 0.
2	The read failed.
3	A partial read failed.

Value	Description
-1	The file is not open for a read.

After a READL, READU, READVL, or READVU statement:

If the statement includes the LOCKED clause, the returned value is the terminal number, as returned by the `WHO` command, of the user who set the lock.

If NLS is enabled, the results depend on the following:

- The existence of the ON ERROR clause
- The setting of the NLSREADELSE parameter in the `uvconfig` file
- The location of the unmappable character.

Value	Description
3	The unmappable character is in the record ID.
4	The unmappable character is in the record's data.

After a READSEQ statement:

Value	Description
0	The read is successful.
1	The end of file is encountered, or the number of bytes passed in was less than or equal to 0.
2	A timeout ended the read.
-1	The file is not open for a read.

After a READT, REWIND, WEOF, or WRITET statement:

If the statement takes the ELSE clause, the returned value is 1. Otherwise the returned value is 0.

After an RPC.CALL function, RPC.CONNECT function, or RPC.DISCONNECT function:

Value	Description
81001	A connection was closed for an unspecified reason.
81002	*connection.ID* does not correspond to a valid bound connection.
81004	Error occurred while trying to store an argument in the transmission packet.
81005	Procedure access denied because of a mismatch of RPC versions.
81007	Connection refused because the server cannot accept more clients.
81008	Error occurred because of a bad parameter in *arg.list.*
81009	An unspecified RPC error occurred.
81010	*#args* does not match the expected argument count on the remote machine.
81011	Host was not found in the local /etc/hosts file.
81012	Remote *unirpcd* cannot start the service because it could not fork the process.
81013	The remote *unirpcservices* file cannot be opened.
81014	Service was not found in the remote *unirpcservices* file.
81015	A timeout occurred while waiting for a response from the server.

After a SETLOCALE function:

The `STATUS` function returns 0 if SETLOCALE function, on page 356 is successful, or one of the following error tokens if it fails:

Value	Description
LCE$NO.LOCALES	UniVerse locales are disabled.
LCE$BAD.LOCALE	The specified locale name is not currently loaded, or the string OFF.
LCE $BAD.CATEGORY	You specified an invalid category.
LCE$NULL.LOCALE	The specified locale has more than one field and a category is missing.

After a WRITESEQ, WRITESEQF, or WRITEBLK statement:

The `STATUS` function returns –4 if a write operation runs out of disk space on the device being written to.

Example

Source lines	Program output
OPEN '','EX.BASIC' TO FILE ELSE STOP PRINT 'STATUS() IS ':STATUS() Q=123456 Q=OCONV(Q,"MD2") PRINT 'STATUS() IS ':STATUS() Q='ASDF' Q=OCONV(Q,"D2/") PRINT 'STATUS() IS ':STATUS()	STATUS() IS 1 STATUS() IS 0 STATUS() IS 1

STATUS statement

Use the STATUS statement to determine the status of an open file. The STATUS statement returns the file status as a dynamic array and assigns it to *dynamic.array*.

Syntax

```
STATUS dynamic.array FROM file.variable
        {THEN statements [ELSE statements] | ELSE statements}
```

The following table lists the values of the dynamic array returned by the STATUS statement:

Field	Stored value	Description
1	Current position in the file	Offset in bytes from beginning of the file.
2	End of file reached	1 if EOF, 0 if not.
3	Error accessing file	1 if error, 0 if not.
4	Number of bytes available to read	
5	File mode	UNIX: A combination of permissions (convert to octal) and file type. For example, if the permissions were 777, this value would be: ‘100777’ : standard file’ ‘40777’ : directory (for example, type 1 or 19) ‘10777’ : pipe Windows platforms. This is the UNIX owner-group-other format as converted from the full Windows NT ACL format by the C run-time libraries.
6	File size	In bytes.

Field	Stored value	Description
7	Number of hard links	0 if no links. Windows NT: The value is always 1 on non-NTFS partitions, > 0 on NTFS partitions.
8	User ID of owner	UNIX: The number assigned in /etc/passwd. Windows NT: It is a UniVerse pseudo user ID based on the user name and domain of the user.
9	Group ID of owner	UNIX: The number assigned in /etc/passwd. Windows NT: It is always 0.
10	I-node number	Unique ID of file on file system; on Windows NT the value is the Pelican internal version of the i-node for a file. For dynamic files, the i-node number is the number of the directory holding the components of the dynamic file.
11	Device on which i-node resides	Number of device. The value is an internally calculated value on Windows NT.
12	Device for special character or block	Number of device. The value is the drive number of the disk containing the file on Windows NT.
13	Time of last access	Time in internal format.
14	Date of last access	Date in internal format.
15	Time of last modification	Time in internal format.
16	Date of last modification	Date in internal format.
17	Time and date of last status change	Time and date in internal format. On Windows NT it is the time the file was created.
18	Date of last status change	Date in internal format. On Windows NT it is the date the file was created.
19	Number of bytes left in output queue (applicable to terminals only)	
20	Operating system file name	The internal path name UniVerse uses to access the file.
21	UniVerse file type	For file types 1–19, 25, or 30.
22	UniVerse file modulo	For file types 2–18 only.
23	UniVerse file separation	For file types 2–18 only.
24	Part numbers of part files belonging to a distributed file	Multivalued list. If file is a part file, this field contains the part number, and field 25 is empty.
25	Path names of part files belonging to a distributed file	Multivalued list. If file is a part file, this field is empty.
26	File names of part files belonging to a distributed file	Multivalued list. If file is a part file, this field is empty.
27	Full path name	The full path name of the file. On Windows NT, the value begins with the UNC share name, if available; if not, the drive letter.

Field	Stored value	Description
28	Integer from 1 through 7	SQL file privileges: 1 write-only 2 read-only 3 read/write 4 delete-only 5 delete/write 6 delete/read 7 delete/read/write
29		1 if this is an SQL table, 0 if not. If the file is a view, the STATUS statement fails. (No information on a per-column basis is returned.)
30	User name	User name of the owner of the file.
31	File revision stamp	One of the following: ACEF01*xx* = 32-bit file ACEF02*xx* = 64-bit file *xx* is the file revision level
32	Addressing and Header Support Style	1 = old style file header, 32-bit addressing 3 = new style file header, 32-bit addressing 5 = new style file header, 64-bit addressing
33	Maximum record ID length	See the following table.

file.variable specifies an open file. If *file.variable* evaluates to the null value, the STATUS statement fails and the program terminates with a run-time error message.

If the STATUS array is assigned to *dynamic.array*, the THEN statements are executed and the ELSE statements are ignored. If no THEN statements are present, program execution continues with the next statement. If the attempt to assign the array fails, the ELSE statements are executed; any THEN statements are ignored.

The following table shows maximum record ID lengths for different file sizes:

Separation	Block size	Maximum ID length	Comments
1	512	256	Existing maximum
2	1024	512	
3	1536	768	
4	2048	1024	Dynamic file GROUP.SIZE of 1
5	2560	1280	
6	3076	1538	
7	3584	1792	
8	4096	2048	Dynamic file GROUP.SIZE of 2
9 or higher	4608 and up	2048	All remaining separations

Example

```
OPENSEQ '/etc/passwd' TO test THEN PRINT "File Opened" ELSE ABORT
 STATUS stat FROM test THEN PRINT stat
 field5 = stat<5,1,1>
 field6 = stat<6,1,1>
 field8 = stat<8,1,1>
 PRINT "permissions:": field5
 PRINT "filesize:": field6
 PRINT "userid:": field8
 CLOSESEQ test
```

This is the program output:

```
File Opened
 0F0F0F4164F33188F4164F1F0F2F2303F
   0F6856F59264F6590F42496F6588F42496F6588
     F0F/etc/passwdF0F0F0
 permissions:33188
 filesize:4164
 userid:0
```

STOP statement

Use the STOP statement to terminate program execution and return system control to the invoking process. To terminate a subroutine and return to the calling program, use the RETURN statement.

When *expression* is specified, its value is displayed before the STOP statement is executed. If *expression* evaluates to the null value, nothing is printed.

To stop all processes and return to the command level, use the ABORT statement.

Use the ERRMSG statement if you want to display a formatted error message from the ERRMSG file when the program stops.

Syntax

STOP [*expression*]

STOPE [*expression*]

STOPM [*expression*]

STOPE and STOPM statements

The STOPE statement uses the ERRMSG file for error messages instead of using text specified by *expression*. The STOPM statement uses text specified by *expression* rather than messages in the ERRMSG file. If *expression* in the STOPE statement evaluates to the null value, the default error message is printed:

```
Message ID is NULL: undefined error
```

PICK, IN2, and REALITY flavors

In PICK, IN2, and REALITY flavor accounts, the STOP statement uses the ERRMSG file for error messages instead of using text specified by *expression*. Use the STOP.MSG option of the $OPTIONS statement to get this behavior in IDEAL and INFORMATION flavor accounts.

Example

```
PRINT "1+2=":1+2
 STOP "THIS IS THE END"
```

This is the program output:

```
1+2=3
 THIS IS THE END
```

STORAGE statement

The STORAGE statement performs no function. It is provided for compatibility with other Pick systems.

Syntax

STORAGE *arg1arg2arg3*

STR function

Use the STR function to produce a specified number of repetitions of a particular character string.

Syntax

STR (*string, repeat*)

string is an expression that evaluates to the string to be generated.

repeat is an expression that evaluates to the number of times *string* is to be repeated. If *repeat* does not evaluate to a value that can be truncated to a positive integer, an empty string is returned.

If *string* evaluates to the null value, null is returned. If *repeat* evaluates to the null value, the STR function fails and the program terminates with a run-time error message.

Example

```
PRINT STR('A',10)
 *
 X=STR(5,2)
 PRINT X
 *
 X="HA"
 PRINT STR(X,7)
```

This is the program output:

```
AAAAAAAAAA
 55
 HAHAHAHAHAHAHA
```

STRS function

Use the STRS function to produce a dynamic array containing the specified number of repetitions of each element of *dynamic.array*.

Syntax

STRS (*dynamic.array, repeat*)

CALL -**STRS** (*return.array, dynamic.array, repeat*)

CALL !**STRS** (*return.array, dynamic.array, repeat)*

dynamic.array is an expression that evaluates to the strings to be generated.

repeat is an expression that evaluates to the number of times the elements are to be repeated. If it does not evaluate to a value that can be truncated to a positive integer, an empty string is returned for *dynamic.array*.

If *dynamic.array* evaluates to the null value, null is returned. If any element of *dynamic.array* is the null value, null is returned for that element. If *repeat* evaluates to the null value, the STRS function fails and the program terminates with a run-time error message.

If you use the subroutine syntax, the resulting dynamic array is returned as *return.array*.

Example

```
ABC="A":@VM:"B":@VM:"C"
 PRINT STRS(ABC,3)
```

This is the program output:

```
AAAVBBBVCCC
```

submitRequest function

The submitRequest function will submit a request and get a response.

Syntax

submitRequest(*request_handle, time_out,*
post_data,response_headers,response_data, http_status)

The request is formed on the basis of default HTTP settings and previous setRequestHeader() and addRequestParameter() values. Specifically, for a GET method with parameters added, a parameter string (properly encoded) is created and attached to the URL string after the “?” character.

For a POST request with nonempty post_data, the data is attached to the request message as is. No encoding is performed, and any parameters added through addRequestParameter() will be totally ignored. Otherwise the following processing will be performed.

For a POST request with default content type, the parameter string is assembled, a Content-Length header created, and then the string is attached as the last part of the request message.

For a POST request with multipart/* content type, a unique boundary string is created and then multiple parts are generated in the sequence they were added through calling addRequestParameter(). Each will have a unique boundary, followed by optional Content-*

headers, and data part. The total length is calculated and a Content-Length header is added to the message header.

The request is then sent to the Web server identified by the URL supplied with the request and created through `createRequest`() (maybe via a proxy server). UniVerse BASIC then waits for the web server to respond. Once the response message is received, the status contained in the response is analyzed.

If the response status indicates that redirection is needed (status 301, 302, 305 or 307), it will be performed automatically, up to five consecutive redirections (the limit is set to prevent looping, suggested by RFC 2616).

If the response status is 401 or 407 (access denied), the response headers are examined to see if the server requires (or accepts) BASIC authentication. If no BASIC authentication request is found, the function returns with an error. Otherwise, default Authentication (set by `setHTTPDefault`) is used to re-send the request. If no default authentication is set, and no other cached user authentication is found, the function will return with an error.

If the user provides authentication information through “Authorization” or “Proxy-Authorization” header, the encoded information is cached. If later, a Basic authentication request is raised, no default authentication is found, and only one user/password encoding is cached, it will be used to re-send the request.

The response from the HTTP server is disposed into *response_header* and *response_data*. It is the user’s responsibility to parse the headers and data. UniVerse BASIC only performs transfer encoding (chunked encoding), and nothing else is done on the data. In other words, content-encoding (gzip, compress, deflate, and so forth) are supposed to be handled by the user, as with all MIME types.

Also, if a response contains header “Content-type: multipart/*”, all the data (multiple bodies enclosed in “boundary delimiters,” see RFC 2046) is stored in *response_data*. It is the user’s responsibility to parse it according to “boundary” parameter.

request_handle is the handle to the request.

time_out is the timeout value (in milliseconds) before the wait response is abandoned.

post_data is the data sent with the POST request.

response_headers is a dynamic array to store header/value pairs.

response_data is the resultant data (may be in binary format).

http_status is a dynamic array containing the status code and explanatory ext.

Return codes

The following table describes the status of each return code.

Return code	Status
0	Success.
1	Invalid request handle.
2	Timed out.
3	Network Error.
4	Other Errors.
99	UniVerse failed to obtain a license for an interactive PHANTOM process.

SUBR function

Use the `SUBR` function to return the value of an external subroutine. The `SUBR` function is commonly used in I-descriptors.

Syntax

```
SUBR (name, [argument [,argument …]] )
```

name is an expression that evaluates to the name of the subroutine to be executed. This subroutine must be cataloged in either a local catalog or the system catalog, or it must be a record in the same object file as the calling program. If *name* evaluates to the null value, the SUBR function fails and the program terminates with a run-time error message.

argument is an expression evaluating to a variable name whose value is passed to the subroutine. You can pass up to 254 variables to the subroutine.

Subroutines called by the SUBR function must have a special syntax. The SUBROUTINE statement defining the subroutine must specify a dummy variable as the first parameter. The value of the subroutine is the value of the dummy variable when the subroutine finishes execution. Because the SUBROUTINE statement has this dummy parameter, the SUBR function must specify one argument less than the number of parameters in the SUBROUTINE statement. In other words, the SUBR function does not pass any argument to the subroutine through the first dummy parameter. The first argument passed by the SUBR function is referenced in the subroutine by the second parameter in the SUBROUTINE statement, and so on.

Example

The following example uses the globally cataloged subroutine *TEST:

```
OPEN "","SUN.MEMBER" TO FILE ELSE STOP "CAN'T OPEN DD"
 EXECUTE "SELECT SUN.MEMBER"
 10*
 READNEXT KEY ELSE STOP
 READ ITEM FROM FILE,KEY ELSE GOTO 10
 X=ITEM<7> ;* attribute 7 of file contains year
 Z=SUBR("*TEST",X)
 PRINT "YEARS=", Z
 GOTO 10
```

This is the subroutine TEST:

```
SUBROUTINE TEST(RESULT,X)
 DATE=OCONV(DATE(),"D2/")
 YR=FIELD(DATE,'/',3)
 YR='19':YR
 RESULT=YR-X
 RETURN
```

This is the program output:

```
15 records selected to Select List #0
 YEARS=     3
 YEARS=     5
 YEARS=     2
 YEARS=     6
 YEARS=     1
 YEARS=     0
 YEARS=     0
 YEARS=     1
 YEARS=     4
 YEARS=     6
 YEARS=     1
 YEARS=     2
 YEARS=     7
```

```
YEARS=      1
YEARS=      0
```

SUBROUTINE statement

Use the SUBROUTINE statement to identify an external subroutine. The SUBROUTINE statement must be the first noncomment line in the subroutine. Each external subroutine can contain only one SUBROUTINE statement.

An external subroutine is a separate program or set of statements that can be executed by other programs or subroutines (called *calling programs*) to perform a task. The external subroutine must be compiled and cataloged before another program can call it.

The SUBROUTINE statement can specify a subroutine name for documentation purposes; it need not be the same as the program name or the name by which it is called. The CALL statement must reference the subroutine by its name in the catalog, in the VOC file, or in the object file.

variables are variable names used in the subroutine to pass values between the calling programs and the subroutine. To pass an array, you must precede the array name with the keyword MAT. When an external subroutine is called, the CALL statement must specify the same number of variables as are specified in the SUBROUTINE statement. See the CALL statement, on page 81 for more information.

Syntax

SUBROUTINE [*name*] [([MAT] *variable* [, [MAT] *variable* …])]

Example

The following SUBROUTINE statements specify three variables, EM, GROSS, and TAX, the values of which are passed to the subroutine by the calling program:

```
SUBROUTINE ALONE(EM, GROSS, TAX)

 SUBROUTINE STATE(EM,GROSS,TAX)
```

SUBS function

Use the SUBS function to create a dynamic array of the element-by-element subtraction of two dynamic arrays.

Each element of *array2* is subtracted from the corresponding element of *array1* with the result being returned in the corresponding element of a new dynamic array.

If an element of one dynamic array has no corresponding element in the other dynamic array, the missing element is evaluated as 0. If either of a corresponding pair of elements is the null value, null is returned for that element.

If you use the subroutine syntax, the resulting dynamic array is returned as *return.array*.

Syntax

SUBS (*array1, array2*)

CALL -**SUBS** (return.array, array1, array2)

CALL !**SUBS** (*return.array, array1, array2*)

Example

```
A=2:@VM:4:@VM:6:@SM:18
 B=1:@VM:2:@VM:3:@VM:9
 PRINT SUBS(A,B)
```

This is the program output:

```
1V2V3S18V-9
```

SUBSTRINGS function

Use the SUBSTRINGS function to create a dynamic array each of whose elements are substrings of the corresponding elements of *dynamic.array*.

Syntax

SUBSTRINGS (*dynamic.array, start, length*)

CALL -**SUBSTRINGS** (*return.array, dynamic.array, start, length*)

CALL !**SUBSTRINGS** (*return.array, dynamic.array, start, length*)

start indicates the position of the first character of each element to be included in the substring. If *start* is 0 or a negative number, the starting position is assumed to be 1. If *start* is greater than the number of characters in the element, an empty string is returned.

length specifies the total length of the substring. If *length* is 0 or a negative number, an empty string is returned. If the sum of *start* and *length* is larger than the element, the substring ends with the last character of the element.

If an element of *dynamic.array* is the null value, null is returned for that element. If *start* or *length* evaluates to the null value, the SUBSTRINGS function fails and the program terminates with a run-time error message.

If you use the subroutine syntax, the resulting dynamic array is returned as *return.array*.

Example

```
A="ABCDEF":@VM:"GH":@SM:"IJK"
 PRINT SUBSTRINGS(A,3,2)
```

This is the program output:

```
CDVSK
```

SUM function

Use the SUM function to calculate the sum of numeric data. Only elements at the lowest delimiter level of a dynamic array are summed. The total is returned as a single element at the next highest delimiter level.

The delimiters from highest to lowest are field, value, and subvalue.

There are seven levels of delimiters from CHAR(254) to CHAR(248): field mark, value mark, subvalue mark, text mark, CHAR(250), CHAR(249), and CHAR(248).

The SUM function removes the lowest delimiter level from a dynamic array. In a dynamic array that contains fields, values, and subvalues, the SUM function sums only the subvalues, returning the sums as values. In a dynamic array that contains fields and values, the SUM function sums only the values, returning the sums as fields. In a dynamic array that contains only fields, the SUM function sums the fields, returning the sum as the only field of the array. SUM functions can be applied repeatedly to raise multilevel data to the highest delimiter level or to a single value.

Nonnumeric values, except the null value, are treated as 0. If *dynamic.array* evaluates to the null value, null is returned. Any element that is the null value is ignored, unless all elements of *dynamic.array* are null, in which case null is returned.

Syntax

SUM (*dynamic.array*)

Examples

In the following examples a field mark is shown by F, a value mark is shown by V, and a subvalue mark is shown by S.

Source lines	Program output
X=20:@VM:18:@VM:9:@VM:30:@VM:80 PRINT "SUM(X)=",SUM(X)	SUM(X)= 157
X=17:@FM:18:@FM:15 Y=10:@FM:20 PRINT "SUM(X)+SUM(Y)= ",SUM(X)+SUM(Y)	SUM(X)+SUM(Y)= 80
X=3:@SM:4:@SM:10:@VM:3:@VM:20 Y=SUM(X) PRINT "Y= ",Y Z=SUM(Y) PRINT "Z= ",Z	Y= 17V3V20 Z= 40

SUMMATION function

Use the SUMMATION function to return the sum of all the elements in *dynamic.array*. Nonnumeric values, except the null value, are treated as 0.

Syntax

SUMMATION (*dynamic.array*)

CALL !**SUMMATION** (*result* , *dynamic.array*)

result is a variable containing the result of the sum.

dynamic.array is the dynamic array whose elements are to be added together.

Example

```
A=1:@VM:"ZERO":@SM:20:@FM:-25
 PRINT "SUMMATION(A)=",SUMMATION(A)
```

This is the program output:

```
SUMMATION(A)= -4
```

SWAP statement

The SWAP statement interchanges the values in the variables you specify. *variable* can be any valid variable, for example, integers, numbers, characters, and so forth.

You must ensure that the descriptor contains valid values for SWAP.

Syntax

For variables: **SWAP** *variable1, variable2*

For arrays: **SWAP** MAT *variable1*, MAT *variable2*

Example

The following example illustrates the SWAP statement.

```
A=123
b=123.45
SWAP A, B
PRINT A, B
123.45 123
```

SYSTEM function

Use the SYSTEM function to check on the status of a system function. Use the SYSTEM function to test whether NLS is on when you run a program, and to display information about NLS settings.

Syntax

SYSTEM (*expression*)

expression evaluates to the number of the system function you want to check. If *expression* evaluates to the null value, the SYSTEM function fails and the program terminates with a run-time error message.

The following table lists the values for *expression* and their meanings. Values 100 through 107 (read-only) for the SYSTEM function contain NLS information. See the include file UVNLS.H for their tokens.

Value	Action
1	Checks to see if the PRINTER ON statement has turned the printer on. Returns 1 if the printer is on and 0 if it is not.
2	Returns the page width as defined by the terminal characteristic settings.
3	Returns the page length as defined by the terminal characteristic settings.
4	Returns the number of lines remaining on the current page.
5	Returns the current page number.
6	Returns the current line number.
7	Returns the terminal code for the type of terminal the system believes you are using.

Value	Action
8,*n*	Checks whether the tape is attached. Returns the current block size if it is and -1 if it is not. *n* is the number of the tape unit. If it is not specified, tape unit 0 is assumed.
9	Returns the current CPU millisecond count.
10	Checks whether the DATA stack is active. Returns 1 if it is active and 0 if it is not.
11	Checks whether select list 0 is active. Returns 1 if select list 0 is active and 0 if it is not.
12	By default, returns the current system time in seconds (local time). If the TIME.MILLISECOND option is set (see $OPTIONS statement, on page 23), returns the current system time in milliseconds.
13	Not used. Returns 0.
14	Not used. Returns 0.
15	Not used. Returns 0.
16	Returns 1 if running from a proc, otherwise returns 0.
17	Not used. Returns 0.
18	Returns the terminal number.
19	Returns the login name.
20	Not used. Returns 0.
21	Not used. Returns 0.
22	Not used. Returns 0.
23	Checks whether the Break key is enabled. Returns 1 if the Break key is enabled and 0 if it is not.
24	Checks whether character echoing is enabled. Returns 1 if character echoing is enabled and 0 if it is not.
25	Returns 1 if running from a phantom process, otherwise returns 0.
26	Returns the current prompt character.
27	Returns the user ID of the person using the routine.
28	Returns the effective user ID of the person using the routine. Windows NT: This is the same value as 27.
29	Returns the group ID of the person using the routine. Windows NT: This value is 0.
30	Returns the effective group ID of the person using the routine. Windows NT: This value is 0.
31	Returns the UniVerse serial number.
32	Returns the location of the UV account directory.
33	Returns the last command on the command stack.
34	Returns data pending.
35	Returns the number of users currently in UniVerse.
36	Returns the maximum number of UniVerse users.
37	Returns the number of UNIX users; on Windows NT systems returns same value as 35.
38	Returns the path name of the temporary directory.
42	Returns an empty string. On Windows NT systems returns the current value of the telnet client's IP address, or an empty string if the process evaluating the `SYSTEM` function is not the main UniVerse telnet process.
43	Returns 1 if db suspension is on, returns 0 if it is not.
44	Returns the number of UniVerse processes.

Value	Action
45	Returns the BREAK count, which is the number of times breaks were disabled.
50	Returns the field number of the last READNEXT statement when reading an exploded select list.
51	Returns information about device licensing. If you are not using device licensing, SYSTEM(51) returns a null string. If device licensing is enabled but you are not using uvdls as a shell, UniVerse returns an IP address of 0.0.0.0.
60	Returns the current value of the UniVerse configurable parameter TXMODE. The value can be either 1 or 0.
61	Returns the status of the transaction log daemon. 1 indicates the daemon is active; 0 indicates it is inactive.
62	MODFPTRS status.
63	BLKMAX value.
64	MAXKEYSIZE value.
91	Returns 0; on Windows NT, returns 1.
99	Returns the system time in the number of seconds since midnight Coordinated Universal Time (UTC), January 1, 1970.
100	Returns 1 if NLS is enabled, otherwise returns 0.
101	Returns the value of the NLSLCMODE parameter, otherwise returns 0.
102	Reserved for future NLS extensions.
103	Returns the terminal map name assigned to the current terminal print channel, otherwise returns 0.
104	Returns the auxiliary printer map name assigned to the current terminal print channel, otherwise returns 0.
105	Returns a dynamic array with field marks separating the elements, containing the current values of the `uvconfig` file parameters for NLS maps, otherwise 0. Starting at 11.3.1, the value of NLSDEFSOCKMAP is reported in attribute 18 of the result. See the UVNLS.H include file for a list of tokens that define the field order.
106	Returns the current map name used for sequential I/O. Token is NLS$SEQMAP unless overridden by a `SET.SEQ.MAP` command.
107	Returns the current map name for GCI string arguments unless overridden by a `SET.GCI.MAP` command.
108	NLSsvrmap
109	Returns or sets the AUTOLOGOUT value. The value returned and/or passed to this function uses a unit of seconds. The TCL command `AUTOLOGOUT` uses a unit of minutes.
1000	Q_PGBRK
1001	Returns the UniVerse flavor: 1 for IDEAL, 2 for PICK, 4 for INFORMATION, 8 for REALITY, 16 for IN2, and 64 for PIOPEN.
1002-1016	Printer definition settings.
1017	Returns the user's supplementary UNIX groups in a dynamic array.
1020	Reuse.
1021	Returns the GCI error number.
1022	NLSopenelse
1030	Parse @SENTENCE.
1031	Turn string into a quoted argument. This is used for quoting SQL table names.
1050	Returns a dynamic array of key_cntrl_entries.

Value	Action
1200, *hostname*	Returns the UVNet link number associated with *hostname*. If there is an internal error adding *hostname*, 0 returns. *hostname* is an expression that contains the host name from a file opened through UVNet. It refers to the host name portion of the file's path name. For example, in the path name ORION*!/u1/filename*, *hostname* is ORION.
1201, *hostname*	Returns the RPC connection number associated with *hostname*. The UVNet REMOTE.B interface program uses this number. If there is an internal error adding *hostname*, or if RPC has not yet opened, 0 returns. If the RPC connection was opened but is now closed, –1 returns.
1202, *hostname*	Returns the timeout associated with *hostname*. If there is no timeout associated with *hostname*, 0 returns.
1203	Returns the last RPC connection error number. This number is in the range 81000 through 81999. 81015 indicates that a timeout occurred. These error numbers correspond to error messages in the SYS.MESSAGE file.
1210	Changes a string of 4 characters (IEEE float) to a double (number).
1300	Set to "TRUE" if uvadm is installed. Otherwise it is set to "FALSE."
1301	User name.
1302	Returns information for the !GET.USERS function call in UniVerse BASIC.
1401	Returns all local user accounts on the local machine.
1402	Returns all global users accounts on the domain (MSWIN).
1403	Returns all local group accounts on the system.
1999	Assigns a delay per session to exclusive read locks (READU).
3001-3005	Assigns a value database wide to the UniVerse performance counter, visible through XAdmin (5 counters).
4000	Mode (HP only).
4001	Class (HP only).
4002	Prompt (HP only).
9000	R+R timestamp set up features.
9001	Returns the call stack.
9002	Changes the GtatolTrunc flag.
9003	MCT conversion.
9004	TCL CASe support.
9005	For the XTOOLSUB subroutine, gets the Driver list.
9006	For the XTOOLSUB subroutine, function with LOGTO.
9007	The MAXRLOCK value.
9010	The database type.
9012	Returns 1 if the client access is from InterCall, UniObjects, or other client tools; otherwise 0.
9013	Returns the hostname.

Examples

The first example returns the number of lines left to print on a page, with the maximum defined by the `TERM` command. The second example returns the current page number.

Source lines	Program output
Q=4 PRINT 'SYSTEM(Q)',SYSTEM(Q)	SYSTEM(Q) 20
PRINT 'X=',SYSTEM(5)	X= 0

The next example sets a 30-second timeout for the UVNet connection to the system ORION:

```
TIMEOUT SYSTEM(1200, "ORION"), 30
```

TABSTOP statement

Use the TABSTOP statement to set the current tabstop width for PRINT statement. The initial tabstop setting is 10.

If *expression* evaluates to the null value, the TABSTOP statement fails and the program terminates with a run-time error message.

Syntax

TABSTOP *expression*

Example

```
1A="FIRST"
B="LAST"
PRINT A,B
TABSTOP 15
PRINT A,B
```

This is the program output:

```
FIRST   LAST
 FIRST              LAST
```

TAN function

Use the `TAN` function to return the trigonometric tangent of *expression*. *expression* represents an angle expressed in degrees.

Trying to take the tangent of a right angle results in a warning message, and a return value of 0. Numbers greater than 1E17 produce a warning message, and 0 is returned. If *expression* evaluates to the null value, null is returned.

Syntax

TAN (*expression*)

Example

```
PRINT TAN(45)
```

This is the program output:

```
1
```

TANH function

Use the TANH function to return the hyperbolic tangent of *expression*. *expression* must be numeric and represents the angle expressed in degrees. If *expression* evaluates to the null value, null is returned.

Syntax

TANH (*expression*)

Example

```
PRINT TANH(45)
```

This is the program output:

```
1
```

TERMINFO function

Use the TERMINFO function to access the device-independent terminal handler string defined for the current terminal type. The TERMINFO function returns a dynamic array containing the terminal characteristics for the terminal type set by TERM or SET.TERM.TYPE.

Syntax

TERMINFO (*argument*)

argument can be 0 or 1, depending on whether the terminal characteristics are returned as stored, or converted to printable form. If *argument* is 0, the function returns the terminal characteristics in the form usable by BASIC applications for device-independent terminal handling with the TPARM function and the TPRINT statement. If *argument* is 1, the function returns characteristics in *terminfo* source format. Boolean values are returned as Y = true and N = false. The *terminfo* files contain many unprintable control characters that may adversely affect your terminal.

If *argument* evaluates to the null value, the TERMINFO function fails and the program terminates with a run-time error message.

The easiest way to access the *terminfo* characteristics is by including the BASIC file UNIVERSE.INCLUDE TERMINFO in your program. The syntax is:

```
$INCLUDE UNIVERSE.INCLUDE TERMINFO
```

The file contains lines that equate each dynamic array element returned by TERMINFO with a name, so that each element can be easily accessed in your program. Once this file has been included in your program, you can use the defined names to access terminal characteristics. The following table lists the contents of this file:

terminfo contents	
terminfo$ = terminfo(0)	
EQU TERMINAL.NAME	TO terminfo$<1>
EQU COLUMNS	TO terminfo$<2>
EQU LINES	TO terminfo$<3>
EQU CARRIAGE.RETURN	TO terminfo$<4>

terminfo contents	
EQU LINE.FEED	TO terminfo$<5>
EQU NEWLINE	TO terminfo$<6>
EQU BACKSPACE	TO terminfo$<7>
EQU BELL	TO terminfo$<8>
EQU SCREEN.FLASH	TO terminfo$<9>
EQU PADDING.CHARACTER	TO terminfo$<10>
EQU PAD.BAUD.RATE	TO terminfo$<11>
EQU HARD.COPY	TO terminfo$<12>
EQU OVERSTRIKES	TO terminfo$<13>
EQU ERASES.OVERSTRIKE	TO terminfo$<14>
EQU AUTOMATIC.RIGHT.MARGIN	TO terminfo$<15>
EQU RIGHT.MARGIN.EATS.NEWLINE	TO terminfo$<16>
EQU AUTOMATIC.LEFT.MARGIN	TO terminfo$<17>
EQU UNABLE.TO.PRINT.TILDE	TO terminfo$<18>
EQU ERASE.SCREEN	TO terminfo$<19>
EQU ERASE.TO.END.OF.SCREEN	TO terminfo$<20>
EQU ERASE.TO.BEGINNING.OF.SCREEN	TO terminfo$<21>
EQU ERASE.LINE	TO terminfo$<22>
EQU ERASE.TO.END.OF.LINE	TO terminfo$<23>
EQU ERASE.TO.BEGINNING.OF.LINE	TO terminfo$<24>
EQU ERASE.CHARACTERS	TO terminfo$<25>
EQU MOVE.CURSOR.TO.ADDRESS	TO terminfo$<26>
EQU MOVE.CURSOR.TO.COLUMN	TO terminfo$<27>
EQU MOVE.CURSOR.TO.ROW	TO terminfo$<28>
EQU MOVE.CURSOR.RIGHT	TO terminfo$<29>
EQU MOVE.CURSOR.LEFT	TO terminfo$<30>
EQU MOVE.CURSOR.DOWN	TO terminfo$<31>
EQU MOVE.CURSOR.UP	TO terminfo$<32>
EQU MOVE.CURSOR.RIGHT.PARM	TO terminfo$<33>
EQU MOVE.CURSOR.LEFT.PARM	TO terminfo$<34>
EQU MOVE.CURSOR.DOWN.PARM	TO terminfo$<35>
EQU MOVE.CURSOR.UP.PARM	TO terminfo$<36>
EQU MOVE.CURSOR.TO.HOME	TO terminfo$<37>
EQU MOVE.CURSOR.TO.LAST.LINE	TO terminfo$<38>
EQU CURSOR.SAVE	TO terminfo$<39>
EQU CURSOR.RESTORE	TO terminfo$<40>
EQU INSERT.CHARACTER	TO terminfo$<41>
EQU INSERT.CHARACTER.PARM	TO terminfo$<42>
EQU INSERT.MODE.BEGIN	TO terminfo$<43>
EQU INSERT.MODE.END	TO terminfo$<44>
EQU INSERT.PAD	TO terminfo$<45>
EQU MOVE.INSERT.MODE	TO terminfo$<46>
EQU INSERT.NULL.SPECIAL	TO terminfo$<47>

terminfo contents	
EQU DELETE.CHARACTER	TO terminfo$<48>
EQU DELETE.CHARACTER.PARM	TO terminfo$<49>
EQU INSERT.LINE	TO terminfo$<50>
EQU INSERT.LINE.PARM	TO terminfo$<51>
EQU DELETE.LINE	TO terminfo$<52>
EQU DELETE.LINE.PARM	TO terminfo$<53>
EQU SCROLL.UP	TO terminfo$<54>
EQU SCROLL.UP.PARM	TO terminfo$<55>
EQU SCROLL.DOWN	TOterminfo$<56>
EQU SCROLL.DOWN.PARM	TOterminfo$<57>
EQU CHANGE.SCROLL.REGION	TOterminfo$<58>
EQU SCROLL.MODE.END	TOterminfo$<59>
EQU SCROLL.MODE.BEGIN	TOterminfo$<60>
EQU VIDEO.NORMAL	TO terminfo$<61>
EQU VIDEO.REVERSE	TO terminfo$<62>
EQU VIDEO.BLINK	TO terminfo$<63>
EQU VIDEO.UNDERLINE	TO terminfo$<64>
EQU VIDEO.DIM	TO terminfo$<65>
EQU VIDEO.BOLD	TO terminfo$<66>
EQU VIDEO.BLANK	TO terminfo$<67>
EQU VIDEO.STANDOUT	TO terminfo$<68>
EQU VIDEO.SPACES	TO terminfo$<69>
EQU MOVE.VIDEO.MODE	TO terminfo$<70>
EQU TAB	TO terminfo$<71>
EQU BACK.TAB	TO terminfo$<72>
EQU TAB.STOP.SET	TO terminfo$<73>
EQU TAB.STOP.CLEAR	TO terminfo$<74>
EQU CLEAR.ALL.TAB.STOPS	TO terminfo$<75>
EQU TAB.STOP.INITIAL	TO terminfo$<76>
EQU WRITE.PROTECT.BEGIN	TO terminfo$<77>
EQU WRITE.PROTECT.END	TO terminfo$<78>
EQU SCREEN.PROTECT.BEGIN	TO terminfo$<79>
EQU SCREEN.PROTECT.END	TO terminfo$<80>
EQU WRITE.PROTECT.COLUMN	TO terminfo$<81>
EQU PROTECT.VIDEO.NORMAL	TO terminfo$<82>
EQU PROTECT.VIDEO.REVERSE	TO terminfo$<83>
EQU PROTECT.VIDEO.BLINK	TO terminfo$<84>
EQU PROTECT.VIDEO.UNDERLINE	TO terminfo$<85>
EQU PROTECT.VIDEO.DIM	TO terminfo$<86>
EQU PROTECT.VIDEO.BOLD	TO terminfo$<87>
EQU PROTECT.VIDEO.BLANK	TO terminfo$<88>
EQU PROTECT.VIDEO.STANDOUT	TO terminfo$<89>
EQU BLOCK.MODE.BEGIN	TO terminfo$<90>

terminfo contents	
EQU BLOCK.MODE.END	TO terminfo$<91>
EQU SEND.LINE.ALL	TO terminfo$<92>
EQU SEND.LINE.UNPROTECTED	TO terminfo$<93>
EQU SEND.PAGE.ALL	TO terminfo$<94>
EQU SEND.PAGE.UNPROTECTED	TO terminfo$<95>
EQU SEND.MESSAGE.ALL	TO terminfo$<96>
EQU SEND.MESSAGE.UNPROTECTED	TO terminfo$<97>
EQU TERMINATE.FIELD	TO terminfo$<98>
EQU TERMINATE.LINE	TO terminfo$<99>
EQU TERMINATE.PAGE	TO terminfo$<100>
EQU STORE.START.OF.MESSAGE	TO terminfo$<101>
EQU STORE.END.OF.MESSAGE	TO terminfo$<102>
EQU LINEDRAW.BEGIN	TO terminfo$<103>
EQU LINEDRAW.END	TO terminfo$<104>
EQU MOVE.LINEDRAW.MODE	TO terminfo$<105>
EQU LINEDRAW.CHARACTER	TO terminfo$<106>
EQU LINEDRAW.UPPER.LEFT.CORNER	TO terminfo$<107>
EQU LINEDRAW.UPPER.RIGHT.CORNER	TO terminfo$<108>
EQU LINEDRAW.LOWER.LEFT.CORNER	TO terminfo$<109>
EQU LINEDRAW.LOWER.RIGHT.CORNER	TO terminfo$<110>
EQU LINEDRAW.LEFT.VERTICAL	TO terminfo$<111>
EQU LINEDRAW.CENTER.VERTICAL	TO terminfo$<112>
EQU LINEDRAW.RIGHT.VERTICAL	TO terminfo$<113>
EQU LINEDRAW.UPPER.HORIZONTAL	TO terminfo$<114>
EQU LINEDRAW.CENTER.HORIZONTAL	TO terminfo$<115>
EQU LINEDRAW.LOWER.HORIZONTAL	TO terminfo$<116>
EQU LINEDRAW.UPPER.TEE	TO terminfo$<117>
EQU LINEDRAW.LOWER.TEE	TO terminfo$<118>
EQU LINEDRAW.LEFT.TEE	TO terminfo$<119>
EQU LINEDRAW.RIGHT.TEE	TO terminfo$<120>
EQU LINEDRAW.CROSS	TO terminfo$<121>
EQU CURSOR.NORMAL	TO terminfo$<122>
EQU CURSOR.VISIBLE	TO terminfo$<123>
EQU CURSOR.INVISIBLE	TO terminfo$<124>
EQU SCREEN.VIDEO.ON	TO terminfo$<125>
EQU SCREEN.VIDEO.OFF	TO terminfo$<126>
EQU KEYCLICK.ON	TO terminfo$<127>
EQU KEYCLICK.OFF	TO terminfo$<128>
EQU KEYBOARD.LOCK.ON	TO terminfo$<129>
EQU KEYBOARD.LOCK.OFF	TO terminfo$<130>
EQU MONITOR.MODE.ON	TO terminfo$<131>
EQU MONITOR.MODE.OFF	TO terminfo$<132>
EQU PRINT.SCREEN	TO terminfo$<133>

terminfo contents	
EQU PRINT.MODE.BEGIN	TO terminfo$<134>
EQU PRINT.MODE.END	TO terminfo$<135>
EQU HAS.STATUS.LINE	TO terminfo$<136>
EQU STATUS.LINE.WIDTH	TO terminfo$<137>
EQU STATUS.LINE.BEGIN	TO terminfo$<138>
EQU STATUS.LINE.END	TO terminfo$<139>
EQU STATUS.LINE.DISABLE	TO terminfo$<140>
EQU HAS.FUNCTION.LINE	TO terminfo$<141>
EQU FUNCTION.LINE.BEGIN	TO terminfo$<142>
EQU FUNCTION.LINE.END	TO terminfo$<143>
EQU KEY.BACKSPACE	TO terminfo$<144>
EQU KEY.MOVE.CURSOR.RIGHT	TO terminfo$<145>
EQU KEY.MOVE.CURSOR.LEFT	TO terminfo$<146>
EQU KEY.MOVE.CURSOR.DOWN	TO terminfo$<147>
EQU KEY.MOVE.CURSOR.UP	TO terminfo$<148>
EQU KEY.MOVE.CURSOR.TO.HOME	TO terminfo$<149>
EQU KEY.MOVE.CURSOR.TO.LAST.LINE	TO terminfo$<150>
EQU KEY.INSERT.CHARACTER	TO terminfo$<151>
EQU KEY.INSERT.MODE.ON	TO terminfo$<152>
EQU KEY.INSERT.MODE.END	TO terminfo$<153>
EQU KEY.INSERT.MODE.TOGGLE	TO terminfo$<154>
EQU KEY.DELETE.CHARACTER	TO terminfo$<155>
EQU KEY.INSERT.LINE	TO terminfo$<156>
EQU KEY.DELETE.LINE	TO terminfo$<157>
EQU KEY.ERASE.SCREEN	TO terminfo$<158>
EQU KEY.ERASE.END.OF.LINE	TO terminfo$<159>
EQU KEY.ERASE.END.OF.SCREEN	TO terminfo$<160>
EQU KEY.BACK.TAB	TO terminfo$<161>
EQU KEY.TAB.STOP.SET	TO terminfo$<162>
EQU KEY.TAB.STOP.CLEAR	TO terminfo$<163>
EQU KEY.TAB.STOP.CLEAR.ALL	TO terminfo$<164>
EQU KEY.NEXT.PAGE	TO terminfo$<165>
EQU KEY.PREVIOUS.PAGE	TO terminfo$<166>
EQU KEY.SCROLL.UP	TO terminfo$<167>
EQU KEY.SCROLL.DOWN	TO terminfo$<168>
EQU KEY.SEND.DATA	TO terminfo$<169>
EQU KEY.PRINT	TO terminfo$<170>
EQU KEY.FUNCTION.0	TO terminfo$<171>
EQU KEY.FUNCTION.1	TO terminfo$<172>
EQU KEY.FUNCTION.2	TO terminfo$<173>
EQU KEY.FUNCTION.3	TO terminfo$<174>
EQU KEY.FUNCTION.4	TO terminfo$<175>
EQU KEY.FUNCTION.5	TO terminfo$<176>

terminfo contents	
EQU KEY.FUNCTION.6	TO terminfo$<177>
EQU KEY.FUNCTION.7	TO terminfo$<178>
EQU KEY.FUNCTION.8	TO terminfo$<179>
EQU KEY.FUNCTION.9	TO terminfo$<180>
EQU KEY.FUNCTION.10	TO terminfo$<181>
EQU KEY.FUNCTION.11	TO terminfo$<182>
EQU KEY.FUNCTION.12	TO terminfo$<183>
EQU KEY.FUNCTION.13	TO terminfo$<184>
EQU KEY.FUNCTION.14	TO terminfo$<185>
EQU KEY.FUNCTION.15	TO terminfo$<186>
EQU KEY.FUNCTION.16	TO terminfo$<187>
EQU LABEL.KEY.FUNCTION.0	TO terminfo$<188>
EQU LABEL.KEY.FUNCTION.1	TO terminfo$<189>
EQU LABEL.KEY.FUNCTION.2	TO terminfo$<190>
EQU LABEL.KEY.FUNCTION.3	TO terminfo$<191>
EQU LABEL.KEY.FUNCTION.4	TO terminfo$<192>
EQU LABEL.KEY.FUNCTION.5	TO terminfo$<193>
EQU LABEL.KEY.FUNCTION.6	TO terminfo$<194>
EQU LABEL.KEY.FUNCTION.7	TO terminfo$<195>
EQU LABEL.KEY.FUNCTION.8	TO terminfo$<196>
EQU LABEL.KEY.FUNCTION.9	TO terminfo$<197>
EQU LABEL.KEY.FUNCTION.10	TO terminfo$<198>
EQU LABEL.KEY.FUNCTION.11	TO terminfo$<199>
EQU LABEL.KEY.FUNCTION.12	TO terminfo$<200>
EQU LABEL.KEY.FUNCTION.13	TO terminfo$<201>
EQU LABEL.KEY.FUNCTION.14	TO terminfo$<202>
EQU LABEL.KEY.FUNCTION.15	TO terminfo$<203>
EQU LABEL.KEY.FUNCTION.16	TO terminfo$<204>
EQU KEYEDIT.FUNCTION	TO terminfo$<205>
EQU KEYEDIT.ESCAPE	TO terminfo$<206>
EQU KEYEDIT.EXIT	TO terminfo$<207>
EQU KEYEDIT.BACKSPACE	TO terminfo$<208>
EQU KEYEDIT.MOVE.BACKWARD	TO terminfo$<209>
EQU KEYEDIT.MOVE.FORWARD	TO terminfo$<210>
EQU KEYEDIT.INSERT.CHARACTER	TO terminfo$<211>
EQU KEYEDIT.INSERT.MODE.BEGIN	TO terminfo$<212>
EQU KEYEDIT.INSERT.MODE.END	TO terminfo$<213>
EQU KEYEDIT.INSERT.MODE.TOGGLE	TO terminfo$<214>
EQU KEYEDIT.DELETE.CHARACTER	TO terminfo$<215>
EQU KEYEDIT.ERASE.END.OF.FIELD	TO terminfo$<216>
EQU KEYEDIT.ERASE.FIELD	TO terminfo$<217>
EQU AT.NEGATIVE.1	TO terminfo$<218>
EQU AT.NEGATIVE.2	TO terminfo$<219>

terminfo contents	
EQU AT.NEGATIVE.3	TO terminfo$<220>
EQU AT.NEGATIVE.4	TO terminfo$<221>
EQU AT.NEGATIVE.5	TO terminfo$<222>
EQU AT.NEGATIVE.6	TO terminfo$<223>
EQU AT.NEGATIVE.7	TO terminfo$<224>
EQU AT.NEGATIVE.8	TO terminfo$<225>
EQU AT.NEGATIVE.9	TO terminfo$<226>
EQU AT.NEGATIVE.10	TO terminfo$<227>
EQU AT.NEGATIVE.11	TO terminfo$<228>
EQU AT.NEGATIVE.12	TO terminfo$<229>
EQU AT.NEGATIVE.13	TO terminfo$<230>
EQU AT.NEGATIVE.14	TO terminfo$<231>
EQU AT.NEGATIVE.15	TO terminfo$<232>
EQU AT.NEGATIVE.16	TO terminfo$<233>
EQU AT.NEGATIVE.17	TO terminfo$<234>
EQU AT.NEGATIVE.18	TO terminfo$<235>
EQU AT.NEGATIVE.19	TO terminfo$<236>
EQU AT.NEGATIVE.20	TO terminfo$<237>
EQU AT.NEGATIVE.21	TO terminfo$<238>
EQU AT.NEGATIVE.22	TO terminfo$<239>
EQU AT.NEGATIVE.23	TO terminfo$<240>
EQU AT.NEGATIVE.24	TO terminfo$<241>
EQU AT.NEGATIVE.25	TO terminfo$<242>
EQU AT.NEGATIVE.26	TO terminfo$<243>
EQU AT.NEGATIVE.27	TO terminfo$<244>
EQU AT.NEGATIVE.28	TO terminfo$<245>
EQU AT.NEGATIVE.29	TO terminfo$<246>
EQU AT.NEGATIVE.30	TO terminfo$<247>
EQU AT.NEGATIVE.31	TO terminfo$<248>
EQU AT.NEGATIVE.32	TO terminfo$<249>
EQU AT.NEGATIVE.33	TO terminfo$<250>
EQU AT.NEGATIVE.34	TO terminfo$<251>
EQU AT.NEGATIVE.35	TO terminfo$<252>
EQU AT.NEGATIVE.36	TO terminfo$<253>
EQU AT.NEGATIVE.37	TO terminfo$<254>
EQU AT.NEGATIVE.38	TO terminfo$<255>
EQU AT.NEGATIVE.39	TO terminfo$<256>
EQU AT.NEGATIVE.40	TO terminfo$<257>
EQU AT.NEGATIVE.41	TO terminfo$<258>
EQU AT.NEGATIVE.42	TO terminfo$<259>
EQU AT.NEGATIVE.43	TO terminfo$<260>
EQU AT.NEGATIVE.44	TO terminfo$<261>
EQU AT.NEGATIVE.45	TO terminfo$<262>

terminfo contents	
EQU AT.NEGATIVE.46	TO terminfo$<263>
EQU AT.NEGATIVE.47	TO terminfo$<264>
EQU AT.NEGATIVE.48	TO terminfo$<265>
EQU AT.NEGATIVE.49	TO terminfo$<266>
EQU AT.NEGATIVE.50	TO terminfo$<267>
EQU AT.NEGATIVE.51	TO terminfo$<268>
EQU AT.NEGATIVE.52	TO terminfo$<269>
EQU AT.NEGATIVE.53	TO terminfo$<270>
EQU AT.NEGATIVE.54	TO terminfo$<271>
EQU AT.NEGATIVE.55	TO terminfo$<272>
EQU AT.NEGATIVE.56	TO terminfo$<273>
EQU AT.NEGATIVE.57	TO terminfo$<274>
EQU AT.NEGATIVE.58	TO terminfo$<275>
EQU AT.NEGATIVE.59	TO terminfo$<276>
EQU AT.NEGATIVE.60	TO terminfo$<277>
EQU AT.NEGATIVE.61	TO terminfo$<278>
EQU AT.NEGATIVE.62	TO terminfo$<279>
EQU AT.NEGATIVE.63	TO terminfo$<280>
EQU AT.NEGATIVE.64	TO terminfo$<281>
EQU AT.NEGATIVE.65	TO terminfo$<282>
EQU AT.NEGATIVE.66	TO terminfo$<283>
EQUAT.NEGATIVE.67	TO terminfo$<284>
EQUAT.NEGATIVE.68	TO terminfo$<285>
EQU AT.NEGATIVE.69	TO terminfo$<286>
EQU AT.NEGATIVE.70	TO terminfo$<287>
EQU AT.NEGATIVE.71	TO terminfo$<288>
EQU AT.NEGATIVE.72	TO terminfo$<289>
EQU AT.NEGATIVE.73	TO terminfo$<290>
EQU AT.NEGATIVE.74	TO terminfo$<291>
EQU AT.NEGATIVE.75	TO terminfo$<292>
EQU AT.NEGATIVE.76	TO terminfo$<293>
EQU AT.NEGATIVE.77	TO terminfo$<294>
EQUAT.NEGATIVE.78	TO terminfo$<295>
EQU AT.NEGATIVE.79	TO terminfo$<296>
EQU AT.NEGATIVE.80	TO terminfo$<297>
EQU AT.NEGATIVE.81	TO terminfo$<298>
EQU AT.NEGATIVE.82	TO terminfo$<299>
EQU AT.NEGATIVE.83	TO terminfo$<300>
EQU AT.NEGATIVE.84	TO terminfo$<301>
EQU AT.NEGATIVE.85	TO terminfo$<302>
EQU AT.NEGATIVE.86	TO terminfo$<303>
EQU AT.NEGATIVE.87	TO terminfo$<304>
EQU AT.NEGATIVE.88	TO terminfo$<305>

terminfo contents	
EQU AT.NEGATIVE.89	TO terminfo$<306>
EQU AT.NEGATIVE.90	TO terminfo$<307>
EQU AT.NEGATIVE.91	TO terminfo$<308>
EQU AT.NEGATIVE.92	TO terminfo$<309>
EQU AT.NEGATIVE.93	TO terminfo$<310>
EQU AT.NEGATIVE.94	TO terminfo$<311>
EQU AT.NEGATIVE.95	TO terminfo$<312>
EQU AT.NEGATIVE.96	TO terminfo$<313>
EQU AT.NEGATIVE.97	TO terminfo$<314>
EQU AT.NEGATIVE.98	TO terminfo$<315>
EQU AT.NEGATIVE.99	TO terminfo$<316>
EQU AT.NEGATIVE.100	TO terminfo$<317>
EQU AT.NEGATIVE.101	TO terminfo$<318>
EQU AT.NEGATIVE.102	TO terminfo$<319>
EQU AT.NEGATIVE.103	TO terminfo$<320>
EQU AT.NEGATIVE.104	TO terminfo$<321>
EQU AT.NEGATIVE.105	TO terminfo$<322>
EQU AT.NEGATIVE.106	TO terminfo$<323>
EQU AT.NEGATIVE.107	TO terminfo$<324>
EQU AT.NEGATIVE.108	TO terminfo$<325>
EQU AT.NEGATIVE.109	TO terminfo$<326>
EQU AT.NEGATIVE.110	TO terminfo$<327>
EQU AT.NEGATIVE.111	TO terminfo$<328>
EQU AT.NEGATIVE.112	TO terminfo$<329>
EQU AT.NEGATIVE.113	TO terminfo$<330>
EQU AT.NEGATIVE.114	TO terminfo$<331>
EQU AT.NEGATIVE.115	TO terminfo$<332>
EQU AT.NEGATIVE.116	TO terminfo$<333>
EQU AT.NEGATIVE.117	TO terminfo$<334>
EQU AT.NEGATIVE.118	TO terminfo$<335>
EQU AT.NEGATIVE.119	TO terminfo$<336>
EQU AT.NEGATIVE.120	TO terminfo$<337>
EQU AT.NEGATIVE.121	TO terminfo$<338>
EQU AT.NEGATIVE.122	TO terminfo$<339>
EQU AT.NEGATIVE.123	TO terminfo$<340>
EQU AT.NEGATIVE.124	TO terminfo$<341>
EQU AT.NEGATIVE.125	TO terminfo$<342>
EQU AT.NEGATIVE.126	TO terminfo$<343>
EQU AT.NEGATIVE.127	TO terminfo$<344>
EQU AT.NEGATIVE.128	TO terminfo$<345>
EQU DBLE.LDRAW.UP.LEFT.CORNER	TO terminfo$<379>
EQU DBLE.LDRAW.UP.RIGHT.CORNER	TO terminfo$<380>
EQU DBLE.LDRAW.LO.LEFT.CORNER	TO terminfo$<381>

terminfo contents	
EQU DBLE.LDRAW.LO.RIGHT.CORNER	TO terminfo$<382>
EQU DBLE.LDRAW.HORIZ	TO terminfo$<383>
EQU DBLE.LDRAW.VERT	TO terminfo$<384>
EQU DBLE.LDRAW.UP.TEE	TO terminfo$<385>
EQU DBLE.LDRAW.LO.TEE	TO terminfo$<386>
EQU DBLE.LDRAW.LEFT.TEE	TO terminfo$<387>
EQU DBLE.LDRAW.RIGHT.TEE	TO terminfo$<388>
EQU DBLE.LDRAW.CROSS	TO terminfo$<389>
EQU LDRAW.LEFT.TEE.DBLE.HORIZ	TO terminfo$<390>
EQU LDRAW.LEFT.TEE.DBLE.VERT	TO terminfo$<391>
EQU LDRAW.RIGHT.TEE.DBLE.HORIZ	TO terminfo$<392>
EQU LDRAW.RIGHT.TEE.DBLE.VERT	TO terminfo$<393>
EQU LDRAW.LOWER.TEE.DBLE.HORIZ	TO terminfo$<394>
EQU LDRAW.LOWER.TEE.DBLE.VERT	TO terminfo$<395>
EQU LDRAW.UP.TEE.DBLE.HORIZ	TO terminfo$<396>
EQU LDRAW.UP.TEE.DBLE.VERT	TO terminfo$<397>
EQU LDRAW.UP.LEFT.CORNER.DBLE.HORIZ	TO terminfo$<398>
EQU LDRAW.UP.LEFT.CORNER.DBLE.VERT	TO terminfo$<399>
EQU LDRAW.UP.RIGHT.CORNER.DBLE.HORIZ	TO terminfo$<400>
EQU LDRAW.UP.RIGHT.CORNER.DBLE.VERT	TO terminfo$<401>
EQU LDRAW.LO.LEFT.CORNER.DBLE.HORIZ	TO terminfo$<402>
EQU LDRAW.LO.LEFT.CORNER.DBLE.VERT	TO terminfo$<403>
EQU LDRAW.LO.RIGHT.CORNER.DBLE.HORIZ	TO terminfo$<404>
EQU LDRAW.LO.RIGHT.CORNER.DBLE.VERT	TO terminfo$<405>
EQU LDRAW.CROSS.DBLE.HORIZ	TO terminfo$<406>
EQU LDRAW.CROSS.DBLE.VERT	TO terminfo$<407>
EQU NO.ESC.CTLC	TO terminfo$<408>
EQU CEOL.STANDOUT.GLITCH	TO terminfo$<409>
EQU GENERIC.TYPE	TO terminfo$<410>
EQU HAS.META.KEY	TO terminfo$<411>
EQU MEMORY.ABOVE	TO terminfo$<412>
EQU MEMORY.BELOW	TO terminfo$<413>
EQU STATUS.LINE.ESC.OK	TO terminfo$<414>
EQU DEST.TABS.MAGIC.SMSO	TO terminfo$<415>
EQU TRANSPARENT.UNDERLINE	TO terminfo$<416>
EQU XON.XOFF	TO terminfo$<417>
EQU NEEDS.XON.XOFF	TO terminfo$<418>
EQU PRTR.SILENT	TO terminfo$<419>
EQU HARD.CURSOR	TO terminfo$<420>
EQU NON.REV.RMCUP	TO terminfo$<421>
EQU NO.PAD.CHAR	TO terminfo$<422>
EQU LINES.OF.MEMORY	TO terminfo$<423>
EQU VIRTUAL.TERMINAL	TO terminfo$<424>

terminfo contents	
EQU NUM.LABELS	TO terminfo$<425>
EQU LABEL.HEIGHT	TO terminfo$<426>
EQU LABEL.WIDTH	TO terminfo$<427>
EQU LINE.ATTRIBUTE	TO terminfo$<428>
EQU COMMAND.CHARACTER	TO terminfo$<429>
EQU CURSOR.MEM.ADDRESS	TO terminfo$<430>
EQU DOWN.HALF.LINE	TO terminfo$<431>
EQU ENTER.CA.MODE	TO terminfo$<432>
EQU ENTER.DELETE.MODE	TO terminfo$<433>
EQU ENTER.PROTECTED.MODE	TO terminfo$<434>
EQU EXIT.ATTRIBUTE.MODE	TO terminfo$<435>
EQU EXIT.CA.MODE	TO terminfo$<436>
EQU EXIT.DELETE.MODE	TO terminfo$<437>
EQU EXIT.STANDOUT.MODE	TO terminfo$<438>
EQU EXIT.UNDERLINE.MODE	TO terminfo$<439>
EQU FORM.FEED	TO terminfo$<440>
EQU INIT.1STRING	TO terminfo$<441>
EQU INIT.2STRING	TO terminfo$<442>
EQU INIT.3STRING	TO terminfo$<443>
EQU INIT.FILE	TO terminfo$<444>
EQU INS.PREFIX	TO terminfo$<445>
EQU KEY.IC	TO terminfo$<446>
EQU KEYPAD.LOCAL	TO terminfo$<447>
EQU KEYPAD.XMIT	TO terminfo$<448>
EQU META.OFF	TO terminfo$<449>
EQU META.ON	TO terminfo$<450>
EQU PKEY.KEY	TO terminfo$<451>
EQU PKEY.LOCAL	TO terminfo$<452>
EQU PKEY.XMIT	TO terminfo$<453>
EQU REPEAT.CHAR	TO terminfo$<454>
EQU RESET.1STRING	TO terminfo$<455>
EQU RESET.2STRING	TO terminfo$<456>
EQU RESET.3STRING	TO terminfo$<457>
EQU RESET.FILE	TO terminfo$<458>
EQU SET.ATTRIBUTES	TO terminfo$<459>
EQU SET.WINDOW	TO terminfo$<460>
EQU UNDERLINE.CHAR	TO terminfo$<461>
EQU UP.HALF.LINE	TO terminfo$<462>
EQU INIT.PROG	TO terminfo$<463>
EQU KEY.A1	TO terminfo$<464>
EQU KEY.A3	TO terminfo$<465>
EQU KEY.B2	TO terminfo$<466>
EQU KEY.C1	TO terminfo$<467>

terminfo contents	
EQU KEY.C3	TO terminfo$<468>
EQU PRTR.NON	TO terminfo$<469>
EQU CHAR.PADDING	TO terminfo$<470>
EQU LINEDRAW.CHARS	TO terminfo$<471>
EQU PLAB.NORM	TO terminfo$<472>
EQU ENTER.XON.MODE	TO terminfo$<473>
EQU EXIT.XON.MODE	TO terminfo$<474>
EQU ENTER.AM.MODE	TO terminfo$<475>
EQU EXIT.AM.MODE	TO terminfo$<476>
EQU XON.CHARACTER	TO terminfo$<477>
EQU XOFF.CHARACTER	TO terminfo$<478>
EQU ENABLE.LINEDRAW	TO terminfo$<479>
EQU LABEL.ON	TO terminfo$<480>
EQU LABEL.OFF	TO terminfo$<481>
EQU KEY.BEG	TO terminfo$<482>
EQU KEY.CANCEL	TO terminfo$<483>
EQU KEY.CLOSE	TO terminfo$<484>
EQU KEY.COMMAND	TO terminfo$<485>
EQU KEY.COPY	TO terminfo$<486>
EQU KEY.CREATE	TO terminfo$<487>
EQU KEY.END	TO terminfo$<488>
EQU KEY.ENTER	TO terminfo$<489>
EQU KEY.EXIT	TO terminfo$<490>
EQU KEY.FIND	TO terminfo$<491>
EQU KEY.HELP	TO terminfo$<492>
EQU KEY.MARK	TO terminfo$<493>
EQU KEY.MESSAGE	TO terminfo$<494>
EQU KEY.MOVE	TO terminfo$<495>
EQU KEY.NEXT	TO terminfo$<496>
EQU KEY.OPEN	TO terminfo$<497>
EQU KEY.OPTIONS	TO terminfo$<498>
EQU KEY.PREVIOUS	TO terminfo$<499>
EQU KEY.REDO	TO terminfo$<500>
EQU KEY.REFERENCE	TO terminfo$<501>
EQU KEY.REFRESH	TO terminfo$<502>
EQU KEY.REPLACE	TO terminfo$<503>
EQU KEY.RESTART	TO terminfo$<504>
EQU KEY.RESUME	TO terminfo$<505>
EQU KEY.SAVE	TO terminfo$<506>
EQU KEY.SUSPEND	TO terminfo$<507>
EQU KEY.UNDO	TO terminfo$<508>
EQU KEY.SBEG	TO terminfo$<509>
EQU KEY.SCANCEL	TO terminfo$<510>

terminfo contents	
EQU KEY.SCOMMAND	TO terminfo$<511>
EQU KEY.SCOPY	TO terminfo$<512>
EQU KEY.SCREATE	TO terminfo$<513>
EQU KEY.SDC	TO terminfo$<514>
EQU KEY.SDL	TO terminfo$<515>
EQU KEY.SELECT	TO terminfo$<516>
EQU KEY.SEND	TO terminfo$<517>
EQU KEY.SEOL	TO terminfo$<518>
EQU KEY.SEXIT	TO terminfo$<519>
EQU KEY.SFIND	TO terminfo$<520>
EQU KEY.SHELP	TO terminfo$<521>
EQU KEY.SHOME	TO terminfo$<522>
EQU KEY.SIC	TO terminfo$<523>
EQU KEY.SLEFT	TO terminfo$<524>
EQU KEY.SMESSAGE	TO terminfo$<525>
EQU KEY.SMOVE	TO terminfo$<526>
EQU KEY.SNEXT	TO terminfo$<527>
EQU KEY.SOPTIONS	TO terminfo$<528>
EQU KEY.SPREVIOUS	TO terminfo$<529>
EQU KEY.SPRINT	TO terminfo$<530>
EQU KEY.SREDO	TO terminfo$<531>
EQU KEY.SREPLACE	TO terminfo$<532>
EQU KEY.SRIGHT	TO terminfo$<533>
EQU KEY.SRESUM	TO terminfo$<534>
EQU KEY.SSAVE	TO terminfo$<535>
EQU KEY.SSUSPEND	TO terminfo$<536>
EQU KEY.SUNDO	TO terminfo$<537>
EQU REQ.FOR.INPUT	TO terminfo$<538>
EQU KEY.F17	TO terminfo$<539>
EQU KEY.F18	TO terminfo$<540>
EQU KEY.F19	TO terminfo$<541>
EQU KEY.F20	TO terminfo$<542>
EQU KEY.F21	TO terminfo$<543>
EQU KEY.F22	TO terminfo$<544>
EQU KEY.F23	TO terminfo$<545>
EQU KEY.F24	TO terminfo$<546>
EQU KEY.F25	TO terminfo$<547>
EQU KEY.F26	TO terminfo$<548>
EQU KEY.F27	TO terminfo$<549>
EQU KEY.F28	TO terminfo$<550>
EQU KEY.F29	TO terminfo$<551>
EQU KEY.F30	TO terminfo$<552>
EQU KEY.F31	TO terminfo$<553>

terminfo contents	
EQU KEY.F32	TO terminfo$<554>
EQU KEY.F33	TO terminfo$<555>
EQU KEY.F34	TO terminfo$<556>
EQU KEY.F35	TO terminfo$<557>
EQU KEY.F36	TO terminfo$<558>
EQU KEY.F37	TO terminfo$<559>
EQU KEY.F38	TO terminfo$<560>
EQU KEY.F39	TO terminfo$<561>
EQU KEY.F40	TO terminfo$<562>
EQU KEY.F41	TO terminfo$<563>
EQU KEY.F42	TO terminfo$<564>
EQU KEY.F43	TO terminfo$<565>
EQU KEY.F44	TO terminfo$<566>
EQU KEY.F45	TO terminfo$<567>
EQU KEY.F46	TO terminfo$<568>
EQU KEY.F47	TO terminfo$<569>
EQU KEY.F48	TO terminfo$<570>
EQU KEY.F49	TO terminfo$<571>
EQU KEY.F50	TO terminfo$<572>
EQU KEY.F51	TO terminfo$<573>
EQU KEY.F52	TO terminfo$<574>
EQU KEY.F53	TO terminfo$<575>
EQU KEY.F54	TO terminfo$<576>
EQU KEY.F55	TO terminfo$<577>
EQU KEY.F56	TO terminfo$<578>
EQU KEY.F57	TO terminfo$<579>
EQU KEY.F58	TO terminfo$<580>
EQU KEY.F59	TO terminfo$<581>
EQU KEY.F60	TO terminfo$<582>
EQU KEY.F61	TO terminfo$<583>
EQU KEY.F62	TO terminfo$<584>
EQU KEY.F63	TO terminfo$<585>
EQU CLEAR.MARGINS	TO terminfo$<586>
EQU SET.LEFT.MARGIN	TO terminfo$<587>
EQU SET.RIGHT.MARGIN	TO terminfo$<588>
EQU LABEL.KEY.FUNCTION.17	TO terminfo$<589>
EQU LABEL.KEY.FUNCTION.18	TO terminfo$<590>
EQU LABEL.KEY.FUNCTION.19	TO terminfo$<591>
EQU LABEL.KEY.FUNCTION.20	TO terminfo$<592>
EQU LABEL.KEY.FUNCTION.2	TO terminfo$<593>
EQU LABEL.KEY.FUNCTION.22	TO terminfo$<594>
EQU LABEL.KEY.FUNCTION.2	TO terminfo$<595>
EQU LABEL.KEY.FUNCTION.24	TO terminfo$<596>

terminfo contents	
EQU LABEL.KEY.FUNCTION.25	TO terminfo$<597>
EQU LABEL.KEY.FUNCTION.26	TO terminfo$<598>
EQU LABEL.KEY.FUNCTION.27	TO terminfo$<599>
EQU LABEL.KEY.FUNCTION.28	TO terminfo$<600>
EQU LABEL.KEY.FUNCTION.2	TO terminfo$<601>
EQU LABEL.KEY.FUNCTION.30	TO terminfo$<602>
EQU LABEL.KEY.FUNCTION.31	TO terminfo$<603>
EQU LABEL.KEY.FUNCTION.32	TO terminfo$<604>
EQU LABEL.KEY.FUNCTION.33	TO terminfo$<605>
EQU LABEL.KEY.FUNCTION.34	TO terminfo$<606>
EQU LABEL.KEY.FUNCTION.35	TO terminfo$<607>
EQU LABEL.KEY.FUNCTION.36	TO terminfo$<608>
EQU LABEL.KEY.FUNCTION.37	TO terminfo$<609>
EQU LABEL.KEY.FUNCTION.38	TO terminfo$<610>
EQU LABEL.KEY.FUNCTION.39	TO terminfo$<611>
EQU LABEL.KEY.FUNCTION.40	TO terminfo$<612>
EQU LABEL.KEY.FUNCTION.41	TO terminfo$<613>
EQU LABEL.KEY.FUNCTION.42	TO terminfo$<614>
EQU LABEL.KEY.FUNCTION.43	TO terminfo$<615>
EQU LABEL.KEY.FUNCTION.44	TO terminfo$<616>
EQU LABEL.KEY.FUNCTION.45	TO terminfo$<617>
EQU LABEL.KEY.FUNCTION.46	TO terminfo$<618>
EQU LABEL.KEY.FUNCTION.4	TO terminfo$<619>
EQU LABEL.KEY.FUNCTION.48	TO terminfo$<620>
EQU LABEL.KEY.FUNCTION.49	TO terminfo$<621>
EQU LABEL.KEY.FUNCTION.50S	TO terminfo$<622>
EQU LABEL.KEY.FUNCTION.51	TO terminfo$<623>
EQU LABEL.KEY.FUNCTION.52	TO terminfo$<624>
EQU LABEL.KEY.FUNCTION.53	TO terminfo$<625>
EQU LABEL.KEY.FUNCTION.54	TO terminfo$<626>
EQU LABEL.KEY.FUNCTION.55	TO terminfo$<627>
EQU LABEL.KEY.FUNCTION.56	TO terminfo$<628>
EQU LABEL.KEY.FUNCTION.57	TO terminfo$<629>
EQU LABEL.KEY.FUNCTION.58	TO terminfo$<630>
EQU LABEL.KEY.FUNCTION.59	TO terminfo$<631>
EQU LABEL.KEY.FUNCTION.60	TO terminfo$<632>
EQU LABEL.KEY.FUNCTION.61	TO terminfo$<633>
EQU LABEL.KEY.FUNCTION.62	TO terminfo$<634>
EQU LABEL.KEY.FUNCTION.63	TO terminfo$<635>

Example

```
$INCLUDE UNIVERSE.INCLUDE TERMINFO
 PRINT AT.NEGATIVE.1
```

```
PRINT "Your terminal type is":TAB:TERMINAL.NAME
```

The program output on the cleared screen is:

```
Your terminal type is icl6404|ICL 6404CG Color Video Display
```

TIME function

Use the TIME function to return a string value expressing the internal time of day. The internal time is the number of seconds that have passed since midnight to the nearest thousandth of a second (local time).

The parentheses must be used with the TIME function to distinguish it from a user-named variable called *TIME*. However, no arguments are required with the TIME function.

Syntax

TIME ()

UNIX System V

The time is returned only to the nearest whole second.

If the TIME.MILLISECOND option of the $OPTIONS statement is set, the TIME function returns the system time in whole seconds.

Example

```
PRINT TIME()
```

This is the program output:

```
40663.842
```

TIMEDATE function

Syntax

TIMEDATE ()

Use the TIMEDATE function to return the current system time and date in the following format:

hh:*mm*:*ssddmmmyyyy*

Parameter	Description
hh	Hours (based on a 24-hour clock)
mm	Minutes
ss	Seconds
dd	Day
mmm	Month
yyyy	Year

No arguments are required with the TIMEDATE function.

If you want to increase the number of spaces between the time and the date, edit the line beginning with TMD0001 in the msg.txt file in the UV account directory. This line can contain up to four hash signs (#). Each # prints a space between the time and the date.

If NLS mode is enabled, the `TIMEDATE` function uses the convention defined in the `TIMEDATE` field in the NLS.LC.TIME file for combined time and date format. Otherwise, it returns the time and date. For more information about convention records in the Time category, see the *UniVerse NLS Guide*.

Examples

```
PRINT TIMEDATE()
```

This is the program output:

```
11:19:07 18 JUN 1996
```

If the TMD0001 message contains four #s, the program output is:

```
11:19:07 18 JUN 1996
```

TIMEOUT statement

Use the TIMEOUT statement to terminate a READSEQ statement or READBLK statement if no data is read in the specified time. You can also use the TIMEOUT statement to set a time limit for a UVNet link. Use the TTYGET and TTYSET statements to set a timeout value for a file open on a serial communications port.

The TIMEOUT statement is not supported on Windows NT.

Syntax

TIMEOUT {*file.variable* | *link.number*}, *time*

file.variable specifies a file opened for sequential access.

time is an expression that evaluates to the number of seconds the program should wait before terminating the READSEQ or READBLK statement or the UVNet connections. If you specify the time value followed by “UM” or “um” UniVerse uses microseconds for the timeout value. For example, “50UM” specifies 50 microseconds.

link.number is the UVNet link. It is a positive number from 1 through 255 (or the number set in the NET_MAXCONNECT VALUE for UVNet connections).

TIMEOUT causes subsequent READSEQ and READBLK statement to terminate and execute their ELSE statements if the number of seconds specified by *time* elapses while waiting for data. Use the STATUS function, on page 380 to determine if *time* has elapsed. In the event of a timeout, neither READBLK nor READSEQ returns any bytes from the buffer, and the entire I/O operation must be retried.

If either *file.variable* or *time* evaluates to the null value, the TIMEOUT statement fails and the program terminates with a run-time error message.

Examples

```
TIMEOUT SUN.MEMBER, 10
 READBLK VAR1 FROM SUN.MEMBER, 15 THEN PRINT VAR1 ELSE
    IF STATUS() = 2 THEN
     PRINT "TIMEOUT OCCURRED"
    END ELSE
     PRINT "CANNOT OPEN FILE"
    END
```

```
    GOTO EXIT.PROG
 END
```

This is the program output:

```
TIMEOUT OCCURRED
```

The following example sets a 30-second timeout for the UVNet connection to the system ORION:

```
TIMEOUT SYSTEM (1200, "ORION"), 30
 OPEN "ORION!/u1/user/file" TO FU.ORIONFILE
 READ X,Y FROM FU.ORIONFILE
    ELSE
     IF SYSTEM (1203)= 81015
     THEN PRINT "TIMEOUT ON READ"
    END
    ELSE
     PRINT "READ ERROR"
    END
 END
```

TPARM function

Use the `TPARM` function to evaluate a parameterized terminfo string.

Syntax

`TPARM` (*`terminfo.string`*, [*`arg1`*], [*`arg2`*], [*`arg3`*], [*`arg4`*], [*`arg5`*], [*`arg6`*], [*`arg7`*], [*`arg8`*])

terminfo.string represents a string of characters to be compiled by the terminfo compiler, *tic*. These terminal descriptions define the sequences of characters to send to the terminal to perform special functions. *terminfo.string* evaluates to one of four types of capability: numeric, Boolean, string, or parameterized string. If *terminfo.string* or any of the eight arguments evaluates to the null value, the `TPARM` function fails and the program terminates with a run-time error message.

Numeric capabilities are limited to a length of five characters that must form a valid number. Only nonnegative numbers 0 through 32,767 are allowed. If a value for a particular capability does not apply, the field should be left blank.

Boolean capabilities are limited to a length of one character. The letter Y (in either uppercase or lowercase) indicates that the specified capability is present. Any value other than Y indicates that the specified capability is not present.

String capabilities are limited to a length of 44 characters. You can enter special characters as follows:

Character	Description
\E or \e	The ESC character (ASCII 27).
\n or \l	The LINEFEED character (ASCII 10).
\r	The RETURN character (ASCII 13).
\t	The TAB character (ASCII 9).
\b	The BACKSPACE character (ASCII 8).
\f	The formfeed character (ASCII 12).
\s	A space (ASCII 32).

Character	Description
^*x*	The representation for a control character (ASCII 0 through 31). The character can be either uppercase or lowercase. A list of some control character representations follows:

Representation	Control character
^A	^a
ASCII 1 (Ctrl-A)	ASCII 1 (Ctrl-A)
^@	ASCII 0
^[	ASCII 27 (Esc)
^\	ASCII 28
^]	ASCII 29
^^	ASCII 30
^_	ASCII 31
^?	ASCII 127 (Del)

Character	Description
nnn	Represents the ASCII character with a value of *nnn* in octal—for example \033 is the Esc character (ASCII 27).
\\	Represents the "\" character.
\,	Represents the "," character.
\^	Represents the "^" character.

Parameterized string capabilities, such as cursor addressing, use special encoding to include values in the appropriate format. The parameter mechanism is a stack with several commands to manipulate it:

Value	Description
%p*n*	Push parameter number *n* onto the stack. Parameters number 1 through 8 are allowed and are represented by *arg1* through *arg8* of the `TPARM` function.
%'*c*'	The ASCII value of character *c* is pushed onto the stack.
%[*nnn*]	Decimal number *nnn* is pushed onto the top of the stack.
%d	Pop the top parameter off the stack, and output it as a decimal number.
%*n*d	Pop the top parameter off the stack, and output it as a decimal number in a field *n* characters wide.
%0*n*d	Like %*n*d, except that 0s are used to fill out the field.
%c	The top of the stack is taken as a single ASCII character and output.
%s	The top of the stack is taken as a string and output.
%+ %- %* %/	The top two elements are popped off the stack and added, subtracted, multiplied, or divided. The result is pushed back on the stack. The fractional portion of a quotient is discarded.
%m	The second element on the stack is taken modulo of the first element, and the result is pushed onto the stack.
%& % | %^	The top two elements are popped off the stack and a bitwise AND, OR, or XOR operation is performed. The result is pushed onto the stack.

Value	Description
%= %< %>	The second element on the stack is tested for being equal to, less then, or greater than the first element. If the comparison is true, a 1 is pushed onto the stack, otherwise a 0 is pushed.
%! %~	The stack is popped, and either the logical or the bitwise NOT of the first element is pushed onto the stack.
%i	One (1) is added to the first two parameters. This is useful for terminals that use a one-based cursor address, rather than a zero-based.
%P*x*	Pop the stack, and put the result into variable *x*, where *x* is a lowercase letter (a - z).
%g*x*	Push the value of variable *x* on the top of the stack.
%? *exp* %t *exp* [%e *exp*] %;	Form an if-then-else expression, with "%?" representing "IF", "%t" representing "THEN", "%e" representing "ELSE", and "%;" terminating the expression. The else expression is optional. Else-If expressions are possible. For example: %? C1 %t B1 %e C2 %t B2 %e C3 %t B3 %e C4 %t B4 %e % C*n* are conditions, and B*n* are bodies.
%%	Output a percent sign (%).

A delay in milliseconds can appear anywhere in a string capability. A delay is specified by $<*nnn*>, where *nnn* is a decimal number indicating the number of milliseconds (one thousandth of a second) of delay desired. A proper number of delay characters will be output, depending on the current baud rate.

TPRINT statement

Use the TPRINT statement to send data to the screen, a line printer, or another print file. TPRINT is similar to the PRINT statement, except that TPRINT lets you specify time delay expressions in the print list.

Syntax

TPRINT [ON *print.channel*] [*print.list*]

The ON clause specifies the logical print channel to use for output. *print.channel* is an expression that evaluates to a number from –1 through 255. If you do not use the ON clause, logical print channel 0 is used, which prints to the user's terminal if PRINTER OFF is set (see the PRINTER statement). If print.channel evaluates to the null value, the TPRINT statement fails and the program terminates with a run-time error message. Logical print channel –1 prints the data on the screen, regardless of whether a PRINTER ON statement has been executed.

You can specify HEADING statement, FOOTING statement, $PAGE statement, and PRINTER CLOSE statements for each logical print channel. The contents of the print files are printed in order by logical print channel number.

print.list can contain any BASIC expression. The elements of the list can be numeric or character strings, variables, constants, or literal strings. The list can consist of a single expression or a series of expressions separated by commas (,) or colons (:) for output formatting. If no *print.list* is designated, a blank line is printed. The null value cannot be printed.

print.list can also contain time delays of the form $<*time*>. *time* is specified in milliseconds to the tenth of a millisecond. As the print list is processed, each time delay is executed as it is encountered.

Expressions separated by commas are printed at preset tab positions. The default tabstop setting is 10 characters. See the TABSTOP statement, on page 399 for information about changing the default setting. Use multiple commas together for multiple tabulations between expressions.

Expressions separated by colons are concatenated. That is, the expression following the colon is printed immediately after the expression preceding the colon. To print a list without a LINEFEED and RETURN, end *print.list* with a colon (:).

If NLS is enabled, the TPRINT statement maps data in the same way as the PRINT statement. For more information about maps, see the *UniVerse NLS Guide*.

Example

The following example prints the string ALPHA followed by a delay of 1 second, then the letters in the variable X. The printing of each letter is followed by a delay of one tenth of a second.

```
X="A$<100>B$<100>C$<100>D$<100>E"
TPRINT "ALPHA$<1000.1> ":X
```

This is the program output:

```
ALPHA ABCDE
```

TRANS function

Use the TRANS function to return the contents of a field or a record in a UniVerse file. TRANS opens the file, reads the record, and extracts the specified data.

Syntax

TRANS ([DICT] *filename, record.ID, field#, control.code*)

filename is an expression that evaluates to the name of the remote file. If TRANS cannot open the file, a run-time error occurs, and TRANS returns an empty string.

record.ID is an expression that evaluates to the ID of the record to be accessed. If *record.ID* is multivalued, the translation occurs for each record ID and the result is multivalued (system delimiters separate data translated from each record).

field# is an expression that evaluates to the number of the field from which the data is to be extracted. If *field#* is -1, the entire record is returned, except for the record ID.

control.code is an expression that evaluates to a code specifying what action to take if data is not found or is the null value. The possible control codes are:

Code	Description
X	(Default) Returns an empty string if the record does not exist or data cannot be found.
V	Returns an empty string and produces an error message if the record does not exist or data cannot be found.
C	Returns the value of *record.ID* if the record does not exist or data cannot be found.
N	Returns the value of *record.ID* if the null value is found.

The returned value is lowered. For example, value marks in the original field become subvalue marks in the returned value. For more information, see the LOWER function, on page 249.

If *filename*, *record.ID*, or *field#* evaluates to the null value, the TRANS function fails and the program terminates with a run-time error message. If *control.code* evaluates to the null value, null is ignored and X is used.

The TRANS function is the same as the XLATE function.

Example

```
X=TRANS("VOC","EX.BASIC",1,"X")
 PRINT "X= ":X
 *
 FIRST=TRANS("SUN.MEMBER","6100",2,"X")


LAST=TRANS("SUN.MEMBER","6100",1,"X")
 PRINT "NAME IS ":FIRST:" ":LAST
```

This is the program output:

```
X= F BASIC examples file
 NAME IS BOB MASTERS
```

transaction statements

Syntax

```
BEGIN TRANSACTION
        [statements]  { COMMIT [WORK] | ROLLBACK [WORK] }
       [statements]  [{ COMMIT [WORK] | ROLLBACK [WORK] }
       [statements]  .  .  .]
       END TRANSACTION
```

Syntax (PIOPEN)

```
TRANSACTION START
        {THEN statements [ELSE statements] | ELSE statements}
       TRANSACTION COMMIT
        {THEN statements [ELSE statements] | ELSE statements}
      TRANSACTION ABORT
```

Use transaction statements to treat a sequence of file I/O operations as one logical operation with respect to recovery and visibility to other users. These operations can include file I/O operations or subtransactions.

Note: BASIC accepts PI/open syntax in addition to UniVerse syntax. You cannot mix both types of syntax within a program.

For more information about transaction statements, refer to *UniVerse BASIC*.

TRANSACTION ABORT statement

Use the TRANSACTION ABORT statement to cancel all file I/O changes made during a transaction.

You can use the TRANSACTION ABORT statement in a transaction without a TRANSACTION COMMIT statement to review the results of a possible change. Doing so does not affect the parent transaction or the database.

After the transaction ends, execution continues with the statement following the TRANSACTION ABORT statement.

Syntax

TRANSACTION ABORT

Example

The following example shows the use of the TRANSACTION ABORT statement to terminate a transaction if both the ACCOUNTS RECEIVABLE file and the INVENTORY file cannot be successfully updated:

```
PROMPT ''
 OPEN 'ACC.RECV'    TO ACC.RECV ELSE STOP 'NO OPEN ACC.RECV'
 OPEN 'INVENTORY'   TO INVENTORY ELSE STOP 'NO OPEN INVENTORY'

 PRINT 'Customer Id : ':
 INPUT CUST.ID
 PRINT 'Item No.    : ':
 INPUT ITEM
 PRINT 'Amount      : ':
 INPUT AMOUNT

 * Start a transaction to ensure both or neither records
 * updated
 TRANSACTION START ELSE STOP 'Transaction start failed.'
 * Read customer record from accounts receivable
    READU ACT.REC FROM ACC.RECV, CUST.ID
    ON ERROR
            STOP 'Error reading ':CUST.ID:' from ACC.RECV file.'
    END LOCKED
        * Could not lock record so ABORT transaction
        TRANSACTION ABORT
            STOP 'Record ':CUST.ID:' on file ACC.RECV locked by user ':STATUS()
    END THEN
        * Build new record
        ACT.REC<1,-1> = ITEM:@SM:AMOUNT
        ACT.REC<2> = ACT.REC<2> + AMOUNT
    END ELSE
        * Create new record
        ACT.REC = ITEM:@SM:AMOUNT:@FM:AMOUNT
    END
    * Read item record from inventory
    READU INV.REC FROM INVENTORY, ITEM
    ON ERROR
            STOP 'Error reading ':ITEM:' from INVENTORY file.'
    END LOCKED
        * Could not lock record so ABORT transaction
        TRANSACTION ABORT
            STOP 'Record ':ITEM:' on file INVENTORY locked by user ':STATUS()
    END THEN
        * Build new record
        INV.REC<1> = INV.REC<1> - 1
        INV.REC<2> = INV.REC<2> - AMOUNT
    END ELSE
            STOP 'Record ':ITEM:' is not on file INVENTORY.'
```

```
      END
      * Write updated records to accounts receivable and inventory
      WRITEU ACT.REC TO ACC.RECV, CUST.ID
      WRITEU INV.REC TO INVENTORY, ITEM

   TRANSACTION COMMIT ELSE STOP 'Transaction commit failed.'

   END
```

TRANSACTION COMMIT statement

Use the TRANSACTION COMMIT statement to commit all file I/O changes made during a transaction.

The TRANSACTION COMMIT statement can either succeed or fail. If the TRANSACTION COMMIT statement succeeds, the THEN statements are executed; any ELSE statements are ignored. If the TRANSACTION COMMIT statement fails, the ELSE statements, if present, are executed, and control is transferred to the statement following the TRANSACTION COMMIT statement.

Syntax

```
TRANSACTION COMMIT
      {THEN statements [ELSE statements] | ELSE statements}
```

TRANSACTION START statement

Use the TRANSACTION START statement to begin a new transaction.

Syntax

```
TRANSACTION START
      {THEN statements [ELSE statements] | ELSE statements}
```

THEN and ELSE clauses

You must have a THEN clause or an ELSE clause, or both, in a TRANSACTION START statement.

If the TRANSACTION START statement successfully begins a transaction, the statements in the THEN clause are executed. If for some reason UniVerse is unable to start the transaction, a fatal error occurs, and you are returned to the UniVerse prompt.

TRIM function

Use the `TRIM` function to remove unwanted characters in *expression*.

Syntax

```
TRIM (expression [,character [,option]] )
```

If only *expression* is specified, multiple occurrences of spaces and tabs are reduced to a single tab or space, and all leading and trailing spaces and tabs are removed. If *expression* evaluates to one or more space characters, `TRIM` returns an empty string.

character specifies a character other than a space or a tab. If only *expression* and *character* are specified, multiple occurrences of *character* are replaced with a single occurrence, and leading and trailing occurrences of *character* are removed.

option specifies the type of trim operation to be performed:

Option	Description
A	Remove all occurrences of *character*
B	Remove both leading and trailing occurrences of *character*
D	Remove leading, trailing, and redundant white space characters
E	Remove trailing white space characters
F	Remove leading white space characters
L	Remove all leading occurrences of *character*
R	Remove leading, trailing, and redundant occurrences of *character*
T	Remove all trailing occurrences of *character*

If *expression* evaluates to the null value, null is returned. If *option* evaluates to the null value, null is ignored and option R is assumed. If *character* evaluates to the null value, the `TRIM` function fails and the program terminates with a run-time error message.

If NLS is enabled, you can use `TRIM` to remove other white space characters such as Unicode values 0x2000 through 0x200B, 0x00A0, and 0x3000, marked as TRIMMABLE in the NLS.LC.CTYPE file entry for the specified locale. For more information about Unicode values, see the *UniVerse NLS Guide*.

Example

```
A="     Now is the time     for     all good men to"
 PRINT A
 PRINT TRIM(A)
```

This is the program output:

```
    Now is the time     for     all good men to
Now is the time for all good men to
```

TRIMB function

Use the `TRIMB` function to remove all trailing spaces and tabs from *expression*. All other spaces or tabs in *expression* are left intact. If *expression* evaluates to the null value, null is returned.

If NLS is enabled, you can use `TRIMB` to remove white space characters such as Unicode values 0x2000 through 0x200B, 0x00A0, and 0x3000, marked as TRIMMABLE in the NLS.LC.CTYPE file entry for the specified locale. For more information about Unicode values, see the *UniVerse NLS Guide*.

Syntax

TRIMB (*expression*)

Example

```
A="     THIS IS A   SAMPLE STRING   "
 PRINT "'":A:"'": " IS THE STRING"
 PRINT "'":TRIMB(A):"'":" IS WHAT TRIMB DOES"
```

```
END
```

This is the program output:

```
'       THIS IS A   SAMPLE STRING   ' IS THE STRING
 '  THIS IS A   SAMPLE STRING' IS WHAT TRIMB DOES
```

TRIMBS function

Use the TRIMBS function to remove all trailing spaces and tabs from each element of *dynamic.array*.

TRIMBS removes all trailing spaces and tabs from each element and reduces multiple occurrences of spaces and tabs to a single space or tab.

If *dynamic.array* evaluates to the null value, null is returned. If any element of *dynamic.array* is null, null is returned for that value.

If you use the subroutine syntax, the resulting dynamic array is returned as *return.array*.

If NLS is enabled, you can use TRIMBS to remove white space characters such as Unicode values 0x2000 through 0x200B, 0x00A0, and 0x3000, marked as TRIMMABLE in the NLS.LC.CTYPE file entry for the specified locale. For more information about Unicode values, see the *UniVerse NLS Guide*.

Syntax

TRIMBS (*dynamic.array*)

CALL -**TRIMBS** (*return.array, dynamic.array*)

TRIMF function

Use the TRIMF function to remove all leading spaces and tabs from *expression*. All other spaces or tabs in *expression* are left intact. If *expression* evaluates to the null value, null is returned.

If NLS is enabled, you can use TRIMF to remove white space characters such as Unicode values 0x2000 through 0x200B, 0x00A0, and 0x3000, marked as TRIMMABLE in the NLS.LC.CTYPE file entry for the specified locale. For more information about Unicode values, see the *UniVerse NLS Guide*.

Syntax

TRIMF (*expression*)

Example

```
A="     THIS IS A   SAMPLE STRING   "
 PRINT "'":A:"'":" IS THE STRING"
 PRINT "'":TRIMF(A):"'":" IS WHAT TRIMF DOES"
 END
```

This is the program output:

```
'       THIS IS A   SAMPLE STRING   ' IS THE STRING
 'THIS IS A     SAMPLE STRING   ' IS WHAT TRIMF DOES
```

TRIMFS function

Use the TRIMFS function to remove all leading spaces and tabs from each element of *dynamic.array*.

TRIMFS removes all leading spaces and tabs from each element and reduces multiple occurrences of spaces and tabs to a single space or tab.

If *dynamic.array* evaluates to the null value, null is returned. If any element of *dynamic.array* is null, null is returned for that value.

If you use the subroutine syntax, the resulting dynamic array is returned as *return.array*.

If NLS is enabled, you can use TRIMFS to remove white space characters such as Unicode values 0x2000 through 0x200B, 0x00A0, and 0x3000, marked as TRIMMABLE in the NLS.LC.CTYPE file entry for the specified locale. For more information about Unicode values, see the *UniVerse NLS Guide*.

Syntax

```
TRIMFS (dynamic.array)
CALL -TRIMFS (return.array, dynamic.array)
```

TRIMS function

Use the TRIMS function to remove unwanted spaces and tabs from each element of *dynamic.array*.

TRIMS removes all leading and trailing spaces and tabs from each element and reduces multiple occurrences of spaces and tabs to a single space or tab.

If *dynamic.array* evaluates to the null value, null is returned. If any element of *dynamic.array* is null, null is returned for that value.

If you use the subroutine syntax, the resulting dynamic array is returned as *return.array*.

If NLS is enabled, you can use TRIMS to remove white space characters such as Unicode values 0x2000 through 0x200B, 0x00A0, and 0x3000, marked as TRIMMABLE in the NLS.LC.CTYPE file entry for the specified locale. For more information about Unicode values, see the *UniVerse NLS Guide*.

Syntax

```
TRIMS (dynamic.array)
CALL -TRIMS (return.array, dynamic.array)
```

TTYCTL statement

Use the TTYCTL statement to set terminal device characteristics on Berkeley terminal drivers. *code#* specifies the action to take.

This statement is not supported on UNIX System V or Windows NT.

Syntax

```
TTYCTL file.variable, code#
       {THEN statements [ELSE statements] | ELSE statements}
```

The following table lists the available actions:

Argument	Action
0	No operation, determines if a device is a TTY.
1	Sets HUP (hang up data line) on close of file.
2	Clears HUP on close of file.
3	Sets exclusive use flag for TTY.
4	Resets exclusive use flag.
5	Sets the BREAK.
6	Clears the BREAK.
7	Turns on DTR (Data Terminal Ready).
8	Turns off DTR.
9	Flushes input and output buffers.
10	Waits for the output buffer to drain.

file.variable specifies a file previously opened for sequential access to a terminal device. If *file.variable* evaluates to the null value, the TTYCTL statement fails and the program terminates with a run-time error message.

If the action is taken, the THEN statements are executed. If no THEN statements are present, program execution continues with the next statement.

If an error is encountered during the execution of the TTYCTL operation, or if the file variable is not open to a terminal device, the ELSE statements are executed; any THEN statements are ignored.

Example

```
OPENSEQ 'FILE.E', 'RECORD4' TO FILE ELSE ABORT
 *
 TTYCTL FILE, 0
            THEN PRINT 'THE FILE IS A TTY'
            ELSE PRINT 'THE FILE IS NOT A TTY'
```

This is the program output:

```
THE FILE IS NOT A TTY
```

TTYGET statement

Use the TTYGET statement to assign the characteristics of a terminal, line printer channel, or tape unit as a dynamic array to *variable*. If the FROM clause is omitted, a dynamic array of the terminal characteristics for your terminal is assigned to *variable*.

Syntax

```
TTYGET variable [FROM {file.variable | LPTR [n] | MTU [n] }]
{THEN statements [ELSE statements] | ELSE statements}
```

file.variable is a terminal opened for sequential processing with the OPENDEV statement or OPENSEQ statement. If *file.variable* is specified, the terminal characteristics for the specified terminal are retrieved.

n specifies a logical print channel with LPTR or a tape unit with MTU. (You cannot specify a tape unit on Windows NT.) If *n* is specified, the characteristics for the print channel or tape unit are retrieved. For

logical print channels *n* is in the range of 0 through 225; the default is 0. For tape units *n* is in the range of 0 through 7; the default is 0.

If the terminal characteristics are retrieved, the THEN statements are executed.

If the device does not exist or cannot be opened, or if no dynamic array is returned, the ELSE statements are executed; any THEN statements are ignored.

If either *file.variable* or *n*evaluates to the null value, the TTYGET statement fails and the program terminates with a run-time error message.

The best way to access the information in the dynamic array is to include the BASIC code UNIVERSE.INCLUDE TTY. The syntax for including this file is:

```
$INCLUDE UNIVERSE.INCLUDE TTY
```

This file equates each value of the dynamic array to a name, so that each value can be easily accessed in your program. To take advantage of this code you must call variable *tty$*. Once this code has been included in your program, you can use the names to access the values of the dynamic array. To set values for a terminal line, use the TTYSET statement.

The following table lists the equate names to the values of the dynamic array, and describes each value. The final columns indicate which values are available on different operating systems: SV indicates System V, B indicates Berkeley UNIX, and NT indicates Windows NT.

Value	Name	Description	Availability		
			SV	B	NT
Field 1					
1	mode.type	One of these modes: MODE$LINE or 0 = line MODE$RAW or 1 = raw MODE$CHAR or 2 = character MODE$EMULATE or 3 = emulated	3 3 3 3	3 3 3 3	3 3
2	mode.min	Minimum number of characters before input.	3	3	3
3	mode.time	Minimum time in milliseconds before input.	3	3	3
Field 2					
1	cc.intr	Interrupt character. -1 undefined.	3	3	3
2	cc.quit	Quit character. -1 undefined.	3	3	
3	cc.susp	Suspend character. -1 undefined.	3	3	
4	cc.dsusp	dsusp character. -1 undefined.		3	
5	cc.switch	Switch character. -1 undefined.	3		
6	cc.erase	erase character. -1 undefined.	3	3	3
7	cc.werase	werase character. -1 undefined.		3	
8	cc.kill	Kill character. -1 undefined.	3	3	3
9	cc.lnext	lnext character. -1 undefined.		3	
10	cc.rprint	rprint character. -1 undefined.		3	3
11	cc.eof	eof character. -1 undefined.	3	3	
12	cc.eol	eol character. -1 undefined.	3	3	
13	cc.eol2	eol2 character. -1 undefined.	3		
14	cc.flush	Flush character. -1 undefined.		3	

Value	Name	Description	Availability		
15	cc.start	Start character. -1 undefined. On System V, ^Q only.	3	3	3
16	cc.stop	Stop character. -1 undefined. On System V, ^S only.	3	3	3
17	cc.lcont	lcont character. -1 undefined. Emulated only.	3	3	3
18	cc.fmc	*fmc* character. -1 undefined. Emulated only.	3	3	3
19	cc.vmc	*vmc* character. -1 undefined. Emulated only.	3	3	3
20	cc.smc	*smc* character. -1 undefined. Emulated only.	3	3	3
21	ccdel	Delete character.	3	3	
Field 3					
1	carrier.receive	Terminal can receive data.	3	3	3
2	carrier.hangup	Hang up upon close of terminal.	3	3	
3	carrier.local	Terminal is a local line.	3	3	3
Field 4					
1	case.ucin	Convert lowercase to uppercase on input.	3	3	
2	case.ucout	Convert lowercase to uppercase on output.	3	3	
3	case.xcase	Uppercase is preceded by a backslash (\ ) to distinguish it from lowercase.	3	3	
4	case.invert	Invert case on input. Emulated only.	3	3	3
Field 5					
1	crmode.inlcr	Convert LINEFEED to RETURN on input.	3	3	
2	crmode.igncr	Ignore RETURN on input.	3	3	
3	crmode.icrnl	Convert RETURN to LINEFEED on input.	3	3	
4	crmode.onlcr	Convert LINEFEED to LINEFEED, RETURN on output.	3	3	
5	crmode.ocrnl	Convert RETURN to LINEFEED on output.	3	3	
6	crmode.onocr	Prohibit output of RETURN when cursor is in column 0.	3	3	
7	crmode.onlret	LINEFEED performs RETURN function.	3	3	
Field 6					
1	delay.bs	Set backspace delay.	3	3	
2	delay.cr	Set RETURN delay.	3	3	
3	delay.ff	Set formfeed delay.	3	3	
4	delay.lf	Set LINEFEED delay.	3	3	

Value	Name	Description	Availability		
5	delay.vt	Set vertical tab delay.	3	3	
6	delay.tab	Set tab delay.	3	3	
7	delay.fill	0 = time delay 1 = fill with empty strings 2 = fill with DELETEs	3	3	
Field 7					
1	echo.on	Set terminal echo on.	3	3	3
2	echo.erase	ECHOE$ERASE or 0 = print echo character ECHOE$BS or 1 = echo as backspace ECHOE$BSB or 2 = echo as backspace, space, backspace ECHOE$PRINTER or 3 = echo as a printer	3	3	
3	echo.kill	ECHOK$KILL or 0 = kill as kill character ECHOK$LF or 1 = kill as RETURN, LINEFEED ECHOK$ERASE or 2 = kill as series of erases	3	3	
4	echo.ctrl	Set control to echo as ^ character	3	3	
5	echo.lf	When echo is off, echo RETURN as RETURN, LINEFEED	3	3	3
Field 8					
1	handshake.xon	1 = turns on X-ON/X-OFF protocol 0 = turns off X-ON/X-OFF protocol	3	3	3
2	handshake. startany	1 = any characters acts as X-ON 0 = only X-ON character acts as X-ON	3	3	
3	handshake. tandem	1 = when input buffer is nearly full, X-OFF is sent 0 = turns off automatic X-OFF, X-ON mode	3	3	3
4	handshake.dtr	1 = turns on DTR 0 = turns off DTR	3	3	
Field 9					
1	output.post	Output postprocessing occurs.	3	3	
2	output.tilde	Special output processing for tilde.	3	3	
3	output.bg	Stop background processes at output.	3	3	
4	output.cs	Output clearscreen before reports. Emulated only.	3	3	
5	output.tab	Set output tab expansion.	3	3	
Field 10					

Value	Name	Description	Availability		
1	protocol.line	Line protocol	3	3	
2	protocol.baud	1 = 50 9 = 1200 2 = 75 10 = 1800 3 = 110 11 = 2400 4 = 134 12 = 4800 5 = 150 13 = 9600 6 = 200 14 or EXTA = 19200 7 = 300 15 = EXTB 8 = 600	3	3	3
3	protocol.data	Character size: 5 = 5 bits 7 = 7 bits 6 = 6 bits 8 = 8 bits	3	3	3
4	protocol.stop	2 = 2 stopbits 1 = 1 stopbit	3	3	3
5	protocol.output	Output parity: 0 = no parity 1 = even parity 2 = odd parity	3 3	3 3	3 3 3
6	protocol.input	Input parity: 0 = disable input parity checking 1 = enable input parity checking 2 = mark parity errors 3 = mark parity errors with a null 4 = ignore parity errors	3 3 3 3 3	3 3 3 3 3	3 3 3 3
7	protocol.strip	1 = strip to 7 bits 0 = 8 bits	3	3	
Field 11					
1	signals.enable	Enable signal keys: Interrupt, Suspend, Quit.	3	3	
2	signals.flush	Flush type-ahead buffer.	3	3	
3	signals.brkkey	0 = break ignored 1 = break as interrupt 2 = break as null	3	3	

TTYSET statement

Use the TTYSET statement to set the characteristics of a terminal, line printer channel, or tape unit. If only *dynamic.array* is specified, the terminal characteristics for your terminal are set based on the contents of *dynamic.array*. *dynamic.array* is a dynamic array of eleven fields, each of which has multiple values.

A description of the expected contents of each value of *dynamic.array* is given in the TTYGET statement, on page 427.

Syntax

```
TTYSET dynamic.array [ON {file.variable | LPTR [n] | MTU [n] }]
       {THEN statements [ELSE statements] | ELSE statements}
```

file.variable is a terminal opened for sequential processing with the OPENDEV statement or OPENSEQ statement. If *file.variable* is specified, the terminal characteristics for the specified terminal are set.

n specifies a logical print channel with LPTR or a tape unit with MTU. If *n* is specified, the characteristics for the print channel or tape unit are set. *n* is in the range of 0 through 225 for logical print channels; the default is 0. *n* is in the range of 0 through 7 for tape units; the default is 0. On Windows NT you cannot specify a tape unit.

If the terminal characteristics are set, the THEN statements are executed.

If the device does not exist or cannot be opened, or if no dynamic array is returned, the ELSE statements are executed; any THEN statements are ignored.

If *dynamic.array*, *file.variable*, or *n* evaluates to the null value, the TTYSET statement fails and the program terminates with a run-time error message.

To build *dynamic.array*, get the current values of the terminal line using the TTYGET statement, manipulate the values, and reset them with the TTYSET statement. The best way to access the information in the dynamic array is to include the BASIC code UNIVERSE.INCLUDE TTY. The syntax for including this file is:

```
$INCLUDE UNIVERSE.INCLUDE TTY
```

This file equates each value of *variable* from the TTYGET statement with a name, so that each value can be easily accessed in your program. To take advantage of this code you must call variable *tty$*. Once this code is included in your program, you can use the names to access the values of the dynamic array. The TTYGET Statement Values table lists the names equated to the values of the dynamic array and describes the values.

Timeout handling

You can set the MODE.MIN and MODE.TIME values to define timeouts for read operations over a communications line. MODE.MIN specifies the minimum number of characters to be received. MODE.TIME specifies time in tenths of a second. The two values interact to provide four cases that can be used as follows.

Intercharacter timer

When you set the values of both MODE.MIN and MODE.TIME to greater than 0, MODE.TIME specifies the maximum time interval allowed between successive characters received by the communication line in tenths of a second. Timing begins only after the first character is received.

Blocking read

When you set the value of MODE.MIN to greater than 0 and MODE.TIME to 0, no time limit is set, so the read operation waits until the specified number of characters have been received (or a newline in the case of READSEQ statement).

Read timer

When you set the value of MODE.MIN to 0 and MODE.TIME to greater than 0, MODE.TIME specifies how long the read operation waits for a character to arrive before timing out. If no characters are received in the time specified, the READSEQ and READBLK statement use the ELSE clause if there is one. If you use the NOBUF statement to turn off buffering, the timer is reset after each character is received.

Nonblocking read

When you set the values of both MODE.MIN and MODE.TIME to 0, data is read as it becomes available. The read operation returns immediately.

- If any characters are received:
  - READBLK returns as many characters as specified in the blocksize argument, or all the characters received, whichever is fewer.
  - READSEQ returns characters up to the first newline, or all the characters received if no newline is received.
- If no characters are received, READSEQ and READBLK use the ELSE clause if there is one.

UDOArrayAppendItem

The `UDOArrayAppendItem`() function appends the item you specify to the UDO array.

Syntax

`UDOArrayAppendItem``(udoHandle, value)`

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
udoHandle	Must be a UDO array.
value	The value of the array item you are appending.

If the new array item is of UDO_OBJECT or UDO_ARRAY type, it must be a stand-alone object or array, and it must not be the ancestor of the current UDO object.

UDOArrayDeleteItem

The `UDOArrayDeleteItem`() function deletes the array item you specify by its index.

Syntax

`UDOArrayDeleteItem``(udoHandle,index)`

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
udoHandle	Must be a UDO array.
index	The index of the item to be deleted. Must be a positive integer.

If the array item is of UDO_ARRAY or UDO_OBJECT type, UDO will make either the UDO object or a UDO array as stand-alone and will remove it from memory if it is not referenced by any UniVerse BASIC variable.

UDOArrayGetItem

The UDOArrayGetItem() function returns a UDO array item by its index.

Syntax

UDOArrayGetItem(*udoHandle*, *index*, *value*[out], *value_type*[out])

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
udoHandle	Must be a UDO array.
index	The position of the UDO array index returned. Must be a positive integer.
value[out]	The UDO value type of the array item. If the array item is of UDO_OBJECT or UDO_ARRAY type, the output variable “item” holds only a reference to the object or array. Further changes to the object or array through this reference, such as updating a property value or removing an array item, affect the original item as well. If the array item is of UDO_STRING, UDO_NUMBER, UDO_TRUE, UDO_FALSE or UDO_NULL type, the output variable “item” holds the actual value instead of a reference. Further changes to this variable do not affect the original property value.
value_type[out]	The type of the value returned by *value*.

UDOArrayGetNextItem

The UDOArrayGetNextItem() function returns the next UDO array item relative to the current position, which is the position of the array the last time it was accessed by this function. The initial position is 1.

Syntax

UDOArrayGetNextItem(*udoHandle*, *value*[out], *type*[out])

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
udoHandle	Must be a UDO array.
value[out]	The value of the item.
type[out]	The type of the value returned by *value*.

After exhausting the entire array, the UDOArrayGetNextItem() function returns UDO_ERROR and the current position is reset to 1.

We recommend that you not modify the array when calling the UDOArrayGetNextItem() function. If you must modify the array, remember that UDOArrayGetNextItem() always returns the item at the current position +1.

UDOArrayGetSize

The `UDOArrayGetSize`() function gets the size of a UDO array.

Syntax

`UDOArrayGetSize``(`*`udoHandle, size`*`[out])`

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
udoHandle	Must be a UDO array.
size	The size of the UDO array.

UDOArrayInsertItem

The `UDOArrayInsertItem`() function inserts a UDO array element at the position you specify by index.

Syntax

`UDOArrayInsertItem``(`*`udoHandle, index, value`*`)`

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
udoHandle	Must be a UDO array.
index	The position what you want to insert the item. Must be a positive integer.
value	The value of the array item you are inserting.

If the index is larger than the size of the array, UDO will pad the array with UDO_NULL values before it inserts the array item into the array.

UDOArraySetItem

The `UDOArraySetItem`() function sets or inserts a UDO array element at the position you specify.

Syntax

`UDOArraySetItem``(`*`udoHandle, index, value`*`)`

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
udoHandle	Must be a UDO array.

Parameter	Description
index	The position what you want to set or insert the element. Must be a positive integer.
value	The value of the array item you are setting.

If the index is larger than the size of the array, UDO will pad the array with UDO_NULL values before it inserts the array item into the array.

Otherwise, if the old array item is of UDO_OBJECT or UDO_ARRAY type, either an object or an array will be marked as stand-alone and removed from memory if it is not referenced by any UniVerse BASIC variable.

If the new array item is of UDO_OBJECT or UDO_ARRAY type, it must be a stand-alone object or array and it must not be the ancestor of the current UDO object.

UDOClone

The `UDOClone` function clones a UDO object or array so that changes to the new object or array will not affect the original object.

Syntax

`UDOClone``(`*`udoHandle, newUdoHandle`*`[out])`

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
udoHandle	Must be a UDO array.
newUdoHandle	When the `UDOClone` function returns successfully, newUDOHandle points to a stand-alone object or array that is the exact replication of the original object.

UDOCreate

The `UDOCreate` function creates a UDO item of the type you specify.

Syntax

`UDOCreate``(`*`udoType, udoHandle`*`[out])`

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
udoType	Must be one of UDO_OBJECT, UDO_ARRAY, UDO_TRUE, UDO_FALSE, or UDO_NULL.
udoHandle	If *udoType* is UDO_OBJECT, *udoHandle* holds an empty object. If *udoType* is UDO_ARRAY, *udoHandle* holds an empty array. If *udoType* is UDO_TRUE, UDO_FALSE, or UDO_NULL, *udoHandle*.

UDODeleteProperty

The `UDODeleteProperty` function deletes a property from the UDO object.

Syntax

`UDODeleteProperty``(udoHandle, name)`

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
udoHandle	Must be a UDO object.
name	The name of the property. If the property is of UDO_OBJECT or UDO_ARRAY type, its value (either a UDO object or a UDO array) is marked as stand-alone and will be removed from memory if it is not referenced by any UniVerse BASIC variable.

UDOFree

The `UDOFree` function forcefully removes a UDO object or array from memory.

Syntax

`UDOFree``(udoHandle)`

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
udoHandle	Must be a stand-alone UDO object or array.

UDO will clear all UniVerse BASIC variables that reference the object or array and its descendants. Any attempt to access these variables, other than assigning a new value, fails.

You should always call this function when a UDO object or array is no longer needed. This avoids a potential memory leak.

UDOGetLastError

If the previous UDO call returned UDO_ERROR, use the `UDOGetLastError`() function to return the error code and error message.

Syntax

`UDOGetLastError``(errorCode[out], errorMessage[out])`

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
errorCode	The UDO error code.
errorMessage	The UDO error message.

UDOGetNextProperty

The `UDOGetNextProperty` function provides a convenient way to walk through all the properties in a UDO object, without needing to know the property names in advance.

When all properties on the UDO object are exhausted, the `UDOGetNextProperty`() function returns UDO_ERROR, then goes back to the first property.

We recommend that you avoid modifying the properties on a UDO object when calling the `UDOGetNextProperty`() to retrieve the properties.

Syntax

`UDOGetNextProperty``(`*`udoHandle`*`, `*`name`*`[out], `*`value`*`[out], `*`value_type`*`[out])`

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
udohandle	Must be a UDO type object.
name[out]	The name of the array that holds the names of all the properties in the UDO object.
value[out]	If the property is a UDO_OBJECT or UDO_ARRAY type (it is either a UDO object or an array), the output value holds only a reference to the object or array. Further changes to the object or array through this reference, such as updating a property value on the object or removing an array item, affects the original object as well. If the property is a UDO_STRING, UDO_NUMBER, UDO_TRUE, UDO_FALSE, or UDO_NULL type, the output variable value holds the actual value instead of a reference. Further changes to this variable do not affect the original property value.
value_type[out]	The type of the value returned by *value*.

UDOGetOption

The `UDOGetOption` function gets the value of a UDO option.

Syntax

`UDOGetOption``(`*`option`*`, `*`value`*`[out])`

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
option	The UDOOPTION you want to use.

Parameter	Description
value[out]	A string type option value.

UDOGetProperty

The `UDOGetProperty` function returns the value and type of property on the UDO object.

Syntax

`UDOGetProperty``(udoHandle, name, value[out], value_type[out])`

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
udoHandle	Must be a UDO object.
name	The name of the property.
value[out]	If the property is a UDO_OBJECT or UDO_ARRAY type (it is either a UDO object or an array), the output value holds only a reference to the object or array. Further changes to the object or array through this reference, such as updating a property value on the object or removing an array item, affects the original object as well. If the property is a UDO_STRING, UDO_NUMBER, UDO_TRUE, UDO_FALSE, or UDO_NULL type, the output variable value holds the actual value instead of a reference. Further changes to this variable do not affect the original property value.
value_type[out]	The type of the value returned by *value*.

UDOGetPropertyNames

The `UDOGetPropertyNames` function returns a UDO array that holds the names of all the properties in the UDO object.

Syntax

`UDOGetPropertyNames``(udoHandle, udoArray[out])`

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
udoHandle	Must be a UDO object.
udoArray[out]	The UDO array to hold the names of all the properties in the UDO object.

UDOGetType

The `UDOGetType`() function gets the UDO value type of a UniVerse BASIC variable.

Syntax

UDOGetType(*udoHandle*, *type*[out)

Parameters

The following table describes each parameter of the syntax.

Parameters	Description
udoHandle	Can be a UDO handle, or a UniVerse BASIC string or number.
type[out]	The UDO value type.

UDOIsTypeOf

The UDOIsTypeOf() function tests the UDO value type of a UniVerse BASIC variable.

Syntax

UDOIsTypeOf(*udoHandle*, *type*)

Parameters

The following table describes each parameter of the syntax.

Parameters	Description
udoHandle	Can be a UDO handle, or a UniVerse BASIC string or number.
type[in]	The UDO value type.

UDORead

The UDORead function creates a UDO object from a JSON string or XMLstring.

Syntax

UDORead(*inputString*, *inputType*, *udoHandle*[out])

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
inputString	A JSON or XML string.
inputype	UDOFORMAT_JSON or UDOFORMAT_XML.
udoHandle [out]	The UniVerse BASIC variable that holds a reference to the UDO object upon successful return of the function.

UDOSetOption

Sets the options for the UDO API.

Syntax

UDOSetOption(*option, value*)

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
option	The UDOOPTION you want to use.
value	A string type option value.

UDOSetProperty

The `UDOSetProperty` function creates or updates a property on a UDO object.

Syntax

UDOSetProperty(*udoHandle, name, value*)

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
udoHandle	Must be a UDO object.
name	The name of the property. If the property does not exist, UDO creates a new property for the object. If the property exists, the new value replaces the old value. If the old property is of UDO_OBJECT or UDO_ARRAY type, the old value, either a UDO object or an array, is marked as stand-alone and will be removed from memory if it is not referenced by any UniVerse BASIC variable. If the new value is of UDO_OBJECT or UDO_ARRAY type, it must be a stand-alone object or array, and it must not be the ancestor of the current UDO object.
value	The value of the property.

UDOWrite

Writes a UDO object in JSON or XML format.

Syntax

UDOWrite(*udoHandle, outputType, outputString*[out])

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
udoHandle	Must be a UDO type variable.

Parameter	Description
outputType	UDOFORMAT_JSON or UDOFORMAT_XML.
outputString [out]	The string that holds the serialized output.

UNASSIGNED function

Use the UNASSIGNED function to determine if *variable* is unassigned. UNASSIGNED returns 1 (true) if *variable* is unassigned. It returns 0 (false) if *variable* is assigned a value, including the null value.

Syntax

UNASSIGNED (*variable*)

Example

```
A = "15 STATE STREET"
 C = 23
 X = UNASSIGNED(A)
 Y = UNASSIGNED(B)
 Z = UNASSIGNED(C)
 PRINT X,Y,Z
```

This is the program output:

```
0 1 0
```

UNICHAR function

Use the UNICHAR function to generate a single character from a Unicode value.

Syntax

UNICHAR (*unicode*)

unicode is a decimal number from 0 through 65535 that is the value of the character you want to generate. If *unicode* is invalid, an empty string is returned. If *unicode* evaluates to the null value, null is returned.

The UNICHAR function operates the same way whether NLS mode is enabled or not.

Note: Use BASIC @variables to generate UniVerse system delimiters. Do not use the UNICHAR function.

UNICHARS function

Use the UNICHARS function to generate a dynamic array of characters from a dynamic array of Unicode values.

Syntax

UNICHARS (*dynamic.array*)

dynamic.array is an array of decimal Unicode values separated by system delimiters. If any element of *dynamic.array* is invalid, an empty string is returned for that element. If *dynamic.array* evaluates to the null value, null is returned. If any element of *dynamic.array* is null, null is returned for that element.

The UNICHARS function operates the same way whether NLS mode is enabled or not.

Note: Use BASIC @variables to generate UniVerse system delimiters. Do not use the UNICHARS function.

UNISEQ function

Use the UNISEQ function to generate a Unicode value from *expression*.

Syntax

UNISEQ (*expression*)

The first character of *expression* is converted to its Unicode value, that is, a hexadecimal value in the range 0x0000 through 0x1FFFF. If *expression* is invalid, for example, an incomplete internal string, an empty string is returned. If *expression* evaluates to the null value, null is returned.

The UNISEQ function operates the same way whether NLS mode is enabled or not.

Warning: UNISEQ does not map system delimiters. For example, UNISEQ("û") returns 251 (0x00FB), and UNISEQ(@TM) returns 63739 (0xF8FB). The Unicode value returned is the internal representation of the text mark character that is mapped to a unique area so that it is not confused with any other character. Note that this behaves differently from SEQ(@TM), which returns 251.

For more information about Unicode values and tokens defined for system delimiters, see the *UniVerse NLS Guide*.

UNISEQS function

Use the UNISEQS function to generate an array of Unicode values from a dynamic array of characters.

Syntax

UNISEQS (*dynamic.array*)

dynamic.array specifies an array of characters with the elements separated by system delimiters. The first character of each element of *dynamic.array* is converted to its Unicode value, a hexadecimal value in the range 0x0000 through 0x1FFFF. If any element of *dynamic.array* is invalid, an empty string is returned for that element. If *dynamic.array* evaluates to the null value, null is returned. If any element of *dynamic.array* is the null value, null is returned for that element.

The UNISEQS function operates the same way whether NLS mode is enabled or not.

Warning: UNISEQS does not map system delimiters. For example, UNISEQS("û") returns 251 (0x00FB), and UNISEQS(@TM) returns 63739 (0xF8FB). The Unicode value returned is the internal representation of the text mark character that is mapped to a unique area so that it is not confused with any other character. Note that this behaves differently from SEQ(@TM), which returns 251.

For more information about Unicode values and tokens defined for system delimiters, see the *UniVerse NLS Guide*.

UNLOCK statement

Use the UNLOCK statement to release a process lock set by the LOCK statement.

Syntax

UNLOCK [*expression*]

expression specifies an integer from 0 through 63. If *expression* is not specified, all locks are released (see the LOCK statement).

If *expression* evaluates to an integer outside the range of 0 through 63, an error message appears and no action is taken.

If *expression* evaluates to the null value, the UNLOCK statement fails and the program terminates with a run-time error message.

Examples

The following example unlocks execution lock 60:

```
UNLOCK 60
```

The next example unlocks all locks set during the current login session:

```
UNLOCK
```

The next example unlocks lock 50:

```
X=10
 UNLOCK 60-X
```

UPCASE function

Use the UPCASE function to change all lowercase letters in *expression* to uppercase. If *expression* evaluates to the null value, null is returned.

UPCASE is equivalent to OCONV ("MCU").

If NLS is enabled, the UPCASE function uses the conventions specified by the Ctype category for the NLS.LC.CTYPE file to determine what constitutes uppercase and lowercase. For more information about the NLS.LC.CTYPE file, see the *UniVerse NLS Guide*.

Syntax

UPCASE (*expression*)

Example

```
A="This is an example of the UPCASE function: "
 PRINT A
 PRINT UPCASE(A)
```

This is the program output:

```
This is an example of the UPCASE function:
 THIS IS AN EXAMPLE OF THE UPCASE FUNCTION:
```

UPRINT statement

In NLS mode, use the UPRINT statement to print data that was mapped to an external format using OCONV *mapname*. The UPRINT statement subsequently sends the mapped data to the screen, a line printer, or another print file with no further mapping.

Syntax

UPRINT [ON *print.channel*] [*print.list*]

The ON clause specifies the logical print channel to use for output. *print.channel* is an expression that evaluates to a number from –1 through 255. If you do not use the ON clause, logical print channel 0 is used, which prints to the user's terminal if PRINTER OFF is set (see the PRINTER statement). If *print.channel* evaluates to the null value, the PRINT statement fails and the program terminates with a run-time error message. Logical print channel –1 prints the data on the screen, regardless of whether a PRINTER ON statement has been executed.

You can specify HEADING statement, FOOTING statement, $PAGE statement, and PRINTER CLOSE statements for each logical print channel. The contents of the print files are printed in order by logical print channel number.

print.list can contain any BASIC expression. The elements of the list can be numeric or character strings, variables, constants, or literal strings; the null value, however, cannot be printed. The list can consist of a single expression or a series of expressions separated by commas (,) or colons (:) for output formatting. If no *print.list* is designated, a blank line is printed.

Expressions separated by commas are printed at preset tab positions. The default tabstop setting is 10 characters. For information about changing the default setting, see the TABSTOP statement, on page 399. Use multiple commas together for multiple tabulations between expressions.

Expressions separated by colons are concatenated. That is, the expression following the colon is printed immediately after the expression preceding the colon. To print a list without a LINEFEED and RETURN, end *print.list* with a colon (:).

If NLS is disabled, the UPRINT statement behaves like the PRINT statement.

For more information about maps, see the *UniVerse NLS Guide*.

USERINFO function

Use the USERINFO function to get the pid, user number, and more for the pid or user number specified.

Utilize the USERINFO.H include file to reference the return values, described in USERINFO.H, on page 446.

Syntax

USERINFO (*code*, *value*, *userinfo*)

Parameters

The following table describes each parameter of the syntax.

Parameter	Description	
code	1 is used when the *value* is a pid. 2 is used when the *value* is a @USERNO. [IN]	
value	If *code* is 1, *value* is a pid. If *code* is 2, the *value* is a @USERNO. [IN]	
userinfo	A dynamic array with the UniVerse session's user information stored in attribute 1. The subvalue fields returned are:	
	1	@USERNO for user
	2	Login ID
	3	Pid
	4	userType (phantom or terminal)
	5	User ID (not the same as @USERNO)
	6	TTY/Telnet
	7	The IP address to be returned. If the process is started from a device-licensing-aware client even though device licensing is not enabled in the license configuration, the IP address can be returned. With telnet sessions on UNIX and Linux platforms, the `uvdls` process needs to be called. If the IP address cannot be determined then "N/A" is returned.
	8	Working directory
	9	Logon time
	10	Internal Pick date in local time
	11	Number of seconds since midnight (local time)
	12	Internal Pick date in UTC time
	13	Number of seconds since midnight (UTC time)

Return codes

The following table describes the status of each return code.

Return code	Status
0	No results or invalid user number or pid
1	Success
-1	Invalid code value

USERINFO.H

You can use the `USERINFO` function to extract information from the `USERINFO.H` include file about a particular session's details.

The following tokens can be used when calling the `USERINFO` function.

Value	Token	Description
1	UI$USER_NO	@USERNO for user
2	UI$USER_NAME	Login ID
3	UI$PID	Pid
4	UI$USER_TYPE	userType (phantom or terminal)
5	UI$USER_ID	User ID (not the same as @USERNO)

Value	Token	Description
6	UI$TTY	TTY/Telnet
7	UI$IP_ADDR	The IP address to be returned. If the process is started from a device-licensing-aware client even though device licensing is not enabled in the license configuration, the IP address can be returned. With telnet sessions on UNIX and Linux platforms, the `uvdls` process needs to be called. If the IP address cannot be determined then "N/A" is returned.
8	UI$WORK_DIR	Working directory
9	UI$LOGON_TIME	Logon time
10	UI$LOCAL_DATE	Internal Pick date in local time
11	UI$LOCAL_TIME	Number of seconds since midnight (local time)
12	UI$UTC_DATE	Internal Pick date in UTC time
13	UI$UTC_TIME	Number of seconds since midnight (UTC time)

The following example examines the current `USERINFO` settings:

```
$INCLUDE UNIVERSE.INCLUDE USERINFO.H
CRT "SYS(51): ":SYSTEM(51)
RETCODE = USERINFO(2,@USERNO,RETDATA)
CRT "USER_NO: ":RETDATA<1,UI$USER_NO>
CRT "USER_NAME: ":RETDATA<1,UI$USER_NAME>
CRT "PID: ":RETDATA<1,UI$PID>
CRT "USER_TYPE: ":RETDATA<1,UI$USER_TYPE>
CRT "USER_ID: ":RETDATA<1,UI$USER_ID>
CRT "TTY: ":RETDATA<1,UI$TTY>
CRT "IP_ADDR: ":RETDATA<1,UI$IP_ADDR>
CRT "WORK_DIR: ":RETDATA<1,UI$WORK_DIR>
CRT "LOGON_TIME: ":RETDATA<1,UI$LOGON_TIME>
CRT "LOCAL_DATE: ":RETDATA<1,UI$LOCAL_DATE>
```

WEOF statement

Use the WEOF statement to write an end-of-file (EOF) mark to tape.

Syntax

WEOF [UNIT (*mtu*)] {THEN *statements* [ELSE *statements*] | ELSE *statements*}

The UNIT clause specifies the number of the tape drive unit. Tape unit 0 is used if no unit is specified.

mtu is an expression that evaluates to a three-digit code (decimal). Although the *mtu* expression is a function of the UNIT clause, the WEOF statement uses only the third digit (the *u*). Its value must be in the range of 0 through 7 (see the READT statement, on page 312 for details on the *mtu* expression). If *mtu* evaluates to the null value, the WEOF statement fails and the program terminates with a run-time error message.

Before a WEOF statement is executed, a tape drive unit must be attached (assigned) to the user. Use the `ASSIGN` command to assign a tape unit to a user. If no tape unit is attached or if the unit specification is incorrect, the ELSE statements are executed.

The STATUS function returns 1 if WEOF takes the ELSE clause, otherwise it returns 0.

Example

```
WEOF UNIT(007) ELSE PRINT "OPERATION NOT COMPLETED."
```

WEOFSEQ statement

Use the WEOFSEQ statement to write an end-of-file (EOF) mark in a file opened for sequential access. The end-of-file mark is written at the current position and has the effect of truncating the file at this point. Any subsequent READSEQ statement has its ELSE statements executed.

Syntax

WEOFSEQ*file.variable* [ON ERROR *statements*]

file.variable specifies a file opened for sequential access. If *file.variable* evaluates to the null value, the WEOFSEQ statement fails and the program terminates with a run-time error message.

Note: On Windows NT systems, you cannot use the WEOFSEQ statement with a diskette drive that you opened with the OPENDEV statement. For 1/4- inch cartridge tape drives (60 MB or 150 MB) you can use WEOFSEQ to write an end-of-file (EOF) mark at the beginning of the data or after a write.

The ON ERROR clause

The ON ERROR clause is optional in the WEOFSEQ statement. The ON ERROR clause lets you specify an alternative for program termination when a fatal error is encountered during processing of the WEOFSEQ statement.

If a fatal error occurs, and the ON ERROR clause was not specified, or was ignored (as in the case of an active transaction), the following occurs:

- An error message appears.
- Any uncommitted transactions begun within the current execution environment roll back.
- The current program terminates.
- Processing continues with the next statement of the previous execution environment, or the program returns to the UniVerse prompt.

A fatal error can occur if any of the following occur:

- A file is not open.
- *file.variable* is the null value.
- A distributed file contains a part file that cannot be accessed.

If the ON ERROR clause is used, the value returned by the STATUS function is the error number.

See the OPENSEQ statement, on page 283, READSEQ statement, on page 310, and WRITESEQ statement, on page 454 for more information about sequential file processing.

Note: Some systems do not support the truncation of disk files. WEOFSEQ is ignored on these systems, except that WEOFSEQ always works at the beginning of a file.

Example

The following example writes an end-of-file mark on the record RECORD in the file TYPE1:

```
OPENSEQ 'TYPE1','RECORD' TO FILE ELSE STOP
```

```
WEOFSEQ FILE
```

WRITE statements

Use WRITE statements to write new data to a record in a UniVerse file. The value of *expression* replaces any data previously stored in the record.

Syntax

```
WRITE[U] expression {ON | TO} [file.variable,] record.ID
       [ON ERROR statements] [LOCKED statements]
       [THEN statements] [ELSE statements]
```

```
WRITEV[U] expression {ON | TO} [file.variable,] record.ID, field#
        [ON ERROR statements] [LOCKED statements]
        [THEN statements] [ELSE statements]
```

Use this statement...	To do this...
WRITE	Write to a record.
WRITEU	Write to a record, retaining an update record lock.
WRITEV	Write to a field.
WRITEVU	Write to a field, retaining an update record lock.

If *expression* evaluates to the null value, the WRITE statement fails and the program terminates with a run-time error message.

file.variable specifies an open file. If *file.variable* is not specified, the default file is assumed (for more information on default files, see the OPEN statement, on page 276. If the file is neither accessible nor open, the program terminates with a run-time error message, unless ELSE statements are specified.

The system searches the file for the record specified by *record.ID*. If the record is not found, WRITE creates a new record.

If *file.variable*, *record.ID*, or *field#* evaluates to the null value, all WRITE statements (WRITE, WRITEU, WRITEV, WRITEVU) fail and the program terminates with a run-time error message.

The new value is written to the record, and the THEN statements are executed. If no THEN statements are specified, execution continues with the statement following the WRITE statement. If WRITE fails, the ELSE statements are executed; any THEN statements are ignored.

When updating a record, the WRITE statement releases the update record lock set with a READU statement. To maintain the update record lock set by the READU statement, use a WRITEU statement instead of a WRITE statement.

The WRITE statement does not strip trailing field marks enclosing empty strings from *expression*. Use the MATWRITE statements if that operation is required.

Tables

If the file is a table, the effective user of the program must have SQL INSERT and UPDATE privileges to read records in the file. For information about the effective user of a program, see the AUTHORIZATION statement, on page 71.

If the OPENCHK configurable parameter is set to TRUE, or if the file is opened with the OPENCHECK statement, all SQL integrity constraints are checked for every write to an SQL table. If an integrity check fails, the WRITE statement uses the ELSE clause. Use the ICHECK function to determine what specific integrity constraint caused the failure.

NLS mode

WRITE and other BASIC statements that perform I/O operations map internal data to the external character set using the appropriate map for the output file.

UniVerse substitutes the file map's unknown character for any unmappable character. The results of the WRITE statements depend on the following:

- The inclusion of the ON ERROR clause
- The setting of the NLSWRITEELSE parameter in the *uvconfig* file
- The location of the unmappable character

The values returned by the `STATUS` function and the results are as follows:

STATUS value and results	ON ERROR and parameter setting	Unmappable character location
3 The WRITE fails, no records written. 4 The WRITE fails, no records written.	ON ERROR	Record ID Data
Program terminates with a run-time error message.	No ON ERROR, and NLSWRITEELSE = 1	Record ID or data
Program terminates with a run-time error message. Record is written with unknown characters; lost data.	No ON ERROR, NLSWRITEELSE = 0	Record ID Data

For more information about unmappable characters, see the *UniVerse NLS Guide*.

Use the STATUS function after a WRITE statement is executed, to determine the result of the operation, as follows:

Value	Description
0	The record was locked before the WRITE operation.
-2	The record was unlocked before the WRITE operation.
-3	The record failed an SQL integrity check.
-4	The record failed a trigger program.
–6	Failed to write to a published file while the subsystem was shut down.

The ON ERROR clause

The ON ERROR clause is optional in WRITE statements. Its syntax is the same as that of the ELSE clause. The ON ERROR clause lets you specify an alternative for program termination when a fatal error is encountered during processing of the WRITE statement.

If a fatal error occurs, and the ON ERROR clause was not specified or was ignored (as in the case of an active transaction), the following occurs:

- An error message appears.

- Any uncommitted transactions begun within the current execution environment roll back.
- The current program terminates.
- Processing continues with the next statement of the previous execution environment, or the program returns to the UniVerse prompt.

A fatal error can occur if any of the following occur:

- A file is not open.
- *file.variable* is the null value.
- A distributed file contains a part file that cannot be accessed.

If the ON ERROR clause is used, the value returned by the STATUS function is the error number.

The LOCKED clause

The LOCKED clause is optional, but recommended. Its format is the same as that of the ELSE clause.

The LOCKED clause handles a condition caused by a conflicting lock (set by another user) that prevents the WRITE statement from processing. The LOCKED clause is executed if one of the following conflicting locks exists:

- Exclusive file lock
- Intent file lock
- Shared file lock
- Update record lock
- Shared record lock

If the WRITE statement does not include a LOCKED clause, and a conflicting lock exists, the program pauses until the lock is released.

If a LOCKED clause is used, the value returned by the `STATUS` function is the terminal number of the user who owns the conflicting lock.

The WRITEU statement

Use the WRITEU statement to update a record without releasing the update record lock set by a previous READU statement (see the READ statements, on page 303). To release the update record lock set by a READU statement and maintained by a WRITEU statement, you must use a RELEASE statement, WRITE statements, MATWRITE statements, or WRITEV statement. If you do not explicitly release the lock, the record remains locked until the program executes the STOP statement. When more than one program or user could modify the same record, use a READU statement to lock the record before doing the WRITE or WRITEU.

If *expression* evaluates to the null value, the WRITEU statement fails and the program terminates with a run-time error message.

The WRITEV statement

Use the WRITEV statement to write a new value to a specified field in a record. The WRITEV statement requires that *field#* be specified. *field#* is the number of the field to which *expression* is written. It must be greater than 0. If either the record or the field does not exist, WRITEV creates them.

If *expression* evaluates to the null value, null is written to *field#*, provided that the field allows nulls. If the file is an SQL table, existing SQL security and integrity constraints must allow the write.

The WRITEVU statement

Use the WRITEVU statement to update a specified field in a record without releasing the update record lock set by a previous READU statement (see the READ statement). The WRITEVU syntax is like that of the WRITEV and WRITEU statements.

If *expression* evaluates to the null value, null is written to *field#*, provided that the field allows nulls. If the file is an SQL table, existing SQL security and integrity constraints must allow the write.

Remote files

If in a transaction you try to write to a remote file over UVNet, the write statement fails, the transaction is rolled back, and the program terminates with a run-time error message.

Example

```
CLEAR
 DATA "ELLEN","KRANZER","3 AMES STREET","CAMBRIDGE"
 DATA "MA","02139","SAILING"
 OPEN '','SUN.MEMBER' TO FILE ELSE
    PRINT "COULD NOT OPEN FILE"
    STOP
 END
 PRINT "ENTER YOUR FIRST NAME"
 INPUT FNAME
 PRINT "ENTER YOUR LAST NAME"
 INPUT LNAME
 PRINT "ENTER YOUR ADDRESS (PLEASE WAIT FOR PROMPTS)"
 PRINT "STREET ADDRESS"
 INPUT STREET
 PRINT "ENTER CITY"
 INPUT CITY
 PRINT "ENTER STATE"
 INPUT STATE
 PRINT "ENTER ZIP CODE"
 INPUT ZIP
 PRINT "ENTER YOUR INTERESTS"
 INPUT INTERESTS
 RECORD<1>=LNAME
 RECORD<2>=FNAME
 RECORD<3>=STREET
 RECORD<4>=CITY
 RECORD<5>=STATE
RECORD<6>=ZIP
 RECORD<7>=1989
 RECORD<8>=INTERESTS
 WRITE RECORD TO FILE, 1111
 PRINT
 EXECUTE 'LIST SUN.MEMBER LNAME WITH FNAME EQ ELLEN'
```

This is the program output:

```
ENTER YOUR FIRST NAME
 ?ELLENENTER YOUR LAST NAME
 ?KRANZERENTER YOUR ADDRESS (PLEASE WAIT FOR PROMPTS)
 STREET ADDRESS
 ?3 AMES STREETENTER CITY
 ?CAMBRIDGEENTER STATE
 ?MAENTER ZIP CODE
 ?02139ENTER YOUR INTEREST
 ?SAILING
```

```
SUN.MEMBER  LAST NAME.
 1111 KRANZER

 1 records listed.
```

WRITEBLK statement

Use the WRITEBLK statement to write a block of data to a file opened for sequential processing. Each WRITEBLK statement writes the value of *expression* starting at the current position in the file. The current position is incremented to beyond the last byte written. WRITEBLK does not add a newline at the end of the data.

Syntax

```
WRITEBLK expression ON file.variable
         {THEN statements [ELSE statements] | ELSE statements}
```

file.variable specifies a file opened for sequential processing.

> **Note:** On Windows NT systems, if you use the WRITEBLK statement to write to a 1/4-inch cartridge tape (60 MB or 150 MB) that you opened with the OPENDEV statement, on page 280, you must specify the block size as 512 bytes or a multiple of 512 bytes.

The value of *expression* is written to the file, and the THEN statements are executed. If no THEN statements are specified, program execution continues with the next statement.

If the file cannot be accessed or does not exist, the ELSE statements are executed; any THEN statements are ignored. If the device runs out of disk space, WRITEBLK takes the ELSE clause and returns –4 to the STATUS function.

If either *expression* or *file.variable* evaluates to the null value, the WRITEBLK statement fails and the program terminates with a run-time error message.

If NLS is enabled, the data written is mapped using the appropriate output file map. For more information about maps, see the *UniVerse NLS Guide*.

Example

```
OPENSEQ 'FILE.E','RECORD4' TO FILE ELSE ABORT
 WEOFSEQ FILE
 DATA1='ONE'
 DATA2='TWO'
 *
 WRITEBLK DATA1 ON FILE ELSE ABORT
 WRITEBLK DATA2 ON FILE ELSE ABORT
 * These two lines write two items to RECORD4 in FILE.E without
 * inserting a newline between them.
 WEOFSEQ FILE
 SEEK FILE,0,0 ELSE STOP
 READSEQ A FROM FILE THEN PRINT A
 * This reads and prints the line just written to the file.
```

This is the program output:

```
ONETWO
```

WRITELIST statement

Use the WRITELIST statement to save a list as a record in the &SAVEDLISTS& file.

Syntax

```
WRITELIST dynamic.array ON listname
```

dynamic.array is an expression that evaluates to a string made up of elements separated by field marks. It is the list to be saved.

listname is an expression that evaluates to *record.ID* or *record.ID account.name*

record.ID is the record ID of the select list created in the &SAVEDLISTS& file. If *listname* includes *account.name*, the &SAVEDLISTS& file of the specified account is used instead of the one in the local account. If *record.ID* exists, WRITELIST overwrites the contents of the record.

If either *dynamic.array* or *listname* evaluates to the null value, the WRITELIST statement fails and the program terminates with a run-time error message.

WRITESEQ statement

Use the WRITESEQ statement to write new lines to a file opened for sequential processing. UniVerse keeps a pointer to the current position in the file while it is open for sequential processing. The OPENSEQ statement sets this pointer to the first byte of the file, and it is advanced by the READSEQ statement, READBLK statement, WRITESEQ, and WRITEBLK statement.

WRITESEQ writes the value of *expression* followed by a newline to the file. The data is written at the current position in the file. The pointer is set to the position following the newline. If the pointer is not at the end of the file, WRITESEQ overwrites any existing data byte by byte (including the newline), starting from the current position.

file.variable specifies a file opened for sequential access.

The value of *expression* is written to the file as the next line, and the THEN statements are executed. If THEN statements are not specified, program execution continues with the next statement. If the specified file cannot be accessed or does not exist, the ELSE statements are executed; any THEN statements are ignored.

If *expression* or *file.variable* evaluates to the null value, the WRITESEQ statement fails and the program terminates with a run-time error message.

After executing a WRITESEQ statement, you can use the `STATUS` function to determine the result of the operation:

Value	Description
0	The record was locked before the WRITESEQ operation.
-2	The record was unlocked before the WRITESEQ operation.
–4	The write operation failed because the device ran out of disk space.

File buffering

Normally UniVerse uses buffering for sequential input and output operations. If you use the NOBUF statement after an OPENSEQ statement, buffering is turned off and writes resulting from the WRITESEQ statement are performed right away.

You can also use the FLUSH statement after a WRITESEQ statement to cause all buffers to be written right away.

For more information about buffering, see the FLUSH statement, on page 168 and NOBUF statement, on page 269.

Syntax

```
WRITESEQ expression {ON | TO} file.variable [ON ERROR statements]
        {THEN statements [ELSE statements] | ELSE statements}
```

The ON ERROR clause

The ON ERROR clause is optional in the WRITESEQ statement. The ON ERROR clause lets you specify an alternative for program termination when a fatal error is encountered while the WRITESEQ statement is being processed.

If a fatal error occurs, and the ON ERROR clause was not specified, or was ignored (as in the case of an active transaction), the following occurs:

- An error message appears.
- Any uncommitted transactions begun within the current execution environment roll back.
- The current program terminates.
- Processing continues with the next statement of the previous execution environment, or the program returns to the UniVerse prompt.

A fatal error can occur if any of the following occur:

- A file is not open.
- *file.variable* is the null value.
- A distributed file contains a part file that cannot be accessed.

If the ON ERROR clause is used, the value returned by the `STATUS` function is the error number.

If NLS is enabled, WRITESEQ and other BASIC statements that perform I/O operations always map internal data to the external character set using the appropriate map for the output file. For more information about maps, see the *UniVerse NLS Guide*.

Example

```
DATA 'NEW ITEM 1', 'NEW ITEM 2'
 OPENSEQ 'FILE.E', 'RECORD1' TO FILE ELSE ABORT
 READSEQ A FROM FILE ELSE STOP
 *
 FOR I=1 TO 2
    INPUT B
    WRITESEQ B TO FILE THEN PRINT B ELSE STOP
 NEXT
 *
CLOSESEQ FILE
 END
```

This is the program output:

```
?NEW ITEM 1
 NEW ITEM 1
 ?NEW ITEM 2
```

```
NEW ITEM 2
```

WRITESEQF statement

Use the WRITESEQF statement to write new lines to a file opened for sequential processing, and to ensure that data is physically written to disk (that is, not buffered) before the next statement in the program is executed. The sequential file must be open, and the end-of-file marker must be reached before you can write to the file. You can use the FILEINFO function to determine the number of the line about to be written.

Syntax

```
WRITESEQF expression {ON | TO} file.variable [ON ERROR statements]
        {THEN statements [ELSE statements] | ELSE statements}
```

Normally, when you write a record using the WRITESEQ statement, the record is moved to a buffer that is periodically written to disk. If a system failure occurs, you could lose all the updated records in the buffer. The WRITESEQF statement forces the buffer contents to be written to disk; the program does not execute the statement following the WRITESEQF statement until the buffer is successfully written to disk. A WRITESEQF statement following several WRITESEQ statements ensures that all buffered records are written to disk.

WRITESEQF is intended for logging applications and should not be used for general programming. It increases the disk I/O of your program and therefore degrades performance.

file.variable specifies a file opened for sequential access.

The value of *expression* is written to the file as the next line, and the THEN statements are executed. If THEN statements are not specified, program execution continues with the next statement.

If the specified file cannot be accessed or does not exist, the ELSE statements are executed; any THEN statements are ignored. If the device runs out of disk space, WRITESEQF takes the ELSE clause and returns –4 to the STATUS function.

If *expression* or *file.variable* evaluates to the null value, the WRITESEQF statement fails and the program terminates with a run-time error message.

If NLS is enabled, WRITESEQF and other BASIC statements that perform I/O operations always map internal data to the external character set using the appropriate map for the output file. For more information about maps, see the *UniVerse NLS Guide*.

The ON ERROR clause

The ON ERROR clause is optional in the WRITESEQF statement. Its syntax is the same as that of the ELSE clause. The ON ERROR clause lets you specify an alternative for program termination when a fatal error is encountered while the WRITESEQF statement is being processed.

If a fatal error occurs, and the ON ERROR clause was not specified, or was ignored (as in the case of an active transaction), the following occurs:

- An error message appears.
- Any uncommitted transactions begun within the current execution environment roll back.
- The current program terminates.
- Processing continues with the next statement of the previous execution environment, or the program returns to the UniVerse prompt.

A fatal error can occur if any of the following occur:

- A file is not open.
- *file.variable* is the null value.
- A distributed file contains a part file that cannot be accessed.

If the ON ERROR clause is used, the value returned by the `STATUS` function is the error number.

Values returned by the FILEINFO function

Key 14 (FINFO$CURRENTLINE) of the `FILEINFO` function can be used to determine the number of the line about to be written to the file.

Example

In the following example, the print statement following the WRITESEQF statement is not executed until the record is physically written to disk:

```
WRITESEQF ACCOUNT.REC TO ACCOUNTS.FILE
    THEN WRITTEN = TRUE
    ELSE STOP "ACCOUNTS.FILE FORCE WRITE ERROR"
 PRINT "Record written to disk."
```

writeSocket function

Use the `writeSocket`() function to write data to a socket connection.

Syntax

`writeSocket``(`*`socket_handle, socket_data, time_out, mode, actual_write_size`*`)`

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
socket_handle	A handle to the open socket.
socket_data	The data to be written to the socket.
time_out	The allowable time (in milliseconds) for blocking. This is ignored for a non-blocking write.
mode	0: using current mode 1: blocking mode (default) 2: non-blocking mode
actual_write_size	The number of characters actually written.

Return status

The following table describes the return status of each mode.

Mode	Return status
Blocking	The function will return only after all characters in *socket_data* are written to the socket.

Mode	Return status
Non-blocking	The function may return with fewer character written than the actual length (in the case that the socket is full).

Return codes

The following table describes the status of each return code.

Return code	Description
0	Success.
1-41	See Socket function error return codes, on page 584.
107	Encryption error.
108	Decryption error.

WRITET statement

Use the WRITET statement to write a tape record to tape. The value of *variable* becomes the next tape record. *variable* is an expression that evaluates to the text to be written to tape.

Syntax

```
WRITET [UNIT (mtu)] variable
       {THEN statements [ELSE statements] | ELSE statements}
```

The UNIT clause specifies the number of the tape drive unit. Tape unit 0 is used if no unit is specified. If the UNIT clause is used, *mtu* is an expression that evaluates to a code made up of three decimal digits, as shown in the following table:

Code	Available Options
m (mode)	0 = No conversion 1 = EBCDIC conversion 2 = Invert high bit 3 = Invert high bit and EBCDIC conversion
t (tracks)	0 = 9 tracks. Only 9-track tapes are supported.
u (unit number)	0 through 7

The *mtu* expression is read from right to left. If *mtu* evaluates to a one-digit code, it represents the tape unit number. If *mtu* evaluates to a two-digit code, the rightmost digit represents the unit number and the digit to its left is the track number.

If either *mtu* or *variable* evaluates to the null value, the WRITET statement fails and the program terminates with a run-time error message.

Each tape record is written completely before the next record is written. The program waits for the completion of data transfer to the tape before continuing.

Before a WRITET statement is executed, a tape drive unit must be attached (assigned) to the user. Use the `ASSIGN` command to assign a tape unit to a user. If no tape drive unit is attached or if the unit specification is incorrect, the ELSE statements are executed.

The largest record that the WRITET statement can write is system-dependent. If the actual record is larger, bytes beyond the system byte limit are not written.

Note: UniVerse BASIC does not generate tape labels for the tape file produced with the WRITET statement.

The STATUS function returns 1 if READT takes the ELSE clause, otherwise it returns 0.

If NLS is enabled, WRITET and other BASIC statements that perform I/O operations always map external data to the UniVerse internal character set using the appropriate map for the file. The map defines the external character set for the file that is used to input data on a keyboard, display data on a screen, and so on. For more information about maps, see the *UniVerse NLS Guide*.

PIOPEN flavor

If you have a program that specifies the syntax UNIT *ndmtu*, the *nd* elements are ignored by the compiler and no errors are reported.

Examples

The following example writes a record to tape drive 0:

```
RECORD=1S2S3S4
 WRITET RECORD ELSE PRINT "COULD NOT WRITE TO TAPE"
```

The next example writes the numeric constant 50 to tape drive 2, a 9-track tape with no conversion:

```
WRITET UNIT(002) "50" ELSE PRINT "COULD NOT WRITE"
```

WRITEU statement

Use the WRITEU statement to maintain an update record lock while performing the WRITE statement.

For details, see the WRITE statements, on page 449.

WRITEV statement

Use the WRITEV statement to write on the contents of a specified field of a record of a UniVerse file.

For details, see the WRITE statements, on page 449.

WRITEVU statement

Use the WRITEVU statement to maintain an update record lock while writing on the contents of a specified field of a record of a UniVerse file.

For details, see the WRITE statements, on page 449.

XDOMAddChild function

Finds the *xpathString* in the context *xmlHandle* in the DOM structure, and inserts a node as the last child of the found node. If the inserted node type is XDOM.ATTR.NODE, this node is inserted as an attribute.

Syntax

XDOMAddChild(*xmlHandle, xpathString, nsMap, nodeHandle, dupFlag,nodeType*)

> **Note:** This function is case-sensitive. If you want it to be case-insensitive, you must compile your programs using the BASIC command with the -i option.

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
xmlHandle	The handle to the context. [IN]
xpathString	Relative or absolute XPath string. [IN] The *xpathString* parameter uses the *in-encoding* parameter set in the system-level or account-level xmlconfig file, the XMLSETOPTIONS command, or the XMLSetOptions() API.
nsMap	The map of namespaces that resolves the prefixes in the *xpathString*. Format is xmlns=default_url xmlns:prefix1=prefix1_url xmlns:prefix2=prefix2_url For example: xmlns=http://myproject.mycompany.com xmlns:a_prefix=a.mycompany.com [IN] The *nsMap* parameter uses the *in-encoding* parameter set in the system-level or account-level xmlconfig file, the XMLSETOPTIONS command, or the XMLSetOptions() API.
nodeHandle	Handle to a DOM subtree. If *nodeHandle* points to a DOM document, all of its children are inserted, in the same order. [IN]
dupFlag	XDOM.DUP: Clones *nodeHandle*, and replaces it with the duplicate node. XDOM.NODUP: Replaces with the original node. The subtree is also removed from its original location. [IN]

Return codes

The return code indicates success or failure. The following table describes each return code.

Return code	Description
XML.SUCCESS	Function completed successfully.
XML.ERROR	An error occurred.
XML.INVALID.HANDLE	An invalid DOM handle was returned to the function.

XDOMAppend function

Finds the *xpathString* in the context *xmlHandle* in the DOM structure, and inserts *nodeHandle* into the DOM structure as the next sibling of the found node. If the inserted node type is XDOM.ATTR.NODE, this node is inserted as an attribute.

Syntax

XDOMAppend(*xmlHandle, xpathString, nsMap, nodeHandle, dupFlag*)

Note: This function is case-sensitive. If you want it to be case-insensitive, you must compile your programs using the BASIC command with the -i option.

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
xmlHandle	The handle to the context. [IN]
xpathString	Relative or absolute XPath string. [IN] The *xpathString* parameter uses the *in-encoding* parameter set in the system-level or account-level xmlconfig file, the XMLSETOPTIONS command, or the XMLSetOptions() API.
nsMap	The map of namespaces that resolves the prefixes in the *xpathString*. Format is xmlns=default_url xmlns:prefix1=prefix1_url xmlns:prefix2=prefix2_url For example: xmlns=http://myproject.mycompany.com xmlns:a_prefix=a.mycompany.com [IN] The *nsMap* parameter uses the *in-encoding* parameter set in the system-level or account-level xmlconfig file, the XMLSETOPTIONS command, or the XMLSetOptions() API.
nodeHandle	Handle to a DOM subtree. If *nodeHandle* points to a DOM document, all of its children are inserted, in the same order. [IN]
dupFlag	XDOM.DUP: Clones *nodeHandle*, and replaces it with the duplicate node. XDOM.NODUP: Replaces with the original node. The subtree is also removed from its original location. [IN]

Return codes

The return code indicates success or failure. The following table describes each return code.

Return code	Description
XML.SUCCESS	Function completed successfully.
XML.ERROR	An error occurred.

Return code	Description
XML.INVALID.HANDLE	An invalid DOM handle was returned to the function.

XDOMClone function

The `XDOMClone` function duplicates the DOM subtree specified by *xmlHandle* to a new subtree *newXmlHandle*. The duplicate node has no parent (*parentNode* returns null.).

Cloning an element copies all attributes and their values, including those generated by the XML processor, to represent defaulted attributes, but this method does not copy any text it contains unless it is a deep clone, since the text is contained in a child text node. Cloning any other type of node simply returns a copy of this node.

Syntax

XDOMClone(*xmlHandle, newXmlHandle, depth*)

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
xmlHandle	Handle to the DOM subtree. [IN]
newXmlHandle	Handle to the new DOM subtree. [IN]
depth	XDOM.FALSE: Clone only the node itself. XDOM.TRUE: Recursively clone the subtree under the specified node. [IN]

Return codes

The following table describes the status of each return code.

Return code	Description
XML.SUCCESS	The function completed successfully.
XML.ERROR	An error occurred.
XML.INVALID.HANDLE	An invalid DOM handle was returned to the function.

XDOMClose function

The `XDOMClose` function frees the DOM structure.

Syntax

XDOMClose(*domHandle*)

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
domHandle	Handle to the DOM structure. [IN]

Return codes

The following table describes the status of each return code.

Return code	Description
XML.SUCCESS	The function completed successfully.
XML.ERROR	An error occurred.
XML.INVALID.HANDLE	An invalid DOM handle was returned to the function.

XDOMCreateNode function

`XDOMCreateNode` creates a new node in the DOM structure.

Syntax

`XDOMCreateNode``(`*`xmlHandle, nodeName, nodeValue, nodeType, nodeHandle`*`)`

Note: This function is case-sensitive. If you want it to be case-insensitive, you must compile your programs using the `BASIC` command with the -i option.

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
xmlHandle	A handle to the DOM structure. This handle acts as the context when resolving the namespace_uri from the prefix or resolving the prefix from the namespace_uri. [IN]
nodeName	The name of the node to be created. [IN] The name can be in any of the following formats: ▪ Local_name ▪ prefix: local_name:namespace_uri ▪ prefix:local_name ▪ :local_name:namespace_uri The *nodeName* parameter uses the *in-encoding* parameter set in the system-level or account-level `xmlconfig` file, the `XMLSETOPTIONS` command, or the `XMLSetOptions`() API.
nodeValue	The string to hold the node value. [IN] The *nodeValue* parameter uses the *in-encoding* parameter set in the system-level or account-level `xmlconfig` file, the `XMLSETOPTIONS` command, or the `XMLSetOptions`() API.

Parameter	Description
nodeType	The type of the node to be created. Valid values are: XDOM.ELEMENT.NODE XDOM.ATTR.NODE XDOM.TEXT.NODE XDOM.CDATA.NODE XDOM.ENTITY.REF.NODE XDOM.ENTITY.NODE XDOM.PROC.INST.NODE XDOM.COMMENT.NODE XDOM.DOC.NODE XDOM.DOC.TYPE.NODE XDOM.DOC.FRAG.NODE XDOM.NOTATION.NODE XDOM.XML.DECL.NODE [IN]
nodeHandle	A handle to the node to be created in the DOM structure. [IN]

Return codes

The return code indicates success or failure. The following table describes each return code.

Return code	Description
XML.SUCCESS	Function completed successfully.
XML.ERROR	An error occurred.

XDOMCreateRoot function

The `XDOMCreateRoot` function creates a new DOM structure with root only. You can use the result handle in other functions where a DOM handle or node handle is needed.

Syntax

`XDOMCreateRoot``(`*`domHandle`*`)`

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
domHandle	Handle to the opened DOM structure. [OUT]

Return codes

The following table describes the status of each return code.

Return code	Description
XML.SUCCESS	The function completed successfully.
XML.ERROR	An error occurred.

XDOMEvaluate function

`XDOMEvaluate` returns the value of *xpathString* in the context *xmlHandle* in the DOM structure.

Syntax

`XDOMEvaluate``(`*`xmlHandle, xpathString, nsMap, aValue`*`)`

Note: This function is case-sensitive. If you want it to be case-insensitive, you must compile your programs using the `BASIC` command with the -i option.

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
xmlHandle	The handle to the context. [IN]
xpathString	Relative or absolute XPath string. [IN] The *xpathString* parameter uses the *in-encoding* parameter set in the system-level or account-level `xmlconfig` file, the `XMLSETOPTIONS` command, or the `XMLSetOptions`() API.

nsMap	The map of namespaces that resolves the prefixes in the *xpathString*.
	Format is
	`xmlns=default_url`
	`xmlns:prefix1=prefix1_url`
	`xmlns:prefix2=prefix2_url`
	For example:
	`xmlns=http://myproject.mycompany.com`
	`xmlns:a_prefix=a.mycompany.com`
	[IN]
	The *nsMap* parameter uses the *in-encoding* parameter set in the system-level or account-level `xmlconfig` file, the `XMLSETOPTIONS` command, or the `XMLSetOptions`() API.

aValue	The value of *xpathString*. [OUT] The *aValue* parameter uses the *out-encoding* parameter set in the system-level or account-level xmlconfig file, the `XMLSETOPTIONS` command, or the `XMLSetOptions`() API.

Return codes

The return code indicates success or failure. The following table describes each return code.

Return code	Description
XML.SUCCESS	Function completed successfully.

Return code	Description
XML.ERROR	An error occurred.
XML.INVALID.HANDLE	An invalid DOM handle was returned to the function.

XDOMGetAttribute function

`XDOMGetAttribute` gets the node's attribute node, whose attribute name is *attrName*.

Syntax

`XDOMGetAttribute``(`*`nodeHandle, attrName, nodeHandle`*`)`

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
nodeHandle	Handle to the DOM node. [IN]
attrName	Attribute name. [IN] The *attrName* parameter uses the *in-encoding* parameter set in the system-level or account-level `xmlconfig` file, the `XMLSETOPTIONS` command, or the `XMLSetOptions`() API.
nodeHandle	Handle to the found attribute node. [OUT]

Return codes

The return code indicates success or failure. The following table describes each return code.

Return code	Description
XML.SUCCESS	Function completed successfully.
XML.ERROR	An error occurred.
XML.INVALID.HANDLE	An invalid DOM handle was returned to the function.

XDOMGetChildNodes function

The `XDOMGetChildNodes` function returns all child nodes of *xmlHandle*.

Syntax

`XDOMGetChildNodes``(`*`xmlHandle, nodeListHandle`*`)`

This function behaves in the same way as:

`XDOMLocate``(`*`xmlHandle`*`, "*", "", XML.MULTI)`

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
xmlHandle	Handle to the DOM structure.

Parameter	Description
nodeListHandle	The handle to the node list.

Example

Consider the following XML document:

```
<?xml version="1.0" encoding="utf-8"?>
<ADDRBOOK cmt="my address book">
<ENTRY id="id1" name=”bookentry”>
            <NAME>Name One</NAME>
            <ADDRESS>101 Some Way</ADDRESS>
            <PHONENUM DESC="Work">303-111-1111</PHONENUM>
            <PHONENUM DESC="Fax">303-111-2222</PHONENUM>
            <PHONENUM DESC="Pager">303-111-3333</PHONENUM>
            <EMAIL>name.one@some.com</EMAIL>
</ENTRY>
<ENTRY ID="id2" NAME=”bookentry”>
<NAME>Name Two</NAME>
<ADDRESS>202 Some Way</ADDRESS>
            <PHONENUM DESC="Work">303-222-1111</PHONENUM>
            <PHONENUM DESC="Fax">303-222-2222</PHONENUM>
            <PHONENUM DESC="Home">303-222-3333</PHONENUM>
            <EMAIL>name.two@some.com</EMAIL>
</ENTRY>
</ADDRBOOK>
```

In this example, suppose *xmlHandle* points to `<ENTRY id=”id1” name=”bookentry”>`. After the call to `XDOMGetChildNodes(xmlHandle, nodehandle)`, *nodeHandle* should point to all child nodes, that is, <NAME>, <ADDRESS>, three <PHONENUM>’s, and <EMAIL>.

XDOMGetElementById function

The `XDOMGetElementById` function finds the first element with the ID you specify.

Syntax

`XDOMGetElementById``(`*`xmlHandle`*`,`*`idstr`*`,`*`nodeHandle`*`)`

This behaves in the same way as:

`XDOMLocate``(`*`xmlHandle`*`,.”//*[@ID=’idstr’ or @id=’idstr’]”,””,XML_SINGLE)`

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
xmlHandle	Handle to the DOM structure.
idstr	The ID of the element you want to return.
nodeHandle	Handle to the DOM node.

Example

```
<?xml version="1.0" encoding="utf-8"?>
<ADDRBOOK cmt="my address book">
```

```
<ENTRY id="id1" name=”bookentry”>
            <NAME>Name One</NAME>
            <ADDRESS>101 Some Way</ADDRESS>
            <PHONENUM DESC="Work">303-111-1111</PHONENUM>
            <PHONENUM DESC="Fax">303-111-2222</PHONENUM>
            <PHONENUM DESC="Pager">303-111-3333</PHONENUM>
            <EMAIL>name.one@some.com</EMAIL>
</ENTRY>
<ENTRY ID="id2" NAME=”bookentry”>
            <NAME>Name Two</NAME>
            <ADDRESS>202 Some Way</ADDRESS>
            <PHONENUM DESC="Work">303-222-1111</PHONENUM>
            <PHONENUM DESC="Fax">303-222-2222</PHONENUM>
            <PHONENUM DESC="Home">303-222-3333</PHONENUM>
            <EMAIL>name.two@some.com</EMAIL>
</ENTRY>
</ADDRBOOK>
```

In the example, suppose *xmlHandle* points to the document root. After the call to `XDOMGetElementById(xmlHandle, “id2”, nodeHandle)`, *nodeHandle* should point to element <ENTRY ID=”id2” NAME=”bookentry>.

XDOMGetElementsByName function

The `XDOMGetElementsByName` function tries to find all elements with the name you specify.

Syntax

`XDOMGetElementsByName``(xmlHandle, namestr, nodeListHandle)`

This function behaves in the same way as:

`XDOMLocate``(xmlHandle, “//*[@NAME=’namestr’ or @name=’namestr’]”, “”, XML.MULTI)`

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
xmlHandle	Handle to the DOM structure.
namestr	The name of the element you want to return.
nodeListHandle	The handle to the node list.

Example

Consider the following XML document:

```
<?xml version="1.0" encoding="utf-8"?>
<ADDRBOOK cmt="my address book">
<ENTRY id="id1" name=”bookentry”>
            <NAME>Name One</NAME>
            <ADDRESS>101 Some Way</ADDRESS>
            <PHONENUM DESC="Work">303-111-1111</PHONENUM>
            <PHONENUM DESC="Fax">303-111-2222</PHONENUM>
            <PHONENUM DESC="Pager">303-111-3333</PHONENUM>
            <EMAIL>name.one@some.com</EMAIL>
</ENTRY>
```

```
<ENTRY ID="id2" NAME=”bookentry”>
            <NAME>Name Two</NAME>
            <ADDRESS>202 Some Way</ADDRESS>
            <PHONENUM DESC="Work">303-222-1111</PHONENUM>
            <PHONENUM DESC="Fax">303-222-2222</PHONENUM>
            <PHONENUM DESC="Home">303-222-3333</PHONENUM>
            <EMAIL>name.two@some.com</EMAIL>
</ENTRY>
</ADDRBOOK>
```

In the example, suppose *xmlHandle* points to the document root. After the call to `XDOMGetElementsByName(xmlHandle, ”bookentry”, nodeHandle)`, *nodeHandle* should point to elements <ENTRY id=”id1” name=”bookentry”> and <ENTRY ID=”id2” NAME=”bookentry”>.

XDOMGetElementsByTag function

The `XDOMGetElementsByTag` function tries to find all elements with the tag name you specify.

Syntax

`XDOMGetElementsByTag``(xmlHandle, tagname, nodeListHandle)`

This function behaves in the same way as:

`XDOMLocate``(xmlHandle, “//tagname”, “”, XML.MULTI)`

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
xmlHandle	The input handle, xmlHandle, acts as the context when resolving the namespace uri from the prefix, or resolving the prefix from the namespace uri.
tagname	Tagname can be one of the following formats: ▪ Local_name ▪ Prefix:local_name
nodeListHandle	The handle to the node list.

Example

Consider the following XML document:

```
<?xml version="1.0" encoding="utf-8"?>
<ADDRBOOK cmt="my address book">
<ENTRY id="id1" name=”bookentry”>
            <NAME>Name One</NAME>
            <ADDRESS>101 Some Way</ADDRESS>
            <PHONENUM DESC="Work">303-111-1111</PHONENUM>
            <PHONENUM DESC="Fax">303-111-2222</PHONENUM>
            <PHONENUM DESC="Pager">303-111-3333</PHONENUM>
            <EMAIL>name.one@some.com</EMAIL>
</ENTRY>
<ENTRY ID="id2" NAME=”bookentry”>
            <NAME>Name Two</NAME>
            <ADDRESS>202 Some Way</ADDRESS>
```

```
                <PHONENUM DESC="Work">303-222-1111</PHONENUM>
                <PHONENUM DESC="Fax">303-222-2222</PHONENUM>
                <PHONENUM DESC="Home">303-222-3333</PHONENUM>
                <EMAIL>name.two@some.com</EMAIL>
</ENTRY>
</ADDRBOOK>
```

In this XML document, suppose *xmlHandle* points to the document root. After the call to `XDOMGetElementsByTag(xmlHandle, "PHONENUM", nodeHandle)`, *nodeHandle* should point to all PHONENUM elements.

XMLGetError function

The `XMLGetError` function returns the error code and error message after the previous XML API failed.

Syntax

XMLGetError(*errorCode, errorMessage*)

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
errorCode	The error code. [OUT]
errorMessage	The error message. [OUT]

Return codes

The following table describes the status of each return code.

Return code	Description
XML.SUCCESS	The function completed successfully.
XML.ERROR	An error occurred.

XDOMGetNodeName function

`XDOMGetNodeName` returns the node name.

Syntax

XDOMGetNodeName(*nodeHandle, nodeName*)

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
nodeHandle	Handle to the DOM node. [IN]

Parameter	Description
nodeName	String to store the node name. [OUT] The *nodeName* parameter uses the *out-encoding* parameter set in the system-level or account-level `xmlconfig` file, the `XMLSETOPTIONS` command, or the `XMLSetOptions`() API.

Return codes

The return code indicates success or failure. The following table describes each return code.

Return code	Description
XML.SUCCESS	Function completed successfully.
XML.ERROR	An error occurred.
XML.INVALID.HANDLE	An invalid DOM handle was returned to the function.

XDOMGetNodeType function

The `XDOMGetNodeType` function returns the node type.

Syntax

`XDOMGetNodeType``(`*`nodeHandle, nodeType`*`)`

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
nodeHandle	The handle to the DOM node. [IN]
nodeType	An integer to store the node type. [OUT]

Return codes

The following table describes the status of each return code.

Return code	Description
XML.SUCCESS	The function completed successfully.
XML.ERROR	An error occurred.
XML.INVALID.HANDLE	An invalid DOM handle was returned to the function.

XDOMGetNodeValue function

`XDOMGetNodeValue` gets the node value.

Syntax

`XDOMGetNodeValue``(`*`nodeHandle, nodeValue`*`)`

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
nodeHandle	Handle to the DOM node. [IN]
nodeValue	The string to hold the node value. [OUT] The *nodeValue* parameter uses the *out-encoding* parameter set in the system-level or account-level `xmlconfig` file, the `XMLSETOPTIONS` command, or the `XMLSetOptions`() API.

Return codes

The return code indicates success or failure. The following table describes each return code.

Return code	Description
XML.SUCCESS	Function completed successfully.
XML.ERROR	An error occurred.
XML.INVALID.HANDLE	An invalid DOM handle was returned to the function.

XDOMGetOwnerDocument function

The `XDOMGetOwnerDocument` function returns the DOM handle to which *nodeHandle* belongs.

Syntax

`XDOMGetOwnerDocument``(`*`nodeHandle, domHandle`*`)`

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
nodeHandle	Handle to the DOM node. [IN]
domHandle	Handle to the opened DOM structure. [OUT]

Return codes

The following table describes the status of each return code.

Return code	Description
XML.SUCCESS	The function completed successfully.
XML.ERROR	An error occurred.
XML.INVALID.HANDLE	An invalid DOM handle was returned to the function.

XDOMGetUserData function

The `XDOMGetUserData` function returns the user data associated with the node.

Syntax

XDOMGetUserData(*nodeHandle, userData*)

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
nodeHandle	The handle to the DOM node. [IN]
userData	String to hold the user data. [OUT]

Return codes

The following table describes the status of each return code.

Return code	Description
XML.SUCCESS	The function completed successfully.
XML.ERROR	An error occurred.
XML.INVALID.HANDLE	An invalid DOM handle was returned to the function.

XDOMItem function

The `XDOMItem` function returns the index-th item in the list.

Syntax

XDOMItem(*nodeListHandle, index, dataHandle, dataType*)

If the index is less than 1 or greater than the number of items in the list, use `Error(errorCode,errorMessage)` to return the error message “index out of bounds.”

Parameters

The following table describes each parameter syntax.

Parameter	Description
nodeListHandle	The handle to the node list.
index	The index item to return.
dataHandle	UniVerse stores the returned value, either a DOM handle or a string, in *dataHandle*.
dataType	The data type that is stored in *dataHandle*.

If nodeListHandle was generated from an API other than `XDOMQuery`(), the dataType must be XQ.ITEM.NODE (1). If nodeListHandle was generated by `XDOMQuery`(), the dataType could be XQ.ITEM.NODE(1), or a simple value type such as XQ.ITEM.ANY_SIMPLE_TYPE(2), XQ.ITEM.STRING(21).

The following list shows the data types available.

- XQ.ITEM.NODE (1)
- XQ.ITEM.ANY_SIMPLE_TYPE (2)
- XQ.ITEM.ANY_URI (3)
- XQ.ITEM.BASE_64_BINARY (4)

- XQ.ITEM.BOOLEAN (5)
- XQ.ITEM.DATA (6)
- XQ.ITEM.DATE_TIME (7)
- XQ.ITEM.DAY_TIME_DURATION (8)
- XQ.ITEM.DECIMAL (9)
- XQ.ITEM.DOUBLE (10)
- XQ.ITEM.DURATION (11)
- XQ.ITEM.FLOAT (12)
- XQ.ITEM.G_DAY (13)
- XQ.ITEM.G_MONTH (14)
- XQ.ITEM.G_MONTH_DAY (15)
- XQ.ITEM.G_YEAR (16)
- XQ.ITEM.G_YEAR_MONTH (17)
- XQ.ITEM.HEX_BINARY (18)
- XQ.ITEM.NOTATION (19)
- XQ.ITEM.QNAME (20)
- XQ.ITEM.STRING (21)
- XQ.ITEM.TIME (22)
- XQ.ITEM.UNTYPED_ATOMIC (23)
- XQ.ITEM.YEAR_MONTH_DURATION (24)

XDOMLength function

The `XDOMLength` function determines the number of nodes in the list. The range of the valid child node index is to 1 to *length*, inclusive.

Syntax

`XDOMLength``(`*`nodeListHandle, length`*`)`

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
nodeListHandle	The handle to the node list.
length	The length of the node list.

XDOMLocate function

`XDOMLocate` finds a starting point for relative XPath searching in context *xmlHandle* in the DOM structure. The *xpathString* should specify only one node; otherwise, this function will return an error.

Syntax

XDOMLocate(*xmlHandle, xpathString, nsMap, nodeHandle*

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
xmlHandle	A handle to the DOM structure. [IN]
xpathString	Relative or absolute XPath string. [IN] The *xpathString* parameter uses the *in-encoding* parameter set in the system-level or account-level `xmlconfig` file, the `XMLSETOPTIONS` command, or the `XMLSetOptions`() API.

nsMap	The map of namespaces that resolves the prefixes in the *xpathString*.
	Format is
	`xmlns=default_url`
	`xmlns:prefix1=prefix1_url`
	`xmlns:prefix2=prefix2_url`
	For example:
	`xmlns=http://myproject.mycompany.com`
	`xmlns:a_prefix=a.mycompany.com`
	[IN]
	The *nsMap* parameter uses the *in-encoding* parameter set in the system-level or account-level `xmlconfig` file, the `XMLSETOPTIONS` command, or the `XMLSetOptions`() API.

nodeHandle	Handle to the found node. [OUT]

Return codes

The return code indicates success or failure. The following table describes each return code.

Return code	Description
XML.SUCCESS	Function completed successfully.
XML.ERROR	An error occurred.
XML.INVALID.HANDLE	An invalid DOM handle was returned to the function.

Note: In this document, *xmlHandle* is a generic type, it can be *domHandle* or *nodeHandle*. *DomHandle* stands for a whole document, while *nodeHandle* stands for a subtree. *DomHandle* is also a *nodeHandle*.

XDOMLocateNode function

The `XDOMLocateNode` function traverses from *nodeHandle* and gets the next node according to direction and *childIndex*.

Syntax

XDOMLocateNode(*nodeHandle, direction, childIndex, nodeType, newNodeHandle*)

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
nodeHandle	The handle to the starting node. [IN]
direction	Direction to traverse. Valid values are: ▪ XDOM.PREV.SIBLING ▪ XDOM.NEXT.SIBLING ▪ XDOM.NEXT.SIBLING.WITH.SAME.NAME ▪ XDOM.PREV.SIBLING.WITH.SAME.NAME ▪ XDOM.PARENT ▪ XDOM.CHILD [IN]
childIndex	The index in the child array. Valid values are: ▪ XDOM.FIRST.CHILD ▪ XDOM.LAST.CHILD ▪ Positive Integer [IN]

Parameter	Description
nodeType	The type of node to be located. Valid values are: ▪ XDOM.NONE ▪ XDOM.ELEMENT.NODE ▪ XDOM.ATTR.NODE ▪ XDOM.TEXT.NODE ▪ XDOM.CDATA.NODE ▪ XDOM.ENTITY.REF.NODE ▪ XDOM.ENTITY.NODE ▪ XDOM.PROC.INST.NODE ▪ XDOM.COMMENT.NODE ▪ XDOM.DOC.NODE ▪ XDOM.DOC.TYPE.NODE ▪ XDOM.DOC.FRAG.NODE ▪ XDOM.NOTATION.NODE ▪ XDOM.XML.DECL.NODE If *nodeType* is not XDOM.NONE, UniVerse uses this argument, along with *direction* and *childIndex*, to get the right typed node. For example, if *direction* is XDOM.PREV.SIBLING, and *nodeType* is XDOM.ELEMENT.NODE, UniVerse finds the element node which is the first previous sibling of *nodeHandle*. If *direction* is XDOM.CHILD, *childIndex* is XDOM.FIRST.CHILD, and *nodeType* is XDOM.ELEMENT.NODE, UniVerse finds the element node which is the first element child of *nodeHandle*. If the *direction* is XDOM.CHILD, *childIndex* is 2, and *nodeType* is XDOM.ELEMENT.NODE, UniVerse finds the element node which is the second element child of *nodeHandle*. When the *direction* is XDOM.NEXT.SIBLING.WITH.SAME.NAME, XDOM.PREV.SIBLING.WITH.SAME.NAME, or XDOM.PARENT, this argument is not used. [IN]
newNodeHandle	Handle to the found node. [OUT]

Return codes

The following table describes the status of each return code.

Return code	Description
XML.SUCCESS	The function completed successfully.
XML.ERROR	An error occurred.
XML.INVALID.HANDLE	An invalid DOM handle was returned to the function.

XDOMOpen function

The `XDOMOpen` function reads an *xmlDocument* and creates DOM structure. If the DTD is included in the document, UniVerse validates the document. The *xmlDocument* can be from a string, or from a file, depending on the *docLocation* flag.

Syntax

XDOMOpen(*xmlDocument, docLocation, domHandle*)

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
xmlDocument	The XML document. [IN]
docLocation	A flag to specify whether *xmlDocument* is a string holding the XML document, or it is a file containing the XML document. Valid values are: ▪ XML.FROM.FILE ▪ XML.FROM.STRING [IN]
domHandle	Handle to the opened DOM structure. [OUT]

Return codes

The following table describes the status of each return code.

Return code	Description
XML.SUCCESS	The function completed successfully.
XML.ERROR	An error occurred.
XML.INVALID.HANDLE	An invalid DOM handle was returned to the function.

Option

When the XML does not have an encoding set in the declaration and the data in the document is not UTF-8, as of UniVerse 10.2 the encoding is assumed to be UTF-8, as shown in the following example:

```
<?xml version="1.0" ?>
<ROOT>
<PRODUCTS _ID = "M1000" PRODID = "M1000" LIST = "$1,990" DESCRIPTION = "Low cost
, entry level, light duty, monochrome copier"/>
</ROOT>
```

Since there is no encoding set in the declaration line, opening the file with the `XDOMOpen` function fails if there is a character from another encoding set (for example an ISO-8859-1 character) in the data.

Reading a file from a browser that has the wrong encoding of the data will also produce an error similar to the following example:

```
An invalid character was found in text content. Error processing
resource 'file:///C:/U2/UV/&XML&/example.xml'. Line 4, Po...
```

At UniVerse 11.1.14, new XML option, xdomopen-encoding, was added. This option specifies what encoding to use when there is no encoding defined in the declaration. When ‘xdomopen-encoding’ is not set, or is set to “”, UTF-8 is assumed.

XDOMQuery function

The `XDOMQuery` function runs *xquery* on the current document or document node you specify with *xmlHandle*.

Syntax

XDOMQuery(*xmlHandle, xquery, xqueryLocation, itemListHandle*)

Depending on *xqueryLocation, xquery* contains the query if *xqueryLocation* is XML.FROM.STRING. *xquery* uses a file name which contains the query if *xqueryLocation* is XML.FROM.FILE. The output *itemListHandle* is the resulting item lists.

XDOMRemove function

`XDOMRemove` finds the *xpathString* in the context *xmlHandle* in the DOM structure, and then removes the found node or its attribute with name *attrName*.

Syntax

XDOMRemove(*xmlHandle, xpathString, nsMap, attrName, nodeHandle*)

Note: This function is case-sensitive. If you want it to be case-insensitive, you must compile your programs using the `BASIC` command with the -i option.

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
xmlHandle	The handle to the context. [IN]
xpathString	Relative or absolute XPath string. [IN] The *xpathString* parameter uses the *in-encoding* parameter set in the system-level or account-level `xmlconfig` file, the `XMLSETOPTIONS` command, or the `XMLSetOptions`() API.
nsMap	The map of namespaces that resolves the prefixes in the *xpathString*. Format is `xmlns=default_url xmlns:prefix1=prefix1_url xmlns:prefix2=prefix2_url` For example: `xmlns=http://myproject.mycompany.com xmlns:a_prefix=a.mycompany.com` [IN] The *nsMap* parameter uses the *in-encoding* parameter set in the system-level or account-level `xmlconfig` file, the `XMLSETOPTIONS` command, or the `XMLSetOptions`() API.
attrName	The attribute name. [IN] The *attrName* parameter uses the *in-encoding* parameter set in the system-level or account-level xmlconfig file, the `XMLSETOPTIONS` command, or the `XMLSetOptions`() API.

Parameter	Description
nodeHandle	The removed node, if *nodeHandle* is not NULL. [OUT]

Return codes

The return code indicates success or failure. The following table describes each return code.

Return code	Description
XML.SUCCESS	Function completed successfully.
XML.ERROR	An error occurred.
XML.INVALID.HANDLE	An invalid DOM handle was returned to the function.

XDOMReplace function

`XDOMReplace` finds the *xpathString* in the context *xmlHandle* in the DOM structure, and replaces the found node with *nodeHandle*.

Syntax

`XDOMReplace``(`*`xmlHandle, xpathString, nsMap, nodeHandle, dupFlag`*`)`

Note: This function is case-sensitive. If you want it to be case-insensitive, you must compile your programs using the `BASIC` command with the -i option.

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
xmlHandle	The handle to the context. [IN]
xpathString	Relative or absolute XPath string. [IN] The *xpathString* parameter uses the *in-encoding* parameter set in the system-level or account-level `xmlconfig` file, the `XMLSETOPTIONS` command, or the `XMLSetOptions`() API.

nsMap	The map of namespaces that resolves the prefixes in the *xpathString*.
	Format is
	`xmlns=default_url`
	`xmlns:prefix1=prefix1_url`
	`xmlns:prefix2=prefix2_url`
	For example:
	`xmlns=http://myproject.mycompany.com`
	`xmlns:a_prefix=a.mycompany.com`
	[IN]
	The *nsMap* parameter uses the *in-encoding* parameter set in the system-level or account-level `xmlconfig` file, the `XMLSETOPTIONS` command, or the `XMLSetOptions`() API.

nodeHandle	Handle to a DOM subtree. If *nodeHandle* points to a DOM document, the found node is replaced by all of *nodeHandle* children, which are inserted in the same order. [IN]

Parameter	Description
dupFlag	XDOM.DUP: Clones *nodeHandle*, and replaces it with the duplicate node. XDOM.NODUP: Replaces with the original node. The subtree is also removed from its original location. [IN]

Return codes

The return code indicates success or failure. The following table describes each return code.

Return code	Description
XML.SUCCESS	Function completed successfully.
XML.ERROR	An error occurred.
XML.INVALID.HANDLE	An invalid DOM handle was returned to the function.

XDOMSetNodeValue function

`XDOMSetNodeValue` sets the node value.

Syntax

`XDOMSetNodeValue``(`*`nodeHandle, nodeValue`*`)`

Note: This function is case-sensitive. If you want it to be case-insensitive, you must compile your programs using the `BASIC` command with the -i option.

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
nodeHandle	Handle to the DOM node. [IN]
nodeValue	The string to hold the node value. [IN] The *nodeValue* parameter uses the *in-encoding* parameter set in the system-level or account-level `xmlconfig` file, the `XMLSETOPTIONS` command, or the `XMLSetOptions`() API.

Return codes

The return code indicates success or failure. The following table describes each return code.

Return code	Description
XML.SUCCESS	Function completed successfully.
XML.ERROR	An error occurred.
XML.INVALID.HANDLE	An invalid DOM handle was returned to the function.

XDOMSetUserData function

The `XDOMSetUserData` function sets the user data associated with the node.

Syntax

XDOMSetUserData(*nodeHandle, userData*)

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
nodeHandle	Handle to the DOM node. [IN]
userData	String to hold the user data. [IN]

Return codes

The following table describes the status of each return code.

Return code	Description
XML.SUCCESS	The function completed successfully.
XML.ERROR	An error occurred.
XML.INVALID.HANDLE	An invalid DOM handle was returned to the function.

XDOMTransform function

The `XDOMTransform` function transforms input DOM structure using the style sheet specified by *styleSheet* to output DOM structure.

Note: Beginning at 11.3.1, the UniVerse XML parser will only allow well-formed XML documents. Applications that use poorly-formed XML may not be able to use the `XDOMtransform` functionality. General XML functionality may also be impacted with poorly-formed XML documents.

Syntax

XDOMTransform(*domHandle, styleSheet, ssLocation, outDomHandle*)

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
domHandle	Handle to the opened DOM structure. [OUT]
styleSheet	Handle to the context [IN]
ssLocation	A flag to specify whether *styleSheet* contains style sheet itself, or is just the style sheet file name. Value values are: XML.FROM.FILE (default) XML.FROM.STRING [IN]
outDomHandle	Handle to the resulting DOM structure. [OUT]

Return codes

The following table describes the status of each return code.

Return code	Description
XML.SUCCESS	The function completed successfully.
XML.ERROR	An error occurred.
XML.INVALID.HANDLE	An invalid DOM handle was returned to the function.

XDOMValidate function

The `XDOMValidate` function validates the DOM document using the schema specified by *schFile*.

Syntax

`XDOMValidate``(`*`xmlDocument, docLocation, schFile, schLocation`*`)`

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
xmlDocument	The name of the XML document. [IN]
docLocation	A flag to specify whether *xmlDocument* is the document itself, or the document file name. Valid values are: ▪ XML.FROM.FILE (default) ▪ XML.FROM.STRING XML.FROM.DOM [IN]
schFile	The schema file.
schLocation	A flag to specify whether schFile is the schema itself, or the schema file name. Valid values are: ▪ XML.FROM.FILE (default) ▪ XML.FROM.STRING [IN]

Return codes

The following table describes the status of each return code.

Return code	Description
XML.SUCCESS	The function completed successfully.
XML.ERROR	An error occurred.
XML.INVALID.HANDLE	An invalid DOM handle was passed to the function.

XDOMWrite function

The `XDOMWrite` function writes the DOM structure to xmlDocument. xmlDocument can be a string or a file, depending on the value of the *docLocation* flag.

Syntax

XDOMWrite(*domHandle, xmlDocument, docLocation*)

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
domHandle	The handle to the opened DOM structure. [IN]
xmlDocument	The XML document [OUT]
docLocation	A flag to specify whether *xmlDocument* is an output string which should hold the XML document, or it is a file where the XML document should be written. Valid values are: ▪ XML.TO.FILE ▪ XML.TO.STRING [IN]

Return codes

The following table describes the status of each return code.

Return code	Description
XML.SUCCESS	The function completed successfully.
XML.ERROR	An error occurred.
XML.INVALID.HANDLE	An invalid DOM handle was passed to the function.

XLATE function

Use the XLATE function to return the contents of a field or a record in a UniVerse file. XLATE opens the file, reads the record, and extracts the specified data.

Syntax

XLATE ([DICT] *filename, record.ID, field#, control.code*)

filename is an expression that evaluates to the name of the remote file. If XLATE cannot open the file, a run-time error occurs, and XLATE returns an empty string.

record.ID is an expression that evaluates to the ID of the record to be accessed. If *record.ID* is multivalued, the translation occurs for each record ID and the result is multivalued (system delimiters separate data translated from each record).

field# is an expression that evaluates to the number of the field from which the data is to be extracted. If *field#* is -1, the entire record is returned, except for the record ID.

control.code is an expression that evaluates to a code specifying what action to take if data is not found or is the null value. The possible control codes are:

Code	Description
X	(Default) Returns an empty string if the record does not exist or data cannot be found.

Code	Description
V	Returns an empty string and produces an error message if the record does not exist or data cannot be found.
C	Returns the value of *record.ID* if the record does not exist or data cannot be found.
N	Returns the value of *record.ID* if the null value is found.

The returned value is lowered. For example, value marks in the original field become subvalue marks in the returned value. For more information, see the LOWER function, on page 249.

If *filename*, *record.ID*, or *field#* evaluates to the null value, the `XLATE` function fails and the program terminates with a run-time error message. If *control.code* evaluates to the null value, null is ignored and X is used.

The `XLATE` function is the same as the `TRANS` function.

Example

```
X=XLATE("VOC","EX.BASIC",1,"X")
 PRINT "X= ":X
 *
 FIRST=XLATE("SUN.MEMBER","6100",2,"X")


LAST=XLATE("SUN.MEMBER","6100",1,"X")
 PRINT "NAME IS ":FIRST:" ":LAST
```

This is the program output:

```
X= F BASIC examples file
 NAME IS BOB MASTERS
```

XMAPAppendRec

The `XMAPAppendRec` function formats the specified record from the UniVerse file as a U2XMAP dataset record and appends it to the U2XMAP dataset.

Syntax

XMAPAppendRec(*XMAPhandle, file_name, record*)

> **Note:** This function is case-sensitive. If you want it to be case-insensitive, you must compile your programs using the `BASIC` command with the -i option.

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
XMAPhandle	The handle to the U2XMAP dataset. The *XMAPhandle* parameter uses the *in-encoding* parameter set in the system-level or account-level xmlconfig file, the `XMLSETOPTIONS` command, or the `XMLSetOptions`() API for the input record value.

Parameter	Description
file_name	The name of the UniVerse file that is being mapped in the U2 XMAP dataset.
record	The data record formatted according to the dictionary record of the UniVerse file.

Return codes

The return code indicates success or failure. The following table describes each return code.

Return code	Description
XML_SUCCESS	The XML document was opened successfully.
XML_ERROR	An error occurred opening the XML document.
XML_INVALID_HANDLE	The XMAP dataset was invalid.

XMAPClose function

The `XMAPClose` function closes the U2XMAP dataset handle and frees all related structures and memory.

Syntax

`XMAPClose``(`*`XMAP_handle`*`)`

where *XMAP_handle* is the handle to the U2XMAP dataset.

Return values

The following table describes the return values of this function.

Return value	Description
XML_SUCCESS	The XML document was closed successfully.
XML_ERROR	An error occurred closing the XML document.
XML_INVALID_HANDLE	The XMAP dataset was invalid.

XMAPCreate Function

The `XMAPCreate` function creates an empty XML document for transferring data from the UniVerse database to XML according the mapping rules you define.

Syntax

`XMAPCreate``(`*`u2xmapping_rules, mapping_flag, XMAPhandle`*`)`

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
u2xmapping_rules	The name of the U2XMAP file, or the UniVerse BASIC variable containing the XML to Database mapping rules.

Parameter	Description
mapping_flag	A flag indicating if the mapping file is the U2XMAP file itself or a string located within the UniVerse BASIC program. Valid values are: ▪ XMAP.FROM.FILE - the mapping rules are contained in a U2XMAP file. ▪ XMAP.FROM.STRING - *u2xmapping_rules* is the name of the variable containing the mapping rules.
XMAPhandle	The handle to the XMAP dataset.

Return values

The following table describes the return values of this function.

Return value	Description
XML_SUCCESS	The XML document was created successfully.
XML_ERROR	An error occurred creating the XML document.
XML_INVALID_HANDLE	The XMAP dataset was invalid.

XMAPOpen function

The `XMAPOpen` function opens an XML document as a U2XMAP data set.

Syntax

XMAPOpen(*xml_document, doc_flag, u2xmapping_rules, u2xmap_flag, XMAPhandle*)

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
xml_document	The name of the XML document.
doc_flag	A flag defining the type of *xml_document*. Valid values are: ▪ XML.FROM.DOM - *xml_document* is a DOM handle. ▪ XML.FROM.FILE - *xml_document* is a file name. ▪ XML.FROM.STRING - *xml_document* is the name of a variable containing the XML document.
u2xmapping_rules	The name of the U2XMAP file, or the UniVerse Basic variable containing the XML to Database mapping rules.
u2xmap_flag	A flag indicating if the mapping file is the U2XMAP file itself or a string located within the UniVerse Basic program. Valid values are: ▪ XMAP.FROM.FILE - the mapping rules are contained in a U2XMAP file. ▪ XMAP.FROM.STRING - *u2xmap_flag* is the name of the variable containing the mapping rules.

Parameter	Description
XMAPhandle	The handle to the XMAP dataset. This API registers the current *in-encoding* and *out-encoding* parameters in the *XMAPhandle*. These parameters are used throughout the life of the *XMAPhandle*.

Return codes

The return code indicates success or failure. The following table describes each return code.

Return code	Description
XML_SUCCESS	The XML document was opened successfully.
XML_ERROR	An error occurred opening the XML document.

XMAPReadNext function

The `XMAPReadNext` function retrieves the next record from the U2XMAP dataset and formats it as a record of the UniVerse file that is being mapped.

Syntax

`XMAPReadNext``(`*`XMAPhandle, file_name, record`*`)`

Note: This function is case-sensitive. If you want it to be case-insensitive, you must compile your programs using the `BASIC` command with the -i option.

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
XMAPhandle	The U2XMAP dataset handle. The *XMAPhandle* parameter uses the *out-encoding* parameter set in the system-level or account-level `xmlconfig` file, the `XMLSETOPTIONS` command, or the XMLSetOptions() API for the record value.
file_name	The name of the UniVerse file that is being mapped in the U2XMAP dataset.
record	The data record formatted according to the dictionary record of the file.

Return codes

The return code indicates success or failure. The following table describes each return code.

Return code	Description
XML_SUCCESS	The XMAPReadNext was executed successfully.
XML_ERROR	An error occurred in executing XMAPReadNext.
XML_INVALID_HANDLE	U2 XMAP dataset handle was invalid.

Return code	Description
XML_EOF	The end of the U2XMAP dataset has been reached.

XMAPToXMLDoc function

The `XMAPToXMLDoc` function generates an XML document from the data in the U2XMAP dataset using the mapping rules you define. The XML document can be either an XML DOM handle or an XML document. UniVerse writes the data to a file or a UniVerse BASIC variable.

Syntax

XMAPToXMLDoc(*XMAPhandle, xmlfile, doc_flag*)

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
XMAPhandle	The handle to the U2XMAP dataset.
xmlfile	The name of the XML file, or the name of a UniVerse BASIC variable to hold the XML document.
doc_flag	Indicates where to write the XML document. Valid values are: ▪ XML.TO.DOM - Writes the XML document to an XML DOM handle. ▪ XML.TO.FILE - Writes the XML document to a file. ▪ XML.TO.STRING - Writes the XML document to a UniVerse BASIC variable.

Return values

The following table describes the return values of this function.

Return value	Description
XML_SUCCESS	The XML document was opened successfully.
XML_ERROR	An error occurred opening the XML document.
XML_INVALID_HANDLE	The XMAP dataset was invalid.

XMLError function

Use the `XMLError` function to get the last error message.

Syntax

XMLError(*errmsg*)

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
errmsg	The error message string, or one of the following return values: XML.SUCCESS: Success. XML.ERROR: Failed

XMLExecute function

The `XMLExecute` function enables you to create an XML document using the RetrieVe LIST statement or the UniVerse SQL SELECT statement from a UniVerse BASIC program.

Syntax

`XMLExecute``(`*`cmd, options, xmlvar, xsdvar`*`)`

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
cmd	Holds the text string of the RetrieVe LIST statement or the UniVerse SQL SELECT statement. [IN]

Parameter	Description	
options	Each XML-related option is separated by a field mark (@FM). If the option requires a value, the values are contained in the same field, separated by value marks (@VM).	
	WITHDTD	Creates a DTD and binds it with the XML document. By default, UniVerse creates an XML schema. However, if you include WITHDTD in your RetrieVe or UniVerse SQL statement, UniVerse does not create an XML schema, but only produces the DTD.
	ELEMENTS	The XML output is in element-centric format.
	'XMLMAPPING': @VM:'*mapping_file_ name*'	Specifies the mapping file containing transformation rules for display. This file must exist in the &XML& directory.
	'SCHEMA':@VM: '*type*'	The default schema format is ref type schema. You can use the SCHEMA attribute to define a different schema format.
	HIDEMV, HIDEMS	Normally, when UniVerse processes multivalued or multi-subvalued fields, UniVerse adds another level of elements to produce multiple levels of nesting. You have the option of disabling this additional level by adding the HIDEMV and HIDEMS attributes. When these options are on, the generated XML document and the associated DTD or XML schema have fewer levels of nesting.
	HIDEROOT	Allows you to specify to only create a segment of an XML document, for example, using the SAMPLE keyword and other conditional clauses. If you specify HIDEROOT, UniVerse only creates the record portion of the XML document, it does not create a DTD or XML schema.
	'RECORD':@VM: '*newrecords*'	The default record name is FILENAME_*record*. The record attribute in the ROOT element changes the record name.
	'ROOT':@VM: '*newroot*'	The default root element name in an XML document is ROOT. You can change the name of the root element as shown in the following example: root="*root_element_name*"
	TARGETNAMESPACE:@FM:' *namespaceURL*'	UniVerse displays the *targetnamespace* attribute in the XMLSchema as *targetNamespace*, and uses the URL you specify to define *schemaLocation*. If you define the *targetnamespace* and other explicit *namespace* definitions, UniVerse checks if the explicitly defined *namespace* has the same URL and the *targetnamespace*. If it does, UniVerse uses the *namespace* name to qualify the schema element, and the XML document element name.
	COLLAPSEMV, COLLAPSEMS	Normally, when UniVerse processes multivalued or multi-subvalued fields, UniVerse adds another level of elements to produce multiple levels of nesting. You have the option of disabling this additional level by adding the COLLAPSEMV and COLLAPSE MS attributes. When these options are on, the generated XML document and the associated DTD or XML Schema have fewer levels of nesting.

Parameter	Description
XmlVar	The name of the variable to which to write the generated XML document [OUT]
XsdVar	The name of the variable in which to store the XML Schema if one is generated along with the XML document. [OUT]

Example

The following example illustrates the `XMLExecute` function:

```
CMD="SELECT SEMESTER,COURSE_NBR FROM STUDENT;"
OPTIONS := "COLLAPSEMS"
OPTIONS := @FM:"HIDEROOT"
OPTIONS := @FM:"root":@VM:"mystudent"
OPTIONS :=@FM:"record":@VM:"myrecord"
OPTIONS :=@FM:"targetnamespace":@VM:"http://www.rocketsoftware.com"
OPTIONS := @FM:"elementformdefault"
STATUS = XMLEXECUTE(CMD,OPTIONS,XMLVAR,XSDVAR)
PRINT XSDVAR
PRINT XMLVAR
```

XMLTODB function

You can also populate the UniVerse database by calling the UniVerse BASIC `XMLTODB` function. `XMLTODB` does the same thing as the TCL XML.TODB command. It cannot transform data from a specific subtree in an XML document. If you want to transform specific data, use the XMAP API.

Syntax

XMLTODB(*xml_document, doc_flag, u2xmapping_rules, u2xmap_ flag, status*)

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
xml_document	The name of the XML document.
doc_flag	A flag defining the type of *xml_document*. Valid values are: ▪ XML.FROM.DOM - *xml_document* is a DOM handle. ▪ XML.FROM.FILE - *xml_document* is a file name. ▪ XML.FROM.STRING - *xml_document* is a string located within the UniVerse BASIC program.
u2xmapping_rules	The mapping rules for the XML document.
u2xmap_flag	A flag indicating if the mapping file is the U2XMAP file itself or a string located within the UniVerse BASIC program. Valid values are: ▪ XMAP.FROM.FILE - the mapping rules are contained in a U2XMAP file. ▪ XMAP.FROM.STRING - *u2xmapping_rules* is the name of the variable containing the mapping rules.
Status	The return status.

While the UniVerse BASIC function `XMLTODB`() provides and easy way of transferring data from an XML document to a set of related database files, you may want to have greater control over which

part of the XML document you want to use for transferring data. For example, XMLTODB() lets you start the data transfer from a particular sibling of the start node. An example of such finer control is transferring only the second school data and its dependent subtree to the database from the sample XML document. You can accomplish this using a combination of the DOM API functions and the XMAP API functions.

Example

The following example illustrates the use of XMLTODB:

```
*XMLTODB("myXMlFile",XML.FROM.FILE,"myMapFile",XML.FROM.FILE,STATUS)
```

XTD function

Use the XTD function to convert a string of hexadecimal characters to an integer. If *string* evaluates to the null value, null is returned.

Syntax

XTD (*string*)

Example

```
Y = "0019"
 Z = XTD (Y)
 PRINT Z
```

This is the program output:

```
25
```

Appendix A: Quick reference

This appendix is a quick reference for all UniVerse BASIC statements and functions. The statements and functions are grouped according to their uses:

- Compiler directives
- Declarations
- Assignments
- Program flow control
- File I/O
- Sequential file I/O
- Printer and terminal I/O
- Tape I/O
- Select lists
- String handling
- Data conversion and formatting
- NLS
- Mathematical functions
- Relational functions
- System
- Remote procedure calls
- Miscellaneous

Compiler directives

The following table describes compiler directive statements.

Command	Description
$* statement	Identifies a line as a comment line. Same as the !, $*, and REM statements.
! statement	Identifies a line as a comment line. Same as the *, $*, and REM statements.
#INCLUDE statement	Inserts and compiles UniVerse BASIC source code from another program into the program being compiled. Same as the $INCLUDE and INCLUDE statements.
$* statement	Identifies a line as a comment line. Same as the *, !, and REM statements.
$CHAIN statement	Inserts and compiles UniVerse BASIC source code from another program into the program being compiled.
$COPYRIGHT statement	Inserts comments into the object code header. (UniVerse supports this statement for compatibility with existing software.)
$DEFINE statement	Defines a compile time symbol.
$EJECT statement	Begins a new page in the listing record. (UniVerse supports this statement for compatibility with existing software.) Same as the $PAGE statement.

Command	Description
$IFDEF statement	Tests for the definition of a compile time symbol.
$IFNDEF statement	Tests for the definition of a compile time symbol.
$INCLUDE statement	Inserts and compiles UniVerse BASIC source code from another program into the program being compiled. Same as the #INCLUDE and INCLUDE statements.
$INSERT statement	Performs the same operation as $INCLUDE; the only difference is in the syntax. (UniVerse supports this statement for compatibility with existing software.)
$MAP statement	In NLS mode, specifies the map for the source code.
$OPTIONS statement	Sets compile time emulation of UniVerse flavors.
$PAGE statement	Begins a new page in the listing record. (UniVerse supports this statement for compatibility with existing software.) Same as the $EJECT statement.
EQUATE statement	Assigns a symbol as the equivalent of a variable, function, number, character, or string.
#INCLUDE statement	Inserts and includes the specified BASIC source code from another program into the program being compiled. Same as the #INCLUDE and $INCLUDE statements.
NULL statement	Indicates that no operation is to be performed.
REM statement	Identifies a line as a comment line. Same as the *, !, and $* statements.
$UNDEFINE statement	Removes the definition for a compile time symbol.

Declarations

The following table describes Declaration statements.

Command	Description
COMMON statement	Defines a storage area in memory for variables commonly used by programs and external subroutines.
DEFFUN statement	Defines a user-written function.
DIMENSION statement	Declares the name, dimensionality, and size constraints of an array variable.
FUNCTION statement	Identifies a user-written function.
PROGRAM statement	Identifies a program.
SUBROUTINE statement	Identifies a series of statements as a subroutine.

Assignments

The following table describes Assignment functions and statements.

Command	Description
ASSIGNED function	Determines if a variable is assigned a value.
CLEAR statement	Assigns a value of 0 to specified variables.
LET statement	Assigns a value to a variable.
MAT statement	Assigns a new value to every element of an array with one statement.

Command	Description
UNASSIGNED function	Determines if a variable is unassigned.

Program flow control

The following table describes Program Flow Control functions and statements.

Command	Description
ABORT statement	Terminates all programs and returns to the UniVerse command level.
BEGIN CASE statement	Indicates the beginning of a set of CASE statements.
CALL statement	Executes an external subroutine.
CASE statements	Alters program flow based on the results returned by expressions.
CHAIN command	Terminates a BASIC program and executes a UniVerse command.
CONTINUE	Transfers control to the next logical iteration of a loop.
END statement	Indicates the end of a program or a block of statements.
END CASE statement	Indicates the end of a set of CASE statements.
ENTER statement	Executes an external subroutine.
EXECUTE statement	Executes UniVerse sentences and paragraphs from within the BASIC program.
EXIT statement	Quits execution of a LOOP…REPEAT loop and branches to the statement following the REPEAT statement.
FOR statement	Allows a series of instructions to be performed in a loop a given number of times.
GOSUB statement	Branches to and returns from an internal subroutine.
GOTO statement	Branches unconditionally to a specified statement within the program or subroutine.
IF statement	Determines program flow based on the evaluation of an expression.
LOOP statement	Repeatedly executes a sequence of statements under specified conditions.
NEXT statement	Defines the end of a FOR…NEXT loop.
ON statement	Transfers program control to a specified internal subroutine or to a specified statement, under specified conditions.
PERFORM statement	Executes a specified UniVerse sentence, paragraph, menu, or command from within the BASIC program, and then returns execution to the statement following the PERFORM statement.
REPEAT statement	Repeatedly executes a sequence of statements under specified conditions.
RETURN statement	Transfers program control from an internal or external subroutine back to the calling program.
RETURN (value) statement	Returns a value from a user-written function.
STOP statement	Terminates the current program.
SUBR function	Returns the value of an external subroutine.
WHILE/UNTIL	Provides conditions under which the LOOP…REPEAT statement or FOR…NEXT statement terminates.

File I/O

The following table describes File I/O functions and statements.

Command	Description
AUTHORIZATION statement	Specifies the effective run-time UID (user identification) number of the program.
BEGIN TRANSACTION statement	Indicates the beginning of a set of statements that make up a single transaction.
BSCAN statement	Scans the leaf-nodes of a B-tree file (type 25) or a secondary index.
CLEARFILE statement	Erases all records from a file.
CLOSE statement	Writes data written to the file physically on the disk and releases any file or update locks.
COMMIT statement	Commits all changes made during a transaction, writing them to disk.
DELETE statements	Deletes a record from a UniVerse file.
DELETEU statement	Deletes a record from a previously opened file.
END TRANSACTION statement	Indicates where execution should continue after a transaction terminates.
FILELOCK statement	Sets a file update lock on an entire file to prevent other users from updating the file until this program releases it.
FILEUNLOCK statement	Releases file locks set by the FILELOCK statement.
INDICES function	Returns information about the secondary key indexes in a file.
MATREAD statements	Assigns the data stored in successive fields of a record from a UniVerse file to the consecutive elements of an array.
MATREADL statement	Sets a shared read lock on a record, then assigns the data stored in successive fields of the record to the consecutive elements of an array.
MATREADU statement	Sets an exclusive update lock on a record, then assigns the data stored in successive fields of the record to the consecutive elements of an array.
MATWRITE statements	Assigns the data stored in consecutive elements of an array to the successive fields of a record in a UniVerse file.
MATWRITEU statement	Assigns the data stored in consecutive elements of an array to the successive fields of a record in a UniVerse file, retaining any update locks set on the record.
OPEN statement	Opens a UniVerse file to be used in a BASIC program.
OPENPATH statement	Opens a file to be used in a BASIC program.
PROCREAD statement	Assigns the contents of the primary input buffer of the proc to a variable.
PROCWRITE statement	Writes the specified string to the primary input buffer of the proc that called your BASIC program.
READ statements	Assigns the contents of a record to a dynamic array variable.
READL statement	Sets a shared read lock on a record, then assigns the contents of the record to a dynamic array variable.
READU statement	Sets an exclusive update lock on a record, then assigns the contents of the record to a dynamic array variable.
READV statement	Assigns the contents of a field of a record to a dynamic array variable.

Command	Description
READVL statement	Sets a shared read lock on a record, then assigns the contents of a field of a record to a dynamic array variable.
READVU statement	Sets an exclusive update lock on a record, then assigns the contents of a field of the record to a dynamic array variable.
RECORDLOCKED function	Establishes whether or not a record is locked by a user.
RECORDLOCKL	Sets a shared read-only lock on a record in a file.
RECORDLOCKU	Locks the specified record to prevent other users from accessing it.
RELEASE statement	Unlocks records locked by READL, READU, READVL, READVU, MATREADL, MATREADU, MATWRITEV, WRITEV, or WRITEVU statements.
ROLLBACK statement	Rolls back all changes made during a transaction. No changes are written to disk.
SET TRANSACTION ISOLATION LEVEL statement	Sets the default transaction isolation level for your program.
TRANS function	Returns the contents of a field in a record of a UniVerse file.
TRANSACTION ABORT statement	Discards changes made during a transaction. No changes are written to disk.
TRANSACTION COMMIT statement	Commits all changes made during a transaction, writing them to disk.
TRANSACTION START statement	Indicates the beginning of a set of statements that make up a single transaction.
WRITE statements	Replaces the contents of a record in a UniVerse file.
WRITEU	Replaces the contents of the record in a UniVerse file without releasing the record lock
WRITEV	Replaces the contents of a field of a record in a UniVerse file.
WRITEVU	Replaces the contents of a field in the record without releasing the record lock.
XLATE function	Returns the contents of a field in a record of a UniVerse file.

Sequential file I/O

The following table describes the Sequential File I/O statements.

Command	Description
CLOSESEQ statement	Writes an end-of-file mark at the current location in the record and then makes the record available to other users.
CREATE statement	Creates a record in a UniVerse type 1 or type 19 file or establishes a path.
FLUSH statement	Immediately writes all buffers.
GET statements	Reads a block of data from an input stream associated with a device, such as a serial line or terminal.
GETX statement	Reads a block of data from an input stream associated with a device, and returns the characters in ASCII hexadecimal format.
NOBUF statement	Turns off buffering for a sequential file.
OPENSEQ statement	Prepares a UniVerse file for sequential use by the BASIC program.
READBLK statement	Reads a block of data from a UniVerse file open for sequential processing and assigns it to a variable.

Command	Description
READSEQ statement	Reads a line of data from a UniVerse file opened for sequential processing and assigns it to a variable.
SEND statement	Writes a block of data to a device that has been opened for I/O using OPENDEV or OPENSEQ.
STATUS statement	Determines the status of a UniVerse file open for sequential processing.
TIMEOUT statement	Terminates READSEQ or READBLK if no data is read in the specified time.
TTYCTL statement	Controls sequential file interaction with a terminal device.
TTYGET statement	Gets a dynamic array of the terminal characteristics of a terminal, line printer channel, or magnetic tape channel.
TTYSET statement	Sets the terminal characteristics of a terminal, line printer channel, or magnetic tape channel.
WEOFSEQ statement	Writes an end-of-file mark to a UniVerse file open for sequential processing at the current position.
WRITEBLK statement	Writes a block of data to a record in a sequential file.
WRITESEQ statement	Writes new values to the specified record of a UniVerse file sequentially.
WRITESEQF statement	Writes new values to the specified record of a UniVerse file sequentially and ensures that the data is written to disk.

Printer and terminal I/O

The following table describes the Printer and Terminal I/O functions and statements.

Command	Description
@ function	Returns an escape sequence used for terminal control.
BREAK statement	Enables or disables the Break key on the keyboard.
CLEARDATA statement	Clears all data previously stored by the DATA statement.
CRT statement	Outputs data to the screen.
DATA statement	Stores values to be used in subsequent requests for data input.
DISPLAY statement	Outputs data to the screen.
ECHO statement	Controls the display of input characters on the terminal screen.
FOOTING statement	Specifies text to be printed at the bottom of each page.
HEADING statement	Specifies text to be printed at the top of each page.
HUSH statement	Suppresses all text normally sent to a terminal during processing.
INPUT statement	Allows data input from the keyboard during program execution.
INPUT @ statement	Positions the cursor at a specified location and defines the length of the input field.
INPUTCLEAR statement	Clears the type-ahead buffer.
INPUTDISP statement	Positions the cursor at a specified location and defines a format for the variable to print.
INPUTERR statement	Prints a formatted error message from the ERRMSG file on the bottom line of the terminal.
INPUTNULL statement	Defines a single character to be recognized as the empty string in an INPUT @ statement.

Command	Description
INPUTTRAP statement	Branches to a program label or subroutine on a TRAP key.
KEYEDIT statement	Assigns specific editing functions to the keys on the keyboard to be used with the INPUT statement.
KEYEXIT statement	Specifies exit traps for the keys assigned editing functions by the KEYEDIT statement.
KEYIN function	Reads a single character from the input buffer and returns it.
KEYTRAP statement	Specifies traps for the keys assigned specific functions by the KEYEDIT statement.
OPENDEV statement	Opens a device for input or output.
$PAGE statement	Prints a footing at the bottom of the page, advances to the next page, and prints a heading at the top.
PRINT statement	Outputs data to the terminal screen or to a printer.
PRINTER CLOSE	Indicates the completion of a print file and readiness for the data stored in the system buffer to be printed on the line printer.
PRINTER ON | OFF	Indicates whether print file 0 is to output to the terminal screen or to the line printer.
PRINTER RESET	Resets the printing options.
PRINTERR statement	Prints a formatted error message from the ERRMSG file on the bottom line of the terminal.
PROMPT statement	Defines the prompt character for user input.
TABSTOP statement	Sets the current tabstop width for PRINT statements.
TERMINFO function	Accesses the information contained in the *terminfo* files.
TPARM function	Evaluates a parameterized *terminfo* string.
TPRINT statement	Sends data with delays to the screen, a line printer, or another specified print file (that is, a logical printer).

Tape I/O

The following table describes the Tape I/O statements.

Command	Description
READT statement	Assigns the contents of the next record from a magnetic tape unit to the named variable.
REWIND statement	Rewinds the magnetic tape to the beginning of the tape.
WEOF statement	Writes an end-of-file mark to a magnetic tape.
WRITET	Writes the contents of a record onto magnetic tape.

Select lists

The following table describes Select Lists functions and statements.

Command	Description
CLEARSELECT statement	Sets a select list to empty.
DELETELIST statement	Deletes a select list saved in the &SAVEDLISTS& file.

Command	Description
GETLIST statement	Activates a saved select list so it can be used by a READNEXT statement.
READLIST statement	Assigns an active select list to a variable.
READNEXT statement	Assigns the next record ID from an active select list to a variable.
SELECT statements	Creates a list of all record IDs in a UniVerse file for use by a subsequent READNEXT statement. SELECT, SELECTN, and SELECTV are included in the SELECT statements.
SELECTE statement	Assigns the contents of select list 0 to a variable.
SELECTINDEX statement	Creates select lists from secondary key indexes.
SELECTINFO function	Returns the activity status of a select list.
SSELECT statement	Creates a sorted list of all record IDs from a UniVerse file.
WRITELIST statement	Saves a list as a record in the &SAVEDLISTS& file.

String handling

The following table describes the String Handling functions and statements.

Command	Description
ALPHA function	Determines whether the expression is an alphabetic or non-alphabetic string.
CATS function	Concatenates elements of two dynamic arrays.
CHANGE function	Substitutes an element of a string with a replacement element.
CHECKSUM function	Returns a cyclical redundancy code (a checksum value).
COL1 function	Returns the column position immediately preceding the selected substring after a BASIC `FIELD` function is executed.
COL2 function	Returns the column position immediately following the selected substring after a BASIC `FIELD` function is executed.
COMPARE function	Compares two strings for sorting.
CONVERT statement	Converts specified characters in a string to designated replacement characters.
CONVERT function	Replaces every occurrence of specified characters in a variable with other specified characters.
COUNT function	Evaluates the number of times a substring is repeated in a string.
COUNTS function	Evaluates the number of times a substring is repeated in each element of a dynamic array.
DCOUNT function	Evaluates the number of delimited fields contained in a string.
DEL statement	Deletes the specified field, value, or subvalue from a dynamic array.
DELETE function	Deletes a field, value, or subvalue from a dynamic array.
DOWNCASE function	Converts all uppercase letters in an expression to lowercase.
DQUOTE function	Encloses an expression in double quotation marks.
EREPLACE function	Substitutes an element of a string with a replacement element.
EXCHANGE function	Replaces one character with another or deletes all occurrences of a specific character.
EXTRACT function	Extracts the contents of a specified field, value, or subvalue from a dynamic array.

Command	Description
FIELD function	Examines a string expression for any occurrence of a specified delimiter and returns a substring that is marked by that delimiter.
FIELDS function	Examines each element of a dynamic array for any occurrence of a specified delimiter and returns substrings that are marked by that delimiter.
FIELDSTORE function	Replaces, deletes, or inserts substrings in a specified character string.
FIND statement	Locates a given occurrence of an element within a dynamic array.
FINDSTR statement	Locates a given occurrence of a substring.
FOLD function	Divides a string into a number of shorter sections.
GETREM function	Returns the numeric value for the position of the REMOVE pointer associated with a dynamic array.
GROUP function	Returns a substring that is located between the stated number of occurrences of a delimiter.
GROUPSTORE statement	Modifies existing character strings by inserting, deleting, or replacing substrings that are separated by a delimiter character.
INDEX function	Returns the starting column position of a specified occurrence of a particular substring within a string expression.
INDEXS function	Returns the starting column position of a specified occurrence of a particular substring within each element of a dynamic array.
INS statement	Inserts a specified field, value, or subvalue into a dynamic array.
INSERT function	Inserts a field, value, or subvalue into a dynamic array.
LEFT function	Specifies a substring consisting of the first *n* characters of a string.
LEN function	Calculates the length of a string.
LENS function	Calculates the length of each element of a dynamic array.
LOCATE statement (IDEAL and REALITY syntax)	Searches a dynamic array for a particular value or string, and returns the index of its position.
LOWER function	Converts system delimiters that appear in expressions to the next lower-level delimiter.
MATBUILD statement	Builds a string by concatenating the elements of an array.
MATCHFIELD function	Returns the contents of a substring that matches a specified pattern or part of a pattern.
MATPARSE statement	Assigns the elements of an array from the elements of a dynamic array.
QUOTE function	Encloses an expression in double quotation marks.
RAISE function	Converts system delimiters that appear in expressions to the next higher-level delimiter.
REMOVE statement	Removes substrings from a dynamic array.
REMOVE function	Successively removes elements from a dynamic array. Extracts successive fields, values, etc., for dynamic array processing.
REVREMOVE statement	Successively removes elements from a dynamic array, starting from the last element and moving right to left. Extracts successive fields, values, etc., for dynamic array processing.
REPLACE function	Replaces all or part of the contents of a dynamic array.
REUSE function	Reuses the last value in the shorter of two multivalue lists in a dynamic array operation.
RIGHT function	Specifies a substring consisting of the last *n* characters of a string.

Command	Description
SETREM statement	Sets the position of the REMOVE pointer associated with a dynamic array.
SOUNDEX function	Returns the soundex code for a string.
SPACE function	Generates a string consisting of a specified number of blank spaces.
SPACES function	Generates a dynamic array consisting of a specified number of blank spaces for each element.
SPLICE function	Inserts a string between the concatenated values of corresponding elements of two dynamic arrays.
SQUOTE function	Encloses an expression in single quotation marks.
STR function	Generates a particular character string a specified number of times.
STRS function	Generates a dynamic array whose elements consist of a character string repeated a specified number of times.
SUBSTRINGS function	Creates a dynamic array consisting of substrings of the elements of another dynamic array.
TRIM function	Deletes extra blank spaces and tabs from a character string.
TRIMB function	Deletes all blank spaces and tabs after the last non-blank character in an expression.
TRIMBS function	Deletes all trailing blank spaces and tabs from each element of a dynamic array.
TRIMF function	Deletes all blank spaces and tabs up to the first non-blank character in an expression.
TRIMFS function	Deletes all leading blank spaces and tabs from each element of a dynamic array.
TRIMS function	Deletes extra blank spaces and tabs from the elements of a dynamic array.
UPCASE function	Converts all lowercase letters in an expression to uppercase.

Data conversion and formatting

The following table describes the Data Conversion and Formatting functions and statements.

Command	Description
ASCII function	Converts EBCDIC representation of character string data to the equivalent ASCII character code values.
CHAR function	Converts a numeric value to its ASCII character string equivalent.
CHARS function	Converts numeric elements of a dynamic array to their ASCII character string equivalents.
DTX function	Converts a decimal integer into its hexadecimal equivalent.
EBCDIC function	Converts data from its ASCII representation to the equivalent code value in `EBCDIC`.
FIX function	Rounds an expression to a decimal number having the accuracy specified by the PRECISION statement.
FMT function	Converts data from its internal representation to a specified format for output.
FMTS function	Converts elements of a dynamic array from their internal representation to a specified format for output.
ICONV function	Converts data to internal storage format.

Command	Description
ICONVS function	Converts elements of a dynamic array to internal storage format.
OCONV function	Converts data from its internal representation to an external output format.
OCONVS function	Converts elements of a dynamic array from their internal representation to an external output format.
PRECISION statement	Sets the maximum number of decimal places allowed in the conversion from the internal binary format of a numeric value to the string representation.
SEQ function	Converts an ASCII character code value to its corresponding numeric value.
SEQS function	Converts each element of a dynamic array from an ASCII character code to a corresponding numeric value.
XTD function	Converts a hexadecimal string into its decimal equivalent.

NLS

The following table describes the NLS functions and statements.

Command	Description
$MAP statement	Directs the compiler to specify the map for the source code.
AUXMAP statement	Assigns the map for the auxiliary printer to print unit 0 (i.e., the terminal).
BYTE function	Generates a string made up of a single byte.
BYTELEN function	Generates the number of bytes contained in the string value in an expression.
BYTETYPE function	Determines the function of a byte in a character.
BYTEVAL function	Retrieves the value of a byte in a string value in an expression.
FMTDP function	Formats data for output in display positions rather than character lengths.
FMTSDP function	Formats elements of a dynamic array for output in display positions rather than character lengths.
FOLDDP function	Divides a string into a number of substrings separated by field marks, in display positions rather than character lengths.
GETLOCALE function	Retrieves the names of specified categories of the current locale.
INPUTDISP statement	Lets the user enter data in display positions rather than character lengths.
LENDP function	Returns the number of display positions in a string.
LENSDP function	Returns a dynamic array of the number of display positions in each element of a dynamic array.
LOCALEINFO function	Retrieves the settings of the current locale.
SETLOCALE function	Changes the setting of one or all categories for the current locale.
UNICHAR function	Generates a character from a Unicode integer value.
UNICHARS function	Generates a dynamic array from an array of Unicode values.
UNISEQ function	Generates a Unicode integer value from a character.
UNISEQS function	Generates an array of Unicode values from a dynamic array.

Command	Description
UPRINT statement	Prints data without performing any mapping. Typically used with data that has already been mapped using OCONV (*mapname*).

Mathematical functions

The following table describes mathematical functions and statements

Function	Description
ABS function	Returns the absolute (positive) numeric value of an expression.
ABSS function	Creates a dynamic array containing the absolute values of a dynamic array.
ACOS function	Calculates the trigonometric arc-cosine of an expression.
ADDS function	Adds elements of two dynamic arrays.
ASIN function	Calculates the trigonometric arc-sine of an expression.
ATAN function	Calculates the trigonometric arctangent of an expression.
BITAND function	Performs a bitwise AND of two integers.
BITNOT function	Performs a bitwise NOT of two integers.
BITOR function	Performs a bitwise OR of two integers.
BITRESET function	Resets one bit of an integer.
BITSET function	Sets one bit of an integer.
BITTEST function	Tests one bit of an integer.
BITXOR function	Performs a bitwise XOR of two integers.
COS function	Calculates the trigonometric cosine of an angle.
COSH function	Calculates the hyperbolic cosine of an expression.
DIV function	Outputs the whole part of the real division of two real numbers.
DIVS function	Divides elements of two dynamic arrays.
EXP function	Calculates the result of base "e" raised to the power designated by the value of the expression.
INT function	Calculates the integer numeric value of an expression.
FADD function	Performs floating-point addition on two numeric values. This function is provided for compatibility with existing software.
FDIV function	Performs floating-point division on two numeric values.
FFIX function	Converts a floating-point number to a string with a fixed precision. FFIX is provided for compatibility with existing software.
FFLT function	Rounds a number to a string with a precision of 14.
FMUL function	Performs floating-point multiplication on two numeric values. This function is provided for compatibility with existing software.
FSUB function	Performs floating-point subtraction on two numeric values.
LN function	Calculates the natural logarithm of an expression in base "e".
MAXIMUM function	Returns the element with the highest numeric value in a dynamic array.
MINIMUM function	Returns the element with the lowest numeric value in a dynamic array.
MOD function	Calculates the modulo (the remainder) of two expressions.

Function	Description
MODS function	Calculates the modulo (the remainder) of elements of two dynamic arrays.
MULS function	Multiplies elements of two dynamic arrays.
NEG function	Returns the arithmetic additive inverse of the value of the argument.
NEGS function	Returns the negative numeric values of elements in a dynamic array. If the value of an element is negative, the returned value is positive.
NUM function	Returns true (1) if the argument is a numeric data type; otherwise, returns false (0).
NUMS function	Returns true (1) for each element of a dynamic array that is a numeric data type; otherwise, returns false (0).
PWR function	Calculates the value of an expression when raised to a specified power.
RANDOMIZE statement	Initializes the RND function to ensure that the same sequence of random numbers is generated after initialization.
REAL function	Converts a numeric expression into a real number without loss of accuracy.
REM function	Calculates the value of the remainder after integer division is performed.
RND function	Generates a random number between zero and a specified number minus one.
SADD function	Adds two string numbers and returns the result as a string number.
SCMP function	Compares two string numbers.
SDIV function	Outputs the quotient of the whole division of two integers.
SIN function	Calculates the trigonometric sine of an angle.
SINH function	Calculates the hyperbolic sine of an expression.
SMUL function	Multiplies two string numbers.
SQRT function	Calculates the square root of a number.
SSUB function	Subtracts one string number from another and returns the result as a string number.
SUBS function	Subtracts elements of two dynamic arrays.
SUM function	Calculates the sum of numeric data within a dynamic array.
SUMMATION function	Adds the elements of a dynamic array.
TAN function	Calculates the trigonometric tangent of an angle.
TANH function	Calculates the hyperbolic tangent of an expression.

.

Relational functions

The following table describes the Relational functions.

Function	Description
ANDS function	Performs a logical AND on elements of two dynamic arrays.
EQS function	Compares the equality of corresponding elements of two dynamic arrays.

Function	Description
GES function	Indicates when elements of one dynamic array are greater than or equal to corresponding elements of another dynamic array.
GTS function	Indicates when elements of one dynamic array are greater than corresponding elements of another dynamic array.
IFS function	Evaluates a dynamic array and creates another dynamic array on the basis of the truth or falsity of its elements.
ISNULL function	Indicates when a variable is the null value.
ISNULLS function	Indicates when an element of a dynamic array is the null value.
LES function	Indicates when elements of one dynamic array are less than or equal to corresponding elements of another dynamic array.
LTS function	Indicates when elements of one dynamic array are less than corresponding elements of another dynamic array.
NES function	Indicates when elements of one dynamic array are not equal to corresponding elements of another dynamic array.
NOT function	Returns the complement of the logical value of an expression.
NOTS function	Returns the complement of the logical value of each element of a dynamic array.
ORS function	Performs a logical OR on elements of two dynamic arrays.

System

The following table describes the System functions and statements.

Function	Description
DATE function	Returns the internal system date.
DEBUG statement	Invokes RAID, the interactive UniVerse BASIC debugger.
ERRMSG statement	Prints a formatted error message from the ERRMSG file.
INMAT function	Used with the MATPARSE, MATREAD, and MATREADU statements to return the number of array elements or with the OPEN statement to return the modulo of a file.
ITYPE function	Returns the value resulting from the evaluation of an I-descriptor.
LOCK statement	Sets an execution lock to protect user-defined resources or events from being used by more than one concurrently running program.
NAP statement	Suspends execution of a BASIC program, pausing for a specified number of milliseconds.
SENTENCE function	Returns the stored sentence that invoked the current process.
SLEEP statement	Suspends execution of a BASIC program, pausing for a specified number of seconds.
STATUS function	Reports the results of a function or statement previously executed.
SYSTEM function	Checks the status of a system function.
TIME function	Returns the time in internal format.
TIMEDATE function	Returns the time and date.
UNLOCK statement	Releases an execution lock that was set with the LOCK statement.
USERINFO function, on page 445	Gets the pid, user number, and more for the pid or user number specified.

Remote procedure calls

The following table describes Remote Procedure Call functions.

Function	Description
RPC.CALL function	Sends requests to a remote server.
RPC.CONNECT function	Establishes a connection with a remote server process.
RPC.DISCONNECT function	Disconnects from a remote server process.

Miscellaneous

The following table describes Miscellaneous functions and statements.

Function	Description
CLEARPROMPTS statement	Clears the value of the in-line prompt.
EOF(ARG.) function	Checks whether the command line argument pointer is past the last command line argument.
FILEINFO function	Returns information about the specified file's configuration.
ILPROMPT function	Evaluates strings containing in-line prompts.
GET(ARG.) statement	Retrieves a command line argument.
SEEK(ARG.) statement	Moves the command line argument pointer.

Appendix B: ASCII and hex equivalents

Decimal	Binary	Octal	Hexadecimal	ASCII
000	00000000	000	00	NUL
001	00000001	001	01	SOH
002	00000010	002	02	STX
003	00000011	003	03	ETX
004	00000100	004	04	EOT
005	00000101	005	05	ENQ
006	00000110	006	06	ACK
007	00000111	007	07	BEL
008	00001000	010	08	BS
009	00001001	011	09	HT
010	00001010	012	0A	LF
011	00001011	013	0B	VT
012	00001100	014	0C	FF
013	00001101	015	0D	CR
014	00001110	016	0E	SO
015	00001111	017	0F	SI
016	00010000	020	10	DLE
017	00010001	021	11	DC1
018	00010010	022	12	DC2
019	00010011	023	13	DC3
020	00010100	024	14	DC4
021	00010101	025	15	NAK
022	00010110	026	16	SYN
023	00010111	027	17	ETB
024	00011000	030	18	CAN
025	00011001	031	19	EM
026	00011010	032	1A	SUB
027	00011011	033	1B	ESC
028	00011100	034	1C	FS
029	00011101	035	1D	GS
030	00011110	036	1E	RS
031	00011111	037	1F	US
032	00100000	040	20	SPACE
033	00100001	041	21	!
034	00100010	042	22	"
035	00100011	043	23	#
036	00100100	044	24	$
037	00100101	045	25	%
038	00100110	046	26	&
039	00100111	047	27	'

Decimal	Binary	Octal	Hexadecimal	ASCII
040	00101000	050	28	(
041	00101001	051	29	)
042	00101010	052	2A	*
043	00101011	053	2B	+
044	00101100	054	2C	,
045	00101101	055	2D	–
046	00101110	056	2E	.
047	00101111	057	2F	/
048	00110000	060	30	0
049	00110001	061	31	1
050	00110010	062	32	2
051	00110011	063	33	3
052	00110100	064	34	4
053	00110101	065	35	5
054	00110110	066	36	6
055	00110111	067	37	7
056	00111000	070	38	8
057	00111001	071	39	9
058	00111010	072	3A	:
059	00111011	073	3B	;
060	00111100	074	3C	<
061	00111101	075	3D	=
062	00111110	076	3E	>
063	00111111	077	3F	?
064	01000000	100	40	@
065	01000001	101	41	A
066	01000010	102	42	B
067	01000011	103	43	C
068	01000100	104	44	D
069	01000101	105	45	E
070	01000110	106	46	F
071	01000111	107	47	G
072	01001000	110	48	H
073	01001001	111	49	I
074	01001010	112	4A	J
075	01001011	113	4B	K
076	01001100	114	4C	L
077	01001101	115	4D	M
078	01001110	116	4E	N
079	01001111	117	4F	O
080	01010000	120	50	P
081	01010001	121	51	Q
082	01010010	122	52	R

Decimal	Binary	Octal	Hexadecimal	ASCII
083	01010011	123	53	S
084	01010100	124	54	T
085	01010101	125	55	U
086	01010110	126	56	V
087	01010111	127	57	W
088	01011000	130	58	X
089	01011001	131	59	Y
090	01011010	132	5A	Z
091	01011011	133	5B	[
092	01011100	134	5C	\
093	01011101	135	5D	]
094	01011110	136	5E	^
095	01011111	137	5F	_
096	01100000	140	60	`
097	01100001	141	61	a
098	01100010	142	62	b
099	01100011	143	63	c
100	01100100	144	64	d
101	01100101	145	65	e
102	01100110	146	66	f
103	01100111	147	67	g
104	01101000	150	68	h
105	01110001	151	69	i
106	01110010	152	6A	j
107	01110011	153	6B	k
108	01110100	154	6C	l
109	01110101	155	6D	m
110	01110110	156	6E	n
111	01110111	157	6F	o
112	01111000	160	70	p
113	01111001	161	71	q
114	01111010	162	72	r
115	01111011	163	73	s
116	01111100	164	74	t
117	01110101	165	75	u
118	01110110	166	76	v
119	01110111	167	77	w
120	01111000	170	78	x
121	01111001	171	79	y
122	01111010	172	7A	z
123	01111011	173	7B	{
124	01111100	174	7C	|
125	01111101	175	7D	}

Decimal	Binary	Octal	Hexadecimal	ASCII
126	01111110	176	7E	~
127	01111111	177	7F	DEL
128	10000000	200	80	SQLNULL
251	11111011	373	FB	TM
252	11111100	374	FC	SM
253	11111101	375	FD	VM
254	11111110	376	FE	FM
255	11111111	377	FF	IM

The next table provides additional hexadecimal and decimal equivalents.

Hexadecimal	Decimal	Hexadecimal	Decimal
80	128	3000	12288
90	144	4000	16384
A0	160	5000	20480
B0	176	6000	24576
C0	192	7000	28672
D0	208	8000	32768
E0	224	9000	36864
F0	240	A000	40960
100	256	B000	45056
200	512	C000	49152
300	768	D000	53248
400	1024	E000	57344
500	1280	F000	61440

Appendix C: Correlative and conversion codes

This appendix describes the correlative and conversion codes used in dictionary entries and with the ICONV, ICONVS, OCONV, and OCONVS functions in BASIC. Use conversion codes with the ICONV function when converting data to internal storage format and with the OCONV function when converting data from its internal representation to an external output format.

Read this entire appendix, ICONV function, on page 205, and OCONV function, on page 272 before attempting to perform internal or external data conversion.

Note: If you try to convert the null value, null is returned and the STATUS function returns 1 (invalid data).

The NLS extended syntax is supported only for Release 9.4.1 and above.

The following table lists correlative and conversion codes.

Code	Description
A code: algebraic functions	Algebraic functions
BB and BX codes: bit conversion	Bit conversion (binary)
BB and BX codes: bit conversion	Bit conversion (hexadecimal)
C code: concatenation	Concatenation
D code: date conversion	Date conversion
DI code: international date conversion	International date conversion
ECS code: extended character set conversion	Extended character set conversion
F code: mathematical functions	Mathematical functions
F code: mathematical functions	Group extraction
L code: length function	Length function
MX, MO, MB, and MU0C codes: radix conversion	Binary conversion
MC Codes: masked character conversion	Masked alphabetic conversion
MC Codes: masked character conversion	Masked non-alphabetic conversion
MC Codes: masked character conversion	Decimal to hexadecimal conversion
MC Codes: masked character conversion	Decimal to hexadecimal conversion
MC Codes: masked character conversion	Masked lowercase conversion
MC Codes: masked character conversion	Masked multibyte conversion

Code	Description
MC Codes: masked character conversion	Masked single-byte conversion
MC Codes: masked character conversion	Masked numeric conversion
MC Codes: masked character conversion	Masked nonnumeric conversion
MC Codes: masked character conversion	Masked unprintable character conversion
MC Codes: masked character conversion	Masked initial capitals conversion
MC Codes: masked character conversion	Masked uppercase conversion
MC Codes: masked character conversion	Masked wide-character conversion
MC Codes: masked character conversion	Hexadecimal to decimal conversion
MC Codes: masked character conversion	Hexadecimal to decimal conversion
MC Codes: masked character conversion	Masked decimal conversion
ML	Masked left conversion
ML and MR codes: formatting numbers	NLS monetary conversion
MX, MO, MB, and MU0C codes: radix conversion	Octal conversion
MP code: packed decimal conversion	Packed decimal conversion
ML and MR codes: formatting numbers	Masked right conversion
MT code: time conversion	Time conversion
MX, MO, MB, and MU0C codes: radix conversion	Hexadecimal Unicode character conversion
MX, MO, MB, and MU0C codes: radix conversion	Hexadecimal conversion
MY code: ASCII conversion	ASCII conversion
NL code: Arabic numeral conversion	NLS Arabic numeral conversion
NLSmapname code: NLS map conversion	Conversion using NLS map name
NR code: roman numeral conversion	Roman numeral conversion
P code: pattern matching	Pattern matching
Q code: exponential notation	Exponential conversion
R code: range function	Range function

Code	Description
S (soundex) code	Soundex
S (substitution) code	Substitution
T code: text extraction	Text extraction
Tfile code: file translation *filename*	File translation
TI code: international time conversion	International time conversion

A code: algebraic functions

The A code converts A codes into F codes in order to perform mathematical operations on the field values of a record, or to manipulate strings. The A code functions in the same way as the F code but is easier to write and to understand.

Format

```
A [;] expression
```

expression can be one or more of the following:

A data location or string:

Expression	Description
loc[R]	Field number specifying a data value, followed by an optional R (repeat code).
N(*name*)	*name* is a dictionary entry for a field. The name is referenced in the file dictionary. An error message is returned if the field name is not found. Any codes specified in field 3 of *name* are applied to the field defined by *name*, and the converted value is processed by the A code.
string	Literal string enclosed in pairs of double quotation marks ("), single quotation marks ('), or backslashes (\ ).
number	Constant number enclosed in pairs of double quotation marks ("), single quotation marks ('), or backslashes (\ ). Any integer, positive, negative, or 0 can be specified.
D	System date (in internal format).
T	System time (in internal format).

A special system counter operand:

Expression	Description
@NI	Current system counter (number of items listed or selected).
@ND	Number of detail lines since the last BREAK on a break line.
@NV	Current value counter for columnar listing only.
@NS	Current subvalue counter for columnar listing only.
@NB	Current BREAK level number. 1 = lowest level break. This has a value of 255 on the grand-total line.
@LPV	**L**oad **P**revious **V**alue: load the result of the last correlative or conversion onto the stack.

A function:

Expression	Description
R(*exp*)	Remainder after integer division of the first operand by the second. For example, R(2,"5") returns the remainder when field 2's value is divided by 5.
S(*exp*)	Sum all multivalues in *exp*. For example, S(6) sums the multivalues of field 6.
IN(*exp*)	Test for the null value.
[]	Extract substring. Field numbers, literal numbers, or expressions can be used as arguments within the brackets. For example, if the value of field 3 is 9, then 7["2",3] returns the second through ninth characters of field 7. The brackets are part of the syntax and must be typed.
IF(*expression*) | THEN(*expression*) | ELSE(*expression*)	A conditional expression.
(*conv*)	Conversion expression in parentheses (except A and F conversions).

An arithmetic operator:

Expression	Description
*	Multiply operands.
/	Divide operands. Division always returns an integer result: for example, "3" / "2" evaluates to 1, not to 1.5.
+	Add operands.
-	Subtract operands.
:	Concatenate operands.

A relational operator:

Expression	Description
=	Equal to
<	Less than
>	Greater than
# or <>	Not equal to
<=	Less than or equal to
>=	Greater than or equal to

A conditional operator:

Expression	Description
AND	Logical AND
OR	Logical OR

In most cases F and A codes do not act on a data string passed to them. The code specification itself contains all the necessary data (or at least the names of fields that contain the necessary data). So the following A codes produce identical F codes, which in turn assign identical results to X:

```
X = OCONV("123", "A;'1'    + '2'" )
 X = OCONV("", "A;'1'  + '2'" )
 X = OCONV(@ID, "A;'1' + '2'" )
 X = OCONV("The quick brown fox jumped over a lazy dog's
    back","A;'1'    + '2'" )
```

The data strings passed to the A code—123, the empty string, the record ID, and “The quick brown fox…” string—simply do not come into play. The only possible exception occurs when the user includes the LPV (load previous value) special operand in the A or F code. The following example adds the value 5 and the previous value 123 to return the sum 128:

```
X = OCONV("123", "A;'5' + LPV" )
```

It is almost never right to call an A or F code using the vector conversion functions OCONVS and ICONVS. In the following example, Y = 123V456V789:

```
X = OCONVS(Y, "A;'5'+ '2' )
```

The statement says, “For each value of Y, call the A code to add 5 and 2.” (V represents a value mark.) The A code gets called three times, and each time it returns the value 7. X, predictably, gets assigned 7. The scalar OCONV function returns the same result in much less time.

What about correlatives and conversions within an A or F code? Since any string in the A or F code can be multivalued, the F code calls the vector functions OCONVS or ICONVS any time it encounters a secondary correlative or conversion. In the following example, the F code—itself called only once—calls OCONVS to ensure that the G code gets performed on each value of @RECORD< 1 >. X is assigned the result *cccVfff*:

```
@RECORD< 1 > = aaa*bbb*cccVddd*eee*fff
 X = OCONV("", "A;1(G2*1)"
```

The value mark is reserved to separate individual code specifications where multiple successive conversions must be performed.

The following dictionary entry specifies that the substring between the first and second asterisks of the record ID should be extracted, then the first four characters of that substring should be extracted, then the masked decimal conversion should be applied to that substring:

```
001: D
 002: 0
 003: G1*1VT1,4VMD2
 004: Foo
 005: 6R
 006: S
```

To attempt to define a multivalued string as part of the A or F code itself rather than as part of the @RECORD produces invalid code. For instance, both:

```
X = OCONV("", "A;'aaa*bbb*cccVddd*eee*fff'(G2*1)" )
```

and the dictionary entry:

```
001: D
 002: 0
 003: A;'aaa*bbb*cccVddd*eee*fff'(G2*1)
 004: Bar
 005: 7L
 006: S
```

are invalid. The first returns an empty string (the original value) and a status of 2. The second returns the record ID; if the STATUS function were accessible from dictionary entries, it would also be set to 2.

BB and BX codes: bit conversion

The BB and BX codes convert data from external binary and hexadecimal format to internal bit string format and vice versa.

Formats

`BB` Binary conversion (base 2)

`BX` Hexadecimal conversion (base 16)

Characters outside of the range for each of the bases produce conversion errors. The ranges are as follows:

Conversion	Range
BB (binary)	0, 1
BX (hexadecimal)	0 through 9, A through F, a through f

With ICONV

When used with the `ICONV` function, BB converts a binary data value to an internally stored bit string. The external binary value must be in the following format:

`B ' bit [bit] … '`

bit is either 1 or 0.

BX converts a hexadecimal data value to an internally stored bit string. The external hexadecimal value must be in the following format:

`X ' hexit [hexit] … '`

hexit is a number from 0 through 9, or a letter from A through F, or a letter from a through f.

With OCONV

When used with the `OCONV` function, BB and BX convert internally stored bit strings to their equivalent binary or hexadecimal output formats, respectively. If the stored data is not a bit string, a conversion error occurs.

C code: concatenation

The C code chains together field values or quoted strings, or both.

Format

`C [;] expression1cexpression2 [cexpression3] …`

The semicolon is optional and is ignored.

c is the character to be inserted between the fields. Any nonnumeric character (except system delimiters) is valid, including a blank. A semicolon (;) is a reserved character that means no separation character is to be used. Two separators cannot follow in succession, with the exceptions of semicolons and blanks.

expression is a field number and requests the contents of that field; or any string enclosed in single quotation marks ('), double quotation marks ("), or backslashes (\ ); or an asterisk (*), which specifies the data value being converted.

You can include any number of delimiters or expressions in a C code.

Note: When the C conversion is used in a field descriptor in a file dictionary, the field number in the LOC or A/AMC field of the descriptor should be 0. If it is any other number and the specified field contains an empty string, the concatenation is not performed.

Examples

Assume a BASIC program with @RECORD = "oneFtwoFthreeVfour":

Statement	Output
PRINT OCONV("x","C;1;'xyz';2")	onexyztwo
PRINT ICONV("x","C;2;'xyz';3")	twoxyzthreeVfour
PRINT OCONV("","C;2;'xyz';3")	
PRINT ICONV(x,"C;1***2")	one*x*two
PRINT OCONV(0,"C;1:2+3")	one:two+threeVfour

There is one anomaly of the C code (as implemented by ADDS Mentor, at least) that the UniVerse C code does not reproduce:

PRINT ICONV (x, "C*1*2*3")	x1x2x3

The assumption that anything following a nonseparator asterisk is a separator seems egregious, so the UniVerse C code implements:

PRINT ICONV (x, "C*1*2*3")	xone*two*threeVfour

Anyone wanting the ADDS effect can quote the numbers.

D code: date conversion

The D code converts input dates from conventional formats to an internal format for storage. It also converts internal dates back to conventional formats for output. When converting an input date to internal format, date conversion specifies the format you use to enter the date. When converting internal dates to external format, date conversion defines the external format for the date.

Format

`D [n] [*m] [ s] [fmt [[f1, f2, f3, f4, f5]]] [E] [L]`

If the D code does not specify a year, the current year is assumed. If the code specifies the year in two-digit form, the years from 0 through 29 mean 2000 through 2029, and the years from 30 through 99 mean 1930 through 1999.

You can set the default date format with the `DATE.FORMAT` command. A system-wide default date format can be set in the `msg.text` file of the UV account directory. Date conversions specified in file dictionaries or in the ICONV function or the OCONV function use the default date format except where they specifically override it. When NLS locales are enabled, the locale overrides any value set in the `msg.text` file.

Format	Description
n	Single digit (normally 1 through 4) that specifies the number of digits of the year to output. The default is 4.

Format	Description
*	Any single nonnumeric character that separates the fields in the case where the conversion must first do a group extraction to obtain the internal date. * cannot be a system delimiter.
m	Single digit that must accompany any use of an asterisk. It denotes the number of asterisk-delimited fields to skip in order to extract the date.
s	Any single nonnumeric character to separate the day, month, and year on output. *s* cannot be a system delimiter. If you do not specify *s*, the date is converted in 09 SEP 1996 form, unless a format option overrides it. If NLS locales are enabled and you do not specify a separator character or *n*, the default date form is 09 SEP 1996. If the Time category is active, the conversion code in the D_FMT field is used. If NLS locales are enabled and you do not specify an *s* or format option, the order and the separator for the day/month/year defaults to the format defined in the DI_FMT or in the D_FMT field. If the day/month/year order cannot be determined from these fields, the conversion uses the order defined in the DEFAULT_DMY_ORDER field. If you do not specify s and the month is numeric, the separator character comes from the DEFAULT_DMY_SEP field.

Format	Description	
fmt	Specifies up to five of the following special format options that let you request the day, day name, month, year, and era name:	
	Y [*n*]	Requests only the year number (*n* digits).
	YA	Requests only the name of the Chinese calendar year. If NLS locales are enabled, uses the YEARS field in the NLS.LC.TIME file.
	M	Requests only the month number (1 through 12).
	MA	Requests only the month name. If NLS locales are enabled, uses the MONS field in the NLS.LC.TIME file. You can use any combination of upper- and lowercase letters for the month; UniVerse checks the combination against the ABMONS field, otherwise it checks the MONS field.
	MB	Requests only the abbreviated month name. If NLS locales are enabled, uses the ABMONS field in the NLS.LC.TIME file; otherwise, uses the first three characters of the month name.
	MR	Requests only the month number in Roman numerals (I through XII).
	D	Requests only the day number within the month (1 through 31).
	W	Requests only the day number within the week (1 through 7, where Sunday is 7). If NLS locales are enabled, uses the DAYS field in the NLS.LC.TIME file, where Sunday is 1.
	WA	Requests only the day name. If NLS locales are enabled, uses the DAYS field in the NLS.LC.TIME file, unless modified by the format modifiers, *f1, f2,* and so forth.
	WB	Requests only the abbreviated day name. If NLS locales are enabled, uses the ABDAYS field in the NLS.LC.TIME file.
	Q	Requests only the quarter number within the year (1 through 4).
	J	Requests only the day number within the year (1 through 366).
	N	Requests only the year within the current era. If NLS is not enabled, this is the same as the year number returned by the Y format option. If NLS locales are enabled, N uses the ERA STARTS field in the NLS.LC.TIME file.
	NA	Requests only the era name corresponding to the current year. If NLS locales are enabled, uses the ERA NAMES or ERA STARTS fields in the NLS.LC.TIME file.
	Z	Requests only the time-zone name, using the name from the operating system.

Format	Description	
[*f1, f2, f3, f4, f5* **]**	*f1*, *f2*, *f3, f4,* and *f5* are the format modifiers for the format options. The brackets are part of the syntax and must be typed. You can specify up to five modifiers, which correspond to the options in *fmt,* respectively. The format modifiers are positional parameters: if you want to specify *f3* only, you must include two commas as placeholders. Each format modifier must correspond to a format option. The value of the format modifiers can be any of the following:	
	n	Specifies how many characters to display. *n* can modify any format option, depending on whether the option is numeric or text. ▪ If numeric, (D, M, W, Q, J, Y, 0), *n* prints *n* digits, right-justified with zeros. ▪ If text (MA, MB, WA, WB, YA, N, '*text*'), *n* left-justifies the option within *n* spaces.
	A[*n*]	Month format is alphabetic. *n* is a number from 1 through 32 specifying how many characters to display. Use A with the Y, M, W, and N format options.
	Z[*n*]	Suppresses leading zeros in day, month, or year. *n* is a number from 1 through 32 specifying how many digits to display. Z works like *n*, but zero-suppresses the output for numeric options.
	'*text*'	Any text enclosed in single or double quotation marks is treated as if there were no quotation marks and placed after the text produced by the format option in the equivalent position. Any separator character is ignored. *'text'* can modify any option.
E	Toggles the European (day/month/year) versus the U.S. (month/day/year) formatting of dates. Since the NLS.LC.TIME file specifies the default day/month/year order, E is ignored if you use a Time convention.	
L	Specifies that lowercase letters should be retained in month or day names; otherwise the routine converts names to all capitals. Since the NLS.LC.TIME file specifies the capitalization of names, L is ignored if you use a Time convention.	

The following table shows the format options you can use together:

Format option	Use with These options
Y	M, MA, D, J, [*f1, f2*, *f3*, *f4*, *f5*]
YA	M, MA, D, [*f1, f2*, *f3*, *f4*, *f5*]
M	Y, YA, D, [*f1, f2*, *f3*, *f4*, *f5*]
MA	Y, YA, D, [*f1, f2*, *f3*, *f4*, *f5*]
MB	Y, YA, D, [*f1, f2*, *f3*, *f4*, *f5*]
D	Y, M, [*f1, f2*, *f3*, *f4*, *f5*]
N	Y, M, MA, MB, D, WA [*f1, f2*, *f3*, *f4*, *f5*]
NA	Y, M, MA, MB, D, WA [*f1, f2*, *f3*, *f4*, *f5*]
W	Y, YA, M, MA, D
WA	Y, YA, M, MA, D
WB	Y, YA, M, MA, D
Q	[*f1*]
J	Y, [*f1, f2*, *f3*, *f4*, *f5*]
Z	[*f1*]

Each format modifier must correspond to a format option. The following table shows which modifiers can modify which options:

Format	Format Option				
Modifiers	**D**	**M**	**Y**	**J**	**W**
A	no	yes	yes	no	yes
n	yes	yes	yes	yes	yes
Z	yes	yes	yes	yes	no
'text'	yes	yes	yes	yes	yes

ICONV and OCONV differences

The syntax for converting dates with the `ICONV` function is the same as for the `OCONV` function, except that:

Parameter	Difference
n	Ignored. The input conversion accepts any number of year's digits regardless of the *n* specification. If no year exists in the input date, the routine uses the year part of the system date.
s	Ignored. The input conversion accepts any single nonnumeric, nonsystem-delimiter character separating the day, month, and year regardless of the *s* specification. If the date is input as an undelimited string of characters, it is interpreted as one of the following formats: [YY]YYMMDD or [YY]YYDDD.
subcodes	Ignored. The input conversion accepts any combination of upper- and lowercase letters in the month part of the date.

In IDEAL and INFORMATION flavor accounts, the input conversion of an improper date returns a valid internal date and a STATUS function value of 3. For example, 02/29/93 is interpreted as 03/01/93, and 09/31/93 is interpreted as 10/01/93. A status of 3 usually represents a common human error. More flagrant errors return an empty string and a STATUS() value of 1.

In PICK, REALITY, and IN2 flavor accounts, the input conversion of an improper date always returns an empty string and a status of 1.

If the data to be converted is the null value, a STATUS() value of 3 is set and no conversion occurs.

Example

The following example shows how to use the format modifiers:

```
D DMY[Z,A3,Z2]
```

Z modifies the day format option (D) by suppressing leading zeros (05 becomes 5). A3 modifies the month format option (M) so that the month is represented by the first three alphabetic characters (APRIL becomes APR). Z2 modifies the year format option (Y) by suppressing leading zeros and displaying two digits. This conversion converts April 5, 1993 to 5 APR 93.

DI code: international date conversion

The international date conversion lets you convert dates in internal format to the default local convention format and vice versa. If NLS locales are not enabled, the DI conversion defaults to D. If NLS locales are enabled, DI uses the date conversion in the DI_FMT field. The DI_FMT field can contain any valid D code.

Format

```
DI
```

ECS code: extended character set conversion

The ECS code resolves clashes between the UniVerse system delimiters and the ASCII characters CHAR(251) through CHAR(255). It converts the system delimiters and ASCII characters to alternative characters using an appropriate localization procedure. If no localization library is in use, the input string is returned without character conversion.

This code is used with an ICONV function or an OCONV function.

Format

```
ECS
```

F code: mathematical functions

The F code performs mathematical operations on the data values of a record, or manipulates strings. It comprises any number of operands or operators in reverse Polish format (Lukasiewicz) separated by semicolons.

The program parses the F code from left to right, building a stack of operands. Whenever it encounters an operator, it performs the requested operation, puts the result on the top of the stack, and pops the lower stack elements as necessary.

Format

```
F [;] element [; element …]
```

The semicolon (;) is the element separator.

element can be one or more of the items from the following categories:

A data location or string:

Element	Description
loc[R]	Numeric location specifying a data value to be pushed onto the stack, optionally followed by an R (repeat code).
C*n*	*n* is a constant to be pushed onto the stack.
string	Literal string enclosed in pairs of double quotation marks ("), single quotation marks ('), or backslashes (\ ).
number	Constant number enclosed in pairs of double quotation marks ("), single quotation marks ('), or backslashes (\ ). Any integer, positive, negative, or 0 can be specified.
D	System date (in internal format).
T	System time (in internal format).

A special system counter operand

Element	Description
@NI	Current item counter (number of items listed or selected).
@ND	Number of detail lines since the last BREAK on a break line.

Element	Description
@NV	Current value counter for columnar listing only.
@NS	Current subvalue counter for columnar listing only.
@NB	Current BREAK level number. 1 = lowest level break. This has a value of 255 on the grand-total line.
@LPV	**L**oad **P**revious **V**alue: load the result of the last correlative code onto the stack.

An operator:

Operators specify an operation to be performed on top stack entries. *stack1* refers to the value on the top of the stack, *stack2* refers to the value just below it, *stack3* refers to the value below *stack2*, and so on.

Element	Description
*[*n*]	Multiply *stack1* by *stack2*. The optional *n* is the descaling factor (that is, the result is divided by 10 raised to the *n*th power).
/	Divide *stack1* into *stack2*, result to *stack1*.
R	Same as /, but instead of the quotient, the remainder is returned to the top of the stack.
+	Add *stack1* to *stack2*.
-	Subtract *stack1* from *stack2*, result to *stack1* (except for REALITY flavor, which subtracts *stack2* from *stack1*).
:	Concatenate *stack1* string onto the end of *stack2* string.
[]	Extract substring. *stack3* string is extracted, starting at the character specified by *stack2* and continuing for the number of characters specified in *stack1*. This is equivalent to the BASIC [*m*,*n*] operator, where *m* is in *stack2* and *n* is in *stack1*.
S	Sum of multivalues in *stack1* is placed at the top of the stack.
_	Exchange *stack1* and *stack2* values.
P or \	Push *stack1* back onto the stack (that is, duplicate *stack1*).
^	Pop the *stack1* value off the stack.
(*conv*)	Standard conversion operator converts data in *stack1*, putting the result into *stack1*.

A logical operator:

Logical operators compare *stack1* to *stack2*. Each returns 1 for true and 0 for false:

Element	Description
=	Equal to.
<	Less than.
>	Greater than.
# or <>	Not equal to.
[	Less than or equal to.
]	Greater than or equal to.
&	Logical AND.
!	Logical OR.
n\	Defines a label by a positive integer enclosed by two backslashes (\\).
#*n*	Connection to label *n* if *stack1* differs from *stack2*.

Element	Description
>*n*	Connection to label *n* if *stack1* is greater than *stack2*.
<*n*	Connection to label *n* if *stack1* is less than *stack2*.
=*n*	Connection to label *n* if *stack1* equals *stack2*.
}*n*	Connection to label *n* if *stack1* is greater than or equal to *stack2*.
{*n*	Connection to label *n* if *stack1* is less than or equal to *stack2*.
IN	Tests *stack1* to see if it is the null value.
F*nnnn*	If *stack1* evaluates to false, branch forward *nnnn* characters in the F code, and continue processing.
B*nnnn*	Branch forward unconditionally *nnnn* characters in the F code, and continue processing.
G*nnnn*	Go to label *nnnn*. The label must be a string delimited by backslashes (\ ).
G*	Go to the label defined in *stack1*. The label must be a string delimited by backslashes (\ ).

Note: The F code performs only integer arithmetic.

G code: group extraction

The G code extracts one or more values, separated by the specified delimiter, from a field.

Format

```
G [skip] delim #fields
```

skip specifies the number of fields to skip; if it is not specified, 0 is assumed and no fields are skipped.

delim is any single nonnumeric character (except IM, FM, VM, SM, and TM) used as the field separator.

#fields is the decimal number of contiguous delimited values to extract.

L code: length function

The L code places length constraints on the data to be returned.

Format

```
L [n [,m]]
```

If L*n* is specified, selection is met if the value's length is less than or equal to *n* characters; otherwise an empty string is returned.

If L*n*,*m* is specified, selection is met if the value's length is greater than or equal to *n* characters, and less than or equal to *m* characters; otherwise an empty string is returned.

If *n* is omitted or 0, the length of the value is returned.

MC Codes: masked character conversion

The MC codes let you change a field's data to upper- or lowercase, to extract certain classes of characters, to capitalize words in the field, and to change unprintable characters to periods.

Formats

The following table describes the available MC formats.

Code	Description
MCA	Extracts all alphabetic characters in the field, both upper- and lowercase. Non-alphabetic characters are not printed. In NLS mode, uses the ALPHABETICS field in the NLS.LC.CTYPE file.
MC/A	Extracts all non-alphabetic characters in the field. Alphabetic characters are not printed. In NLS mode, uses the NON-ALPHABETICS field in the NLS.LC.CTYPE file.
MCD[X]	Converts decimal to hexadecimal equivalents.
MCL	Converts all uppercase letters to lowercase. Does not affect lowercase letters or non-alphabetic characters. In NLS mode, uses the UPPERCASE and DOWNCASED fields in the NLS.LC.CTYPE file.
MCM	Use only if NLS is enabled. Extracts all NLS multibyte characters in the field. Multibyte characters are all those outside the Unicode range (x0000–x007F), the UniVerse system delimiters, and the null value. As long as NLS is enabled, the conversion still works if locales are off. If NLS mode is disabled, the code returns a STATUS function of 2, that is, an invalid conversion code.
MC/M	Use only if NLS is enabled. Extracts all NLS single-byte characters in the field. Single-byte characters are all those in the Unicode range x0000–x007F. As long as NLS is enabled, the conversion still works if locales are off. If NLS mode is disabled, the code returns a STATUS of 2, that is, an invalid conversion code.
MCN	Extracts all numeric characters in the field. Alphabetic characters are not printed. In NLS mode, uses the NUMERICS field in the NLS.LC.CTYPE file.
MC/N	Extracts all nonnumeric characters in the field. Numeric characters are not printed. In NLS mode, uses the NON-NUMERICS field in the NLS.LC.CTYPE file.
MCP	Converts each unprintable character to a period. In NLS mode, uses the PRINTABLE and NON_PRINTABLE fields in the NLS.LC.CTYPE file.
MCT	Capitalizes the first letter of each word in the field (the remainder of the word is converted to lowercase). In NLS mode, uses the LOWERCASE and UPCASED fields of the NLS.LC.CTYPE file. If you set up an NLS Ctype locale category, and you define a character to be trimmable, if this character appears in the middle of a string, it is not lowercased nor are the rest of the characters up to the next separator character. This is because the trimmable character is considered a separator (like <space>).
MCU	Converts all lowercase letters to uppercase. Does not affect uppercase letters or non-alphabetic characters. In NLS mode, uses the LOWERCASE and UPCASED fields in the NLS.LC.CTYPE file.
MCW	Use only if NLS is enabled. Converts between 7-bit standard ASCII (0021-007E range) and their corresponding double-byte characters, which are two display positions in width (FF01-FF5E full-width range). As long as NLS is enabled, the conversion still works if locales are off. If NLS mode is disabled, the code returns a STATUS of 2, that is, an invalid conversion code.
MCX[D]	Converts hexadecimal to decimal equivalents.

If you set up an NLS Ctype locale category, and you define a character to be trimmable, if this character appears in the middle of a string, it is not lowercased nor are the rest of the characters up to the next separator character. This is because the trimmable character is considered a separator (like <space>).

MD code: masked decimal conversion

The MD code converts numeric input data to a format appropriate for internal storage. If the code includes the $, F, I, or Y option, the conversion is monetary, otherwise it is numeric.

The MD code must appear in either an ICONV function or an OCONV function expression. When converting internal representation of data to external output format, masked decimal conversion inserts the decimal point and other appropriate formats into the data.

Note: If NLS is enabled and either the Numeric or Monetary categories are set to OFF, the MD code behaves as if NLS locales were turned off.

Format

```
MD [n [m]] [,] [$] [F ] [I] [Y] [intl] [- | < | C | D] [P] [Z] [T] [fx
]
```

If the value of *n* is 0, the decimal point does not appear in the output.

The optional *m* specifies the power of 10 used to scale the input or output data. On input, the decimal point is moved *m* places to the right before storing. On output, the decimal point is moved *m* places to the left. For example, if *m* is 2 in an input conversion and the input data is 123, it would be stored as 12300. If *m* is 2 in an output conversion and the stored data is 123, it would be output as 1.23. If *m* is not specified, it is assumed to be the same as *n*. In both cases, the last required decimal place is rounded off before excess digits are truncated. Zeros are added if not enough decimal places exist in the original expression.

If NLS is enabled and the conversion is monetary, the thousands separator comes from the THOU_SEP field of the Monetary category of the current locale, and the decimal separator comes from the DEC_SEP field. If the conversion is numeric, the thousands separator comes from the THOU_SEP field of the Numeric category, and the decimal separator comes from the DEC_SEP field.

Code	Description
,	Specifies that thousands separators be inserted every three digits to the left of the decimal point on output.
$	Prefixes a local currency sign to the number before justification. If NLS is enabled, the CURR_SYMBOL of the Monetary category is used.
F	Prefixes a franc sign (F) to the number before justification. (In all flavors except IN2, you must specify F in the conversion code if you want ICONV to accept the character F as a franc sign.)
I	Used with the OCONV function, the international monetary symbol for the locale is used (INTL_CURR_SYMBOL in the Monetary category). Used with the ICONV function, the international monetary symbol for the locale is removed. If NLS is disabled or the Monetary category is turned off, the default symbol is USD.
Y	Used with the OCONV function: if NLS is enabled, the yen/yuan character (Unicode 00A5) is used. If NLS is disabled or the Monetary locale category is turned off, the ASCII character xA5 is used.

Code	Description	
intl	An expression that customizes numeric output according to different international conventions, allowing multibyte characters. The *intl* expression can specify a prefix, a suffix, and the characters to use as a thousands delimiter and as the decimal delimiter, using the locale definition from the NLS.LC.NUMERIC file. The *intl* expression has the following syntax: [*prefix* , *thousands* , *decimal* , *suffix*] The brackets are part of the syntax and must be typed. The four elements are positional parameters and must be separated by commas. Each element is optional, but its position must be held by a comma. For example, to specify a suffix only, type [,,,*suffix*].	
	prefix	Character string to prefix to the number. If *prefix* contains spaces, commas, or right square brackets, enclose it in quotation marks.
	thousands	Character string that separates thousands. If *thousands* contains spaces, commas, or right square brackets, enclose it in quotation marks.
	decimal	Character string to use as a decimal delimiter. If *decimal* contains spaces, commas, or right square brackets, enclose it in quotation marks.
	suffix	Character string to append to the number. If *suffix* contains spaces, commas, or right square brackets, enclose it in quotation marks.
-	Specifies that negative data be suffixed with a minus sign and positive data be suffixed with a blank space.	
<	Specifies that negative data be enclosed in angle brackets for output; positive data is prefixed and suffixed with a blank space.	
C	Specifies that negative data include a suffixed CR; positive data is suffixed with two blank spaces.	
D	Specifies that negative data include a suffixed DB; positive data is suffixed with two blank spaces.	
P	Specifies that no scaling be performed if the input data already contains a decimal point.	
Z	Specifies that 0 be output as an empty string.	
T	Specifies that the data be truncated without rounding. Used with the ICONV function: if NLS is enabled, the yen/yuan character is removed. If NLS is disabled or the Monetary category is turned off, the ASCII character xA5 is removed.	

When NLS locales are enabled, the <, –, C and D options define numbers intended for monetary use. These options override any specified monetary formatting. If the conversion is monetary and no monetary formatting is specified, it uses the POS_FMT, NEG_FMT, POS_SIGN, and NEG_SIGN fields from the Monetary category of the current locale. If the conversion is numeric and the ZERO_SUP field is set to 1, leading zeros of numbers between –1 and 1 are suppressed. For example, –0.5 is output as –.5 .

When converting data to internal format, the *fx* option has the following effect. If the input data has been overlaid on a background field of characters (for example, $###987.65), the *fx* option is used with `ICONV` to indicate that the background characters should be ignored during conversion. The *f* is a one- or two-digit number indicating the maximum number of background characters to be ignored. The *x* specifies the background character to be ignored. If background characters exist in the input data and you do not use the *fx* option, the data is considered bad and an empty string results.

When converting data from internal representation to external output format, the *fx* option causes the external output of the data to overlay a field of background characters. The *f* is a one- or two-digit number indicating the number of times the background character is to be repeated. The *x* specifies the character to be used as a background character. If the $ option is used with the *fx* option, the $ precedes the background characters when the data is output.

MM code: monetary conversion

The MM code provides for local conventions for monetary formatting.

Format

```
MM [n] [I [L] ]
```

Note: If NLS is enabled and either the Numeric or Monetary categories are set to OFF, the MM code behaves as if locales were turned off.

If NLS is enabled and the Monetary category is turned on, the MM code uses the local monetary conventions for decimal and thousands separators. The format options are as follows:

Option	Description
n	Specifies the number of decimal places (0 through 9) to be maintained or output. If *n* is omitted, the DEC_PLACES field from the Monetary category is used; if the I option is also specified, the INTL_DEC_PLACES field is used. If NLS is disabled or the Monetary category is turned off, and *n* is omitted, *n* defaults to 2.
I	Substitutes the INTL_CURR_SYMBOL for the CURR_SYMBOL in the Monetary category of the current locale. If NLS locales are off, the default international currency symbol is USD.
L	Used with the I option to specify that decimal and thousands separators are required instead of the UniVerse defaults (. and ,). The DEC_SEP and THOU_SEP fields from the Monetary category are used.

If you specify MM with no arguments, the decimal and thousands separators come from the Monetary category of the current locale, and the currency symbol comes from the CURR_SYMBOL field. If you specify MM with the I option, the decimal and thousands separators are . (period) and , (comma), and the currency symbol comes from the INTL_CURR_SYMBOL field. If you specify MM with both the I and the L options, the decimal and thousands separators come from the Monetary category of the current locale, and the currency symbol comes from the INTL_CURR_SYMBOL field. The I and L options are ignored when used in the `ICONV` function.

If NLS is disabled or the category is turned off, the default decimal and thousands separators are the period and the comma.

The STATUS values are as follows:

Value	Description
0	Successful conversion. Returns a string containing the converted monetary value.
1	Unsuccessful conversion. Returns an empty string.
2	Invalid conversion code. Returns an empty string.

ML and MR codes: formatting numbers

The ML and MR codes allow special processing and formatting of numbers and monetary amounts. If the code includes the F or I option, the conversion is monetary, otherwise it is numeric. ML specifies left justification; MR specifies right justification.

Format

```
ML [n [m]] [Z] [,] [C | D | M | E | N] [$] [F] [intl] [(fx )]
MR [ n [m]] [Z] [,] [C | D | M | E | N] [$] [F] [ intl] [(fx )]
```

Note: If NLS is enabled and either the Numeric or Monetary categories are set to OFF, the ML and MR codes behave as if locales were turned off.

Parameter	Description
n	Number of digits to be printed to the right of the decimal point. If *n* is omitted or 0, no decimal point is printed.
m	Descales (divides) the number by 10 to the *m*th power. If not specified, *m* = *n* is assumed. On input, the decimal point is moved *m* places to the right before storing. On output, the decimal point is moved *m* places to the left. For example, if *m* is 2 in an input conversion specification and the input data is 123, it would be stored as 12300. If *m* is 2 in an output conversion specification and the stored data is 123, it would be output as 1.23. If the *m* is not specified, it is assumed to be the same as the *n* value. In both cases, the last required decimal place is rounded off before excess digits are truncated. Zeros are added if not enough decimal places exist in the original expression.

If NLS is enabled and the conversion is monetary, the thousands separator comes from the THOU_SEP field of the Monetary category of the current locale, and the decimal separator comes from the DEC_SEP field. If the conversion is numeric, the thousands separator comes from the THOU_SEP field of the Numeric category, and the decimal separator comes from the DEC_SEP field.

When NLS locales are enabled, the <, –, C, and D options define numbers intended for monetary use. These options override any specified monetary formatting. If the conversion is monetary and no monetary formatting is specified, it uses the POS_FMT, NEG_FMT, POS_SIGN, and NEG_SIGN fields from the Monetary category of the current locale.

They are unaffected by the Numeric or Monetary categories. If no options are set, the value is returned unchanged.

Option	Description
Z	Specifies that 0 be output as an empty string.
,	Specifies that thousands separators be inserted every three digits to the left of the decimal point on output.
C	Suffixes negative values with CR.
D	Suffixes positive values with DB.
M	Suffixes negative numbers with a minus sign (-).
E	Encloses negative numbers in angle brackets (< >).
N	Suppresses the minus sign (-) on negative numbers.
$	Prefixes a local currency sign to the number before justification. The $ option automatically justifies the number and places the currency sign just before the first digit of the number output.

Option	Description	
F	Prefixes a franc sign (F) to the number before justification. (In all flavors except IN2, you must specify F in the conversion code if you want `ICONV` to accept the character F as a franc sign.)	
intl	An expression that customizes output according to different international conventions, allowing multibyte characters. The *intl* expression can specify a prefix, a suffix, and the characters to use as a thousands delimiter and as the decimal delimiter. The *intl* expression has the following syntax: [*prefix* , *thousands* , *decimal* , *suffix*] The brackets are part of the syntax and must be typed. The four elements are positional parameters and must be separated by commas. Each element is optional, but its position must be held by a comma. For example, to specify a suffix only, type [,,,*suffix*].	
	prefix	Character string to prefix to the number. If *prefix* contains spaces, commas, or square brackets, enclose it in quotation marks.
	thousands	Character string that separates thousands. If *thousands* contains spaces, commas, or square brackets, enclose it in quotation marks.
	decimal	Character string to use as a decimal delimiter. If *decimal* contains spaces, commas, or right square brackets, enclose it in quotation marks.
	suffix	Character string to append to the number. If *suffix* contains spaces, commas, or right square brackets, enclose it in quotation marks.
f	One of three format codes:	
	#	Data justifies in a field of *x* blanks.
	*	Data justifies in a field of *x* asterisks (*).
	%	Data justifies in a field of *x* zeros.

The format codes precede *x*, the number that specifies the size of the field.

You can also enclose literal strings in the parentheses. The text is printed as specified, with the number being processed right- or left-justified.

NLS mode uses the definitions from the Numeric category, unless the conversion code indicates a definition from the Monetary category. If you disable NLS or turn off the required category, the existing definitions apply.

MP code: packed decimal conversion

The MP code allows decimal numbers to be packed two-to-the-byte for storage. Packed decimal numbers occupy approximately half the disk storage space required by unpacked decimal numbers.

Format

`MP`

Leading + signs are ignored. Leading - signs cause a hexadecimal D to be stored in the lower half of the last internal digit. If there is an odd number of packed halves, four leading bits of 0 are added. The range of the data bytes in internal format expressed in hexadecimal is 00 through 99 and 0D through 9D. Only valid decimal digits (0-9) and signs (+, -) should be input. Other characters cause no conversion to take place.

Packed decimal numbers should always be unpacked for output, since packed values that are output unconverted are not displayed on terminals in a recognizable format.

MT code: time conversion

The MT code converts times from conventional formats to an internal format for storage. It also converts internal times back to conventional formats for output. When converting input data to internal storage format, time conversion specifies the format that is to be used to enter the time. When converting internal representation of data to external output format, time conversion defines the external output format for the time.

Format

`MT [H] [P] [Z] [S] [c] [[f1, f2, f3]]`

MT is required when you specify time in either the ICONV function or the OCONV function. The remaining specifiers are meaningful only in the `OCONV` function; they are ignored when used in the `ICONV` function.

The internal representation of time is the numeric value of the number of seconds since midnight.

If used with `ICONV` in an IDEAL, INFORMATION, or PIOPEN flavor account, the value of midnight is 0. In all other account flavors, the value of midnight is 86400.

To separate hours, minutes, and seconds, you can use any nonnumeric character that is not a system delimiter. Enclose the separator in quotation marks. If no minutes or seconds are entered, they are assumed to be 0. You can use a suffix of AM, A, PM, or P to specify that the time is before or after noon. If an hour larger than 12 is entered, a 24-hour clock is assumed. 12:00 AM is midnight and 12:00 PM is noon.

If NLS is enabled and the Time category is active, the locale specifies the AM and PM strings, and the separator comes from the T_FMT or TI_FMT fields in the Time category.

Parameter	Description	
H	Specifies to use a 12-hour format with the suffixes AM or PM. The 24-hour format is the default. If NLS is enabled, the AM and PM strings come from the AM_STR and PM_STR fields in the Time category.	
P	Same as H, but the AM and PM strings are prefixed, not suffixed.	
Z	Specifies to zero-suppress hours in the output.	
S	Specifies to use seconds in the output. The default omits seconds.	
c	Specifies the character used to separate the hours, minutes, and seconds in the output. The colon (:) is the default. If NLS is enabled and you do not specify *c*, and if the Time category is active, *c* uses the DEFAULT_TIME_SEP field.	
[*f1*, *f2*, *f3*]	Specify format modifiers. You must include the brackets, as they are part of the syntax. You can specify from 1 through 3 modifiers, which correspond to the hours, minutes, and seconds, in that order. The format modifiers are positional parameters: if you want to specify *f3* only, you must include two commas as placeholders. Each format modifier must correspond to a format option. Use the following value for the format modifiers:	
	'text'	Any text you enclose in single or double quotation marks is output without the quotation marks and placed after the appropriate number for the hours, minutes, or seconds.

MX, MO, MB, and MU0C codes: radix conversion

The MX, MO, and MB codes convert data from hexadecimal, octal, and binary format to decimal (base 10) format and vice versa.

Formats

`MX [0C]` Hexadecimal conversion (base 16)

`MO [0C]` Octal conversion (base 8)

`MB [0C]` Binary conversion (base 2)

`MU0C` Hexadecimal Unicode character conversion

With ICONV

The decimal or ASCII format is the internal format for data representation. When used with the `ICONV` function, MX, MO, and MB without the 0C extension convert hexadecimal, octal, or binary data values (respectively) to their equivalent decimal values. MX, MO, and MB with the 0C extension convert hexadecimal, octal, or binary data values to the equivalent ASCII characters rather than to decimal values.

Use the MU0C code only if NLS is enabled. When used with `ICONV`, MU0C converts data in Unicode hexadecimal format to its equivalent in the UniVerse internal character set.

Characters outside of the range for each of the bases produce conversion errors. The ranges are as follows:

Conversion	Ranges
MX (hexadecimal)	0 through 9, A through F, a through f
MO (octal)	0 through 7
MB (binary)	0, 1
MU0C (Unicode)	No characters outside range

With OCONV

When used with the `OCONV` function, MX, MO, and MB without the 0C extension convert decimal values to their equivalent hexadecimal, octal, or binary equivalents for output, respectively. Nonnumeric data produces a conversion error if the 0C extension is not used.

MX, MO, and MB with the 0C extension convert an ASCII character or character string to hexadecimal, octal, or binary output format. Each character in the string is converted to the hexadecimal, octal, or binary equivalent of its ASCII character code.

Use the MU0C code only if NLS is enabled. When used with `OCONV`, MU0C converts characters from their internal representation to their Unicode hexadecimal equivalents for output. The data to convert must be a character or character string in the UniVerse internal character set; each character in the string is converted to its 4-digit Unicode hexadecimal equivalent. Data is converted from left to right, one character at a time, until all data is exhausted.

MY code: ASCII conversion

The MY code specifies conversion from hexadecimal to ASCII on output, and ASCII to hexadecimal on input.

Format

`MY`

When used with the OCONV function, MY converts from hexadecimal to ASCII. When used with the ICONV function, MY converts from ASCII to hexadecimal.

Characters outside of the range for each of the bases produce conversion errors. The ranges are as follows:

MY (hexadecimal)	0 through 9, A through F, a through f

NL code: Arabic numeral conversion

The NL code allows conversion from a locale-dependent set of alternative characters (representing digits in the local language) to Arabic numerals. The alternative characters are the external set, the Arabic characters are the internal set.

Format

`NL`

If NLS is not enabled, characters are checked to ensure only that they are valid ASCII digits 0 through 9, but no characters are changed.

The `STATUS` function returns one of the following:

Value	Description
0	Successful conversion. If NLS is not enabled, input contains valid digits.
1	Unsuccessful conversion. The data to be converted contains a character other than a digit in the appropriate internal or external set.

NLSmapname code: NLS map conversion

The NLS*mapname* code converts data from internal format to external format and vice versa using the specified map. *mapname* is either a valid map name or one of the following: LPTR, CRT, AUX, or OS.

Format

`NLS` *`mapname`*

The `STATUS` function returns one of the following:

Value	Description
0	Conversion successful
1	*mapname* invalid, string returned empty
2	Conversion invalid
3	Data converted, but result may be invalid (map could not deal with some characters)

NR code: roman numeral conversion

The NR code converts Roman numerals into Arabic numerals when used with the `ICONV` function. The decimal, or ASCII, format is the internal format for representation.

When used with the `OCONV` function, the NR code converts Arabic numerals into Roman numerals.

Format

`NR`

The following is a table of Roman/Arabic numeral equivalents:

Roman	Arabic
i	1
v	5
x	10
l	50
c	100
d	500
m	1000
V	5000
X	10,000
L	50,000
C	100,000
D	500,000
M	1,000,000

P code: pattern matching

The P code extracts data whose values match one or more patterns. If the data does not match any of the patterns, an empty string is returned.

Format

`P(`*`pattern`*`) [{ ; | / } (`*`pattern`*`)] …`

pattern can contain one or more of the following codes:

Code	Description
*n*N	An integer followed by the letter N, which tests for *n* numeric characters.
*n*A	An integer followed by the letter A, which tests for *n* alphabetic characters.
*n*X	An integer followed by the letter X, which tests for *n* alphanumeric characters.
nnnn	A literal string, which tests for that literal string.

If *n* is 0, any number of numeric, alphabetic, or alphanumeric characters matches. If either the data or the match pattern is the null value, null is returned.

Separate multiple ranges by a semicolon (;) or a slash (/).

Parentheses must enclose each pattern to be matched. For example, if the user wanted only Social Security numbers returned, P(3N-2N-4N) would test for strings of exactly three numbers, then a hyphen, then exactly two numbers, then a hyphen, then exactly four numbers.

Q code: exponential notation

The Q code converts numeric input data from exponential notation to a format appropriate for internal storage. When converting internal representation of data to external output format, the Q code converts the data to exponential notation by determining how many places to the right of the decimal point are to be displayed and by specifying the exponent.

Format

```
QR [n { E | . } m] [edit] [mask ]
QL [n { E | . } m] [edit] [mask ]
QX
```

Q alone and QR both specify right justification. QL specifies left justification. QX specifies right justification. QX is synonymous with QR0E0 as input and MR as output.

n specifies the number of fractional digits to the right of the decimal point. It can be a number from 0 through 9.

m specifies the exponent. It can be a number from 0 through 9. When used with E, *m* can also be a negative number from –1 through –9.

Separate *n* and *m* with either the letter E or a period (.). Use E if you want to specify a negative exponent.

edit can be any of the following:

Value	Description
$	Prefixes a dollar sign to the value.
F	Prefixes a franc sign to the value.
,	Inserts commas after every thousand.
Z	Returns an empty string if the value is 0. Any trailing fractional zeros are suppressed, and a zero exponent is suppressed.
E	Surrounds negative numbers with angle brackets (< >).
C	Appends cr to negative numbers.
D	Appends db to positive numbers.
B	Appends db to negative numbers.
N	Suppresses a minus sign on negative numbers.
M	Appends a minus sign to negative numbers.
T	Truncates instead of rounding.

mask allows literals to be intermixed with numerics in the formatted output field. The mask can include any combination of literals and the following three special format mask characters:

Character	Description
#*n*	Data is displayed in a field of *n* fill characters. A blank is the default fill character. It is used if the format string does not specify a fill character after the width parameter.
%*n*	Data is displayed in a field of *n* zeros.

Character	Description
**n*	Data is displayed in a field of *n* asterisks.

If NLS is enabled, the Q code formats numeric and monetary values as the ML and MR codes do, except that the *intl* format cannot be specified. See ML and MR codes: formatting numbers, on page 531 for more information.

See the FMT function, on page 169 for more information about formatting numbers.

R code: range function

The R code limits returned data to that which falls within specified ranges. *n* is the lower bound, *m* is the upper bound.

Format

```
Rn,m [{ ; | / } n,m] …
```

Separate multiple ranges by a semicolon (;) or a slash (/).

If range specifications are not met, an empty string is returned.

S (soundex) code

The S code with no arguments specifies a soundex conversion. Soundex is a phonetic converter that converts ordinary English words into a four-character abbreviation comprising one alphabetic character followed by three digits. Soundex conversions are frequently used to build indexes for name lookups.

Format

```
S
```

S (substitution) code

The S code substitutes one of three values depending on whether the data to convert evaluates to 0 or an empty string, to the null value, or to something else.

Format

```
S ; nonzero.substitute ; zero.substitute ; null.substitute
```

If the data to convert evaluates to 0 or an empty string, *zero.substitute* is returned. If the data is nonzero, nonempty, and nonnull, *nonzero.substitute* is returned. If the data is the null value, *null.substitute* is returned. If *null.substitute* is omitted, null values are not replaced.

All three substitute expressions can be one of the following:

- A quoted string
- A field number
- An asterisk

If it is an asterisk and the data evaluates to something other than 0, the empty string, or the null value, the data value itself is returned.

Example

Assume a BASIC program where @RECORD is:

```
AFBFCVD
```

Statement	Output
PRINT OCONV("x","S;2;'zero'")	B
PRINT OCONV("x","S;*;'zero'")	x
PRINT OCONV(0,"S;2;'zero'")	zero
PRINT OCONV('',"S;*;'zero'")	zero

T code: text extraction

The T code extracts a contiguous string of characters from a field.

Format

```
T [start,] length
```

Parameter	Description
start	Starting column number. If omitted, 1 is assumed.
length	Number of characters to extract.

If you specify *length* only, the extraction is either from the left or from the right depending on the justification specified in line 5 of the dictionary definition item. In a BASIC program if you specify *length* only, the extraction is from the right. In this case the starting position is calculated according to the following formula:

```
string.length - substring.length + 1
```

This lets you extract the last *n* characters of a string without having to calculate the string length.

If *start* is specified, extraction is always from left to right.

Tfile code: file translation

The T*file* code converts values from one file to another by translating through a file. It uses data values in the source file as IDs for records in a lookup file. The source file can then reference values in the lookup file.

Format

```
T[DICT] filename ; c [vloc] ; [iloc] ; [oloc] [;bloc]
```

```
T[DICT] filename ; c ; [iloc] ; [oloc] [ ;bloc] [,vloc | [vloc]]
```

To access the lookup file, its record IDs (field 0) must be referenced. If no reference is made to the record IDs of the lookup file, the file cannot be opened and the conversion cannot be performed. The data value being converted must be a record ID in the lookup file.

Parameter	Description
DICT	Specifies the lookup file's dictionary. (In REALITY flavor accounts, you can use an asterisk (*) to specify the dictionary: for instance, T**filename** … .)
filename	Name of the lookup file.

Parameter	Description	
c	Translation subcode, which must be one of the following:	
	V	Conversion item must exist on file, and the specified field must have a value, otherwise an error message is returned.
	C	If conversion is impossible, return the original value-to-be-translated.
	I	Input verify only. Functions like V for input and like C for output.
	N	Returns the original value-to-be-translated if the null value is found.
	O	Output verify only. Functions like C for input and like V for output.
	X	If conversion is impossible, return an empty string.
vloc	Number of the value to be returned from a multivalued field. If you do not specify *vloc* and the field is multivalued, the whole field is returned with all system delimiters turned into blanks. If the *vloc* specification follows the *oloc* or *bloc* specification, enclose *vloc* in square brackets or separate *vloc* from *oloc* or *bloc* with a comma.	
iloc	Field number (decimal) for *input* conversion. The input value is used as a record ID in the lookup file, and the translated value is retrieved from the field specified by the *iloc*. If the *iloc* is omitted, no input translation takes place.	
oloc	Field number (decimal) for *output* translation. When RetrieVe creates a listing, data from the field specified by *oloc* in the lookup file are listed instead of the original value.	
bloc	Field number (decimal) which is used instead of *oloc* during the listing of BREAK.ON and TOTAL lines.	

TI code: international time conversion

The international time conversion lets you convert times in internal format to the default local convention format and vice versa. If NLS locales are not enabled, the TI conversion defaults to MT. If NLS locales are enabled, TI uses the date conversion in the TI_FMT field of the Time category. The TI_FMT field can contain any valid MT code.

Format

```
TI
```

Appendix D: BASIC reserved words

ABORTE
ABORTM
ABS
ABSS
ACOS
ADDS
ALL
ALPHA
AND
ANDS
ARG.
ASCII
ASIN
ASSIGN
ASSIGNED
ATAN
AUTHORIZATION
BCONVERT
BEFORE
BEGIN
BITAND
BITNOT
BITOR
BITRESET
BITSET
BITTEST
BITXOR
BREAK
BSCAN
BY
CALL
CALLING
CAPTURING
CASE
CAT

CATS
CHAIN
CHANGE
CHAR
CHARS
CHECKSUM
CLEAR
CLEARCOMMON
CLEARDATA
CLEARFILE
CLEARINPUT
CLEARPROMPTS
CLEARSELECT
CLOSE
CLOSESEQ
COL1
COL2
COM
COMMIT
COMMON
COMPARE
CONTINUE
CONVERT
COS
COSH
COUNT
COUNTS
CREATE
CRT
DATA
DATE
DCOUNT
DEBUG
DECLARE
DEFFUN
DEL
DELETE
DELETELIST

DELETEU
DIAGNOSTICS
DIM
DIMENSION
DISPLAY
DIV
DIVS
DO
DOWNCASE
DQUOTE
DTX
EBCDIC
ECHO
ELSE
END
ENTER
EOF
EQ
EQS
EQU
EQUATE
EREPLACE
ERRMSG
ERROR
EXCHANGE
EXEC
EXECUTE
EXIT
EXP
EXTRACT
FADD
FDIV
FFIX
FFLT
FIELD
FIELDS
FIELDSTORE
FILEINFO

FILELOCK

FILEUNLOCK

FIND

FINDSTR

FIX

FLUSH

FMT

FMTS

FMUL

FOLD

FOOTING

FOR

FORMLIST

FROM

FSUB

FUNCTION

GARBAGECOLLECT

GCI

GE

GES

GET

GETLIST

GETREM

GETX

GO

GOSUB

GOTO

GROUP

GROUPSTORE

GT

GTS

HEADING

HEADINGE

HEADINGN

HUSH

ICHECK

ICONV

ICONVS

IF

IFS

ILPROMPT

IN

INCLUDE

INDEX

INDEXS

INDICES

INMAT

INPUT

INPUTCLEAR

INPUTDISP

INPUTERR

INPUTIF

INPUTNULL

INPUTTRAP

INS

INSERT

INT

ISNULL

ISNULLS

ISOLATION

ITYPE

KEY

KEYEDIT

KEYEXIT

KEYIN

KEYTRAP

LE

LEFT

LEN

LENS

LES

LET

LEVEL

LIT

LITERALLY

LN

LOCATE
LOCK
LOCKED
LOOP
LOWER
LPTR
LT
LTS
MAT
MATBUILD
MATCH
MATCHES
MATCHFIELD
MATPARSE
MATREAD
MATREADL
MATREADU
MATWRITE
MATWRITEU
MAXIMUM
MESSAGE
MINIMUM
MOD
MODS
MTU
MULS
NAP
NE
NEG
NEGS
NES
NEXT
NOBUF
NO.ISOLATION
NOT
NOTS
NULL
NUM

NUMS

OCONV

OCONVS

OFF

ON

OPEN

OPENCHECK

OPENDEV

OPENPATH

OPENSEQ

OR

ORS

OUT

PAGE

PASSLIST

PCDRIVER

PERFORM

PRECISION

PRINT

PRINTER

PRINTERIO

PRINTERR

PROCREAD

PROCWRITE

PROG

PROGRAM

PROMPT

PWR

QUOTE

RAISE

RANDOMIZE

READ

READ.COMMITTED

READ.UNCOMMITTED

READBLK

READL

READLIST

READNEXT

READSEQ
READT
READU
READV
READVL
READVU
REAL
RECIO
RECORDLOCKED
RECORDLOCKL
RECORDLOCKU
RELEASE
REM
REMOVE
REPEAT
REPEATABLE.READ
REPLACE
RESET
RETURN
RETURNING
REUSE
REVREMOVE
REWIND
RIGHT
RND
ROLLBACK
RPC.CALL
RPC.CONNECT
RPC.DISCONNECT
RQM
RTNLIST
SADD
SCMP
SDIV
SEEK
SELECT
SELECTE
SELECTINDEX

SELECTN

SELECTV

SEND

SENTENCE

SEQ

SEQS

SEQSUM

SERIALIZABLE

SET

SETREM

SETTING

SIN

SINH

SLEEP

SMUL

SOUNDEX

SPACE

SPACES

SPLICE

SQLALLOCONNECT

SQLALLOCENV

SQLALLOCSTMT

SQLBINDCOL

SQLCANCEL

SQLCOLATTRI- BUTES

SQLCONNECT

SQLDESCRIBECOL

SQLDISCONNECT

SQLERROR

SQLEXECDIRECT

SQLEXECUTE

SQLFETCH

SQLFREECONNECT

SQLFREEENV

SQLFREESTMT

SQLGETCURSORNAME

SQLNUMRESULTCOLS

SQLPREPARE

SQLROWCOUNT
SQLSETCONNECT-OPTION
SQLSETCURSORNAME
SQLSETPARAM
SQRT
SQUOTE
SSELECT
SSELECTN
SSELECTV
SSUB
START
STATUS
STEP
STOP
STOPE
STOPM
STORAGE
STR
STRS
SUB
SUBR
SUBROUTINE
SUBS
SUBSTRINGS
SUM
SUMMATION
SYSTEM
TABSTOP
TAN
TANH
TERMINFO
THEN
TIME
TIMEDATE
TIMEOUT
TO
TPARM
TPRINT

TRANS
TRANSACTION
TRIM
TRIMB
TRIMBS
TRIMF
TRIMFS
TRIMS
TTYCTL
TTYGET
TTYSET
UNASSIGNED
UNIT
UNLOCK
UNTIL
UPCASE
USING
WEOF
WEOFSEQ
WEOFSEQF
WHILE
WORDSIZE
WORKWRITE
WRITEBLK
WRITELIST
WRITESEQ
WRITESEQF
WRITET
WRITEU
WRITEV
WRITEVU
XLATE
XTD

Appendix E: @Variables

The following table lists BASIC @variables. The @variables denoted by an asterisk (*) are read-only. All others can be changed by the user.

The `EXECUTE` statement initializes the values of stacked @variables either to 0 or to values reflecting the new environment. These values are not passed back to the calling environment. The values of nonstacked @variables are shared between the `EXECUTE` and calling environments. All @variables listed here are stacked unless otherwise indicated.

Variable	Read-only	Value
@ABORT.CODE	*	A numeric value indicating the type of condition that caused the ON.ABORT paragraph to execute. The values are: 1 – An ABORT statement was executed. 2 – An abort was requested after pressing the Break key followed by option A. 3 – An internal or fatal error occurred. 4 – An AUTO.LOGOUT event occurred.
@ACCOUNT	*	User login name. Same as @LOGNAME. Nonstacked.
@AM	*	Field mark: CHAR(254). Same as @FM.
@ANS		Last I-type answer, value indeterminate.
@AUTHORIZATION	*	Current effective user name.
@COMMAND	*	Last command executed or entered at the UniVerse prompt.
@COMMAND.STACK	*	Dynamic array containing the last 99 commands executed.
@CONV		For future use.
@CRTHIGH	*	Number of lines on the terminal.
@CRTWIDE	*	Number of columns on the terminal.
@DATA.PENDING	*	Dynamic array containing input generated by the DATA statement. Values in the dynamic array are separated by field marks.
@DATE		Internal date when the program was invoked.
@DAY		Day of month from @DATE.
@DICT		For future use.
@FALSE	*	Compiler replaces the value with 0.
@FILE.NAME		Current file name. When used in a virtual field index, @FILENAME reflects the current file name being used in a RetrieVe or UniVerse SQL statement. Same as @FILENAME.
@FILENAME		Current file name. When used in a virtual field index, @FILENAME reflects the current file name being used in a RetrieVe or UniVerse SQL statement. Same as @FILE.NAME.
@FM	*	Field mark: CHAR(254). Same as @AM.
@FORMAT		For future use.

Variable	Read-only	Value
@HDBC	*	ODBC connection environment on the local UniVerse server. Nonstacked.
@HEADER		For future use.
@HENV	*	ODBC environment on the local UniVerse server. Nonstacked.
@HSTMT	*	ODBC statement environment on the local UniVerse server. Nonstacked.
@ID		Current record ID.
@IDX.FILEPATH		Can be used within an indexed subroutine. Contains the full path of the UniVerse file being updated that caused the indexed subroutine to fire.
@IDX.IOTYPE		Specifies the type of operation being performed. Can be integrated in the indexed subroutine to determine they type of database operation that caused the indexed subroutine to fire. The following values are associated with the @IDX.IOTYPE: 0 - The value returned when @IDX.IOTYPE is used outside the context of an indexed subroutine. 1 - The value returned when the SUBR is called because an INSERT operation is performed. 2 - The value returned when the SUBR is called because a DELETE operation is performed. 3 - The value returned when the SUBR is called because an UPDATE operation is used to evaluate the original value operation. 4 - The value returned when a SUBR i called because an UPDATE operation is used to evaluate the new value operation.
@IM	*	Item mark: CHAR(255).
@ISOLATION	*	Current transaction isolation level for the active transaction or the current default isolation level if no transaction exists.
@LEVEL	*	Nesting level of execution statements. Nonstacked.
@LOGNAME	*	User login name. Same as @ACCOUNT.
@LPTRHIGH	*	Number of lines on the device to which you are printing (that is, terminal or printer).
@LPTRWIDE	*	Number of columns on the device to which you are printing (that is, terminal or printer).
@MONTH		Current month.
@MV		Current value counter for columnar listing only. Used only in I-descriptors. Same as @NV.
@NB		Current BREAK level number. 1 is the lowest-level break. @NB has a value of 255 on the grand total line. Used only in I-descriptors.
@ND		Number of detail lines since the last BREAK on a break line. Used only in I-descriptors.
@NI		Current item counter (the number of items listed or selected). Used only in I-descriptors. Same as @RECCOUNT.

Variable	Read-only	Value
@NS		Current subvalue counter for columnar listing only. Used only in I-descriptors.
@NULL	*	The null value. Nonstacked.
@NULL.STR	*	Internal representation of the null value, which is CHAR(128). Nonstacked.
@NV		Current value counter for columnar listing only. Used only in I-descriptors. Same as @MV.
@OPTION		Value of field 5 in the VOC for the calling verb.
@PARASENTENCE	*	Last sentence or paragraph that invoked the current process.
@PATH	*	Path name of the current account.
@PYEXCEPTIONMSG		A string that stores the detailed exception message; if no exception is thrown, its value is an empty string.
@PYEXCEPTIONTRACEBACK		A string that stores the traceback of the exception; if no exception is thrown, its value is an empty string.
@PYEXCEPTIONTYPE		A string that stores the exception type; if no exception is thrown, its value is an empty string.
@RECCOUNT		Current item counter (the number of items listed or selected). Used only in I-descriptors. Same as @NI.
@RECORD		Entire current record.
@RECUR0		Reserved.
@RECUR1		Reserved.
@RECUR2		Reserved.
@RECUR3		Reserved.
@RECUR4		Reserved.
@SCHEMA	*	Schema name of the current UniVerse account. Nonstacked. When users create a new schema, @SCHEMA is not set until the next time they log in to UniVerse.
@SELECTED		Number of elements selected from the last select list. Nonstacked.
@SENTENCE	*	Sentence that invoked the current BASIC program. Any EXECUTE statement updates @SENTENCE.
@SM	*	Subvalue mark: CHAR(252). Same as @SVM.
@SQL.CODE	*	For future use.
@SQL.DATE	*	Current system date. Use in trigger programs. Nonstacked.
@SQL.ERROR	*	For future use.
@SQL.STATE	*	For future use.
@SQL.TIME	*	Current system time. Use in trigger programs. Nonstacked.
@SQL.WARNING	*	For future use.
@SQLPROC.NAME	*	Name of the current SQL procedure.
@SQLPROC.TX.LEVEL	*	Transaction level at which the current SQL procedure began.
@STDFIL		Default file variable.
@SVM	*	Subvalue mark: CHAR(252). Same as @SM.
@SYS.BELL	*	Bell character. Nonstacked.
@SYSTEM.RETURN.CODE		Status codes returned by system processes. Same as @SYSTEM.SET.

Variable	Read-only	Value
@SYSTEM.SET		Status codes returned by system processes. Same as @SYSTEM.RETURN.CODE.
@TERM.TYPE	*	Terminal type. Nonstacked.
@TIME		Internal time when the program was invoked.
@TM	*	Text mark: CHAR(251).
@TRANSACTION	*	A numeric value. Any nonzero value indicates that a transaction is active; the value 0 indicates that no transaction exists.
@TRANSACTION.ID	*	Transaction number of the active transaction. An empty string indicates that no transaction exists.
@TRANSACTION.LEVEL	*	Transaction nesting level of the active transaction. A 0 indicates that no transaction exists.
@TRUE		Compiler replaces the value with 1.
@TTY		Terminal device name. If the process is a phantom, @TTY returns the value ‘phantom’. If the process is a UniVerse API, it returns ‘uvcs’. Note: In PI/Open flavor, @TTY returns an empty string for PHANTOM processes.
@USER0		User-defined.
@USER1		User-defined.
@USER2		User-defined.
@USER3		User-defined.
@USER4		User-defined.
@USERNO	*	User number. Nonstacked. Same as @USER.NO.
@USER.NO	*	User number. Nonstacked. Same as @USERNO.
@USER.RETURN.CODE		Status codes created by the user.
@VM	*	Value mark: CHAR(253).
@WHO	*	Name of the current UniVerse account directory. Nonstacked.
@YEAR		Current year (2 digits).
@YEAR4		Current year (4 digits).

Appendix F: BASIC subroutines

This appendix describes the following subroutines you can call from a UniVerse BASIC program:

! ASYNC subroutine (!AMLC)

!EDIT.INPUT subroutine

!ERRNO subroutine

!FCMP subroutine

!GET.KEY subroutine

!GET.PARTNUM subroutine

!GET.PATHNAME subroutine

!GET.USER.COUNTS subroutine

!GETPU subroutine

!INLINE.PROMPTS subroutine

!INTS subroutine

!MAKE.PATHNAME subroutine

!MATCHES subroutine

!MESSAGE subroutine

!PACK.FNKEYS subroutine

!REPORT.ERROR subroutine

!SET.PTR subroutine

!SETPU subroutine

!TIMDAT subroutine

!USER.TYPE subroutine

!VOC.PATHNAME subroutine

In addition, the subroutines listed in the following table have been added to existing functions for PI/open compatibility.

Subroutine	Associated function
CALL !ADDS	ADDS
CALL !ANDS	ANDS
CALL !CATS	CATS
CALL !CHARS	CHARS
CALL !CLEAR.PROMPTS	CLEARPROMPTS
CALL !COUNTS	COUNTS
CALL !DISLEN	LENDP
CALL !DIVS	DIVS
CALL !EQS	EQS
CALL !FADD	FADD
CALL !FDIV	FDIV
CALL !FIELDS	FIELDS

Subroutine	Associated function
CALL !FMTS	FMTS
CALL !FMUL	FMUL
CALL !FOLD	FOLD
CALL !FSUB	FSUB
CALL !GES	GES
CALL !GTS	GTS
CALL !ICONVS	ICONVS
CALL !IFS	IFS
CALL !INDEXS	INDEXS
CALL !LENS	LENS
CALL !LES	LES
CALL !LTS	LTS
CALL !MAXIMUM	MAXIMUM
CALL !MINIMUM	MINIMUM
CALL !MODS	MODS
CALL !MULS	MULS
CALL !NES	NES
CALL !NOTS	NOTS
CALL !NUMS	NUMS
CALL !OCONVS	OCONVS
CALL !ORS	ORS
CALL !SEQS	SEQS
CALL !SPACES	SPACES
CALL !SPLICE	SPLICE
CALL !STRS	STRS
CALL !SUBS	SUBS
CALL !SUBSTRINGS	SUBSTRINGS
CALL !SUMMATION	SUMMATION

! ASYNC subroutine

Use the !ASYNC subroutine (or its synonym !AMLC) to send data to, and receive data from an asynchronous device.

Syntax

```
CALL !ASYNC (key, line, data, count, carrier)
```

key defines the action to be taken (1 through 5). The values for *key* are defined in the following list:

line is the number portion from the &DEVICE& entry TTY##, where ## represents a decimal number.

data is the data being sent to or received from the line.

count is an output variable containing the character count.

carrier is an output variable that returns a value dependent on the value of *key*. If *key* is 1, 2, or 3, *carrier* returns the variable specified by the user. If *key* has a value of 4 or 5, *carrier* returns 1.

You must first assign an asynchronous device using the ASSIGN command. An entry must be in the &DEVICE& file for the device to be assigned with the record ID format of TTY##, where ## represents a decimal number. The actions associated with each key value are as follows:

Key	Action
1	Inputs the number of characters indicated by the value of *count*.
2	Inputs the number of characters indicated by the value of *count* or until a linefeed character is encountered.
3	Outputs the number of characters indicated by the value of *count*.
4	Returns the number of characters in the input buffer to *count*. On operating systems where the FIONREAD key is not supported, 0 is returned in *count*. When the value of *key* is 4, 1 is always returned to *carrier*.
5	Returns 0 in *count* if there is insufficient space in the output buffer. On operating systems where the TIOCOUTQ key is not supported, 0 is returned in *count*. When the value of *key* is 5, 1 is always returned to *carrier*.

Example

The !ASYNC subroutine returns the first 80 characters from the device defined by ASYNC10 in the &DEVICE& file to the variable data.

```
data=
 count= 80
 carrier= 0
 call !ASYNC(1,10,data,count,carrier)
```

!EDIT.INPUT subroutine

Use the !EDIT.INPUT subroutine to request editable terminal input within a single-line window on the terminal. Editing keys are defined in the *terminfo* files and can be set up using the KEYEDIT statement, KEYTRAP statement and KEYEDIT statement. To ease the implementation, the UNIVERSE.INCLUDE file GTI.FNKEYS.IH can be included to automatically define the editing keys from the current *terminfo* definition. We recommend that you use the INCLUDE file.

Syntax

```
CALL !EDIT.INPUT (keys, wcol, wrow, wwidth, buffer, startpos, bwidth,
ftable, code)
```

All input occurs within a single-line window of the terminal screen, defined by the parameters *wrow*, *wcol*, and *wwidth*. If the underlying buffer length *bwidth* is greater than *wwidth* and the user performs a function that moves the cursor out of the window horizontally, the contents of buffer are scrolled so as to keep the cursor always in the window.

If the specified starting cursor position would take the cursor out of the window, the buffer's contents are scrolled immediately so as to keep the cursor visible. !EDIT.INPUT does not let the user enter more than *bwidth* characters into the buffer, regardless of the value of *wwidth*.

Qualifiers

Qualifier	Description		
keys	Controls certain operational characteristics. *keys* can take the additive values (the token names can be found in the GTI.FNKEYS.IH include file) shown here:		
	Value	**Token**	**Description**
	0	IK$NON	None of the keys below are required.
	1	IK$OCR	Output a carriage return.
	2	IK$ATM	Terminate editing as soon as the user has entered *bwidth* characters.
	4	IK$TCR	Toggle cursor-visible state.
	8	IK$DIS	Display contents of buffer string on entry.
	16	IK$HDX	Set terminal to half-duplex mode (restored on exit).
	32	IK$INS	Start editing in insert mode. Default is overlay mode.
	64	IK$BEG	Separate Begin Line/End Line functionality required.
wcol	The screen column of the start of the window (x-coordinate).		
wrow	The screen row for the window (y-coordinate).		
wwidth	The number of screen columns the window occupies.		
buffer	Contains the following:		
	on entry	The text to display (if key IK$DIS is set).	
	on exit	The final edited value of the text.	
startpos	Indicates the cursor position as follows:		
	on entry	The initial position of the cursor (from start of buffer).	
	on exit	The position of the cursor upon exit.	
bwidth	The maximum number of positions allowed in *buffer*. *bwidth* can be more than *wwidth*, in which case the contents of *buffer* scroll horizontally as required.		
ftable	A packed function key trap table, defining which keys cause exit from the !EDIT.INPUT function. The !PACK.FNKEYS function creates the packed function key trap table.		
code	The reply code:		
	= 0	User pressed Return or entered *bwidth* characters and IK$ATM was set.	
	> 0	The function key number that terminated !EDIT.INPUT.	

!EDIT.INPUT functions

`!EDIT.INPUT` performs up to eight editing functions, as follows:

Value	Token	Description
3	FK$BSP	Backspace
4	FK$LEFT	Cursor left
5	FK$RIGHT	Cursor right
19	FK$INSCH	Insert character
21	FK$INSTXT	Insert/overlay mode toggle
23	FK$DELCH	Delete character
24	FK$DELLIN	Delete line
51	FK$CLEOL	Clear to end-of-line

The specific keys to perform each function can be automatically initialized by including the $INCLUDE UNIVERSE.INCLUDE GTI.FNKEYS.IH statement in the application program.

If any of the values appear in the trap list, its functionality is disabled and the program immediate exits the `!EDIT.INPUT` subroutine when the key associated with that function is pressed.

Unsupported functions

This implementation does not support a number of functions originally available in the Prime INFORMATION version. Because of this, sequences can be generated that inadvertently cause the `!EDIT.INPUT` function to terminate. For this reason, you can create a user-defined terminal keystroke definition file so that `!EDIT.INPUT` recognizes the unsupported sequences. Unsupported sequences cause the `!EDIT.INPUT` subroutine to ring the terminal bell, indicating the recognition of an invalid sequence.

The file CUSTOM.GTI.DEFS defines a series of keystroke sequences for this purpose. You can create the file in each account or in a central location, with VOC entries in satellite accounts referencing the remote file. There is no restriction on how the file can be created. For instance, you can use the command:

```
>CREATE.FILE CUSTOM.GTI.DEFS 2 17 1 /* Information style */
```

or:

```
>CREATE-FILE CUSTOM.GTI.DEFS (1,1,3 17,1,2) /* Pick style */
```

to create the definition file. A terminal keystroke definition record assumes the name of the terminal which the definitions are associated with, for example for *vt100* terminals, the CUSTOM.GTI.DEFS file record ID would be vt100 (case-sensitive). Each terminal keystroke definition record contains a maximum of 82 fields (attributes) which directly correspond to the keystroke code listed in the GTI.FNKEYS.IH include file.

The complete listing of the fields defined within the GTI.FNKEYS.IH include file is shown below:

Key name	Field	Description
FK$FIN	1	Finish
FK$HELP	2	Help
FK$BSP	3	Backspace[a]
FK$LEFT	4	Left arrow[a]
FK$RIGHT	5	Right arrow[a]
FK$UP	6	Up arrow
FK$DOWN	7	Down arrow
FK$LSCR	8	Left screen
FK$RSCR	9	Right screen
FK$USCR	10	Up screen, Previous page
FK$DSCR	11	Down screen, Next page
FK$BEGEND	12	Toggle begin/end line, or Begin line
FK$TOPBOT	13	Top/Bottom, or End line
FK$NEXTWD	14	Next word
FK$PREVWD	15	Previous word
FK$TAB	16	Tab
FK$BTAB	17	Backtab
FK$CTAB	18	Column tab
FK$INSCH	19	Insert character (space)[a]

Key name	Field	Description
FK$INSLIN	20	Insert line
FK$INSTXT	21	Insert text, Toggle insert/overlay mode[a]
FK$INSDOC	22	Insert document
FK$DELCH	23	Delete character[a]
FK$DELLIN	24	Delete line[a]
FK$DELTXT	25	Delete text
FK$SRCHNX	26	Search next
FK$SEARCH	27	Search
FK$REPLACE	28	Replace
FK$MOVE	29	Move text
FK$COPY	30	Copy text
FK$SAVE	31	Save text
FK$FMT	32	Call format line
FK$CONFMT	33	Confirm format line
FK$CONFMTNW	34	Confirm format line, no wrap
FK$OOPS	35	Oops
FK$GOTO	36	Goto
FK$CALC	37	Recalculate
FK$INDENT	38	Indent (set left margin)
FK$MARK	39	Mark
FK$ATT	40	Set attribute
FK$CENTER	41	Center
FK$HYPH	42	Hyphenate
FK$REPAGE	43	Repaginate
FK$ABBREV	44	Abbreviation
FK$SPELL	45	Check spelling
FK$FORM	46	Enter formula
FK$HOME	47	Home the cursor
FK$CMD	48	Enter command
FK$EDIT	49	Edit
FK$CANCEL	50	Abort/Cancel
FK$CLEOL	51	Clear to end of line1
FK$SCRWID	52	Toggle between 80 and 132 mode
FK$PERF	53	Invoke DSS PERFORM emulator
FK$INCLUDE	54	DSS Include scratchpad data
FK$EXPORT	55	DSS Export scratchpad data
FK$TWIDDLE	56	Twiddle character pair
FK$DELWD	57	Delete word
FK$SRCHPREV	58	Search previous
FK$LANGUAGE	59	Language
FK$REFRESH	60	Refresh
FK$UPPER	61	Uppercase
FK$LOWER	62	Lowercase

Key name	Field	Description
FK$CAPIT	63	Capitalize
FK$REPEAT	64	Repeat
FK$STAMP	65	Stamp
FK$SPOOL	66	Spool record
FK$GET	67	Get record
FK$WRITE	68	Write record
FK$EXECUTE	69	Execute macro
FK$NUMBER	70	Toggle line numbering
FK$DTAB	71	Clear tabs
FK$STOP	72	Stop (current activity)
FK$EXCHANGE	73	Exchange mark and cursor
FK$BOTTOM	74	Move bottom
FK$CASE	75	Toggle case sensitivity
FK$LISTB	76	List (buffers)
FK$LISTD	77	List (deletions)
FK$LISTA	78	List (selects)
FK$LISTC	79	List (commands)
FK$DISPLAY	80	Display (current select list)
FK$BLOCK	81	Block (replace)
FK$PREFIX	82	Prefix

a. Indicates supported functionality.

Example

The following BASIC program sets up three trap keys (using the !PACK.FNKEYS subroutine), waits for the user to enter input, then reports how the input was terminated:

```
$INCLUDE UNIVERSE.INCLUDE GTI.FNKEYS.IH
 * Set up trap keys of FINISH, UPCURSOR and DOWNCURSOR
 TRAP.LIST = FK$FIN:@FM:FK$UP:@FM:FK$DOWN
 CALL !PACK.FNKEYS(TRAP.LIST, Ftable)
 * Start editing in INPUT mode, displaying contents in window
 KEYS = IK$INS + IK$DIS
 * Window edit is at x=20, y=2, of length 10 characters;
 * the user can enter up to 30 characters of input into TextBuffer,
 * and the cursor is initially placed on the first character of the
 * window.
 TextBuffer=""
 CursorPos = 1
 CALL !EDIT.INPUT(KEYS, 20, 2, 10, TextBuffer, CursorPos, 30, Ftable,
    ReturnCode)
  * On exit, the user's input is within TextBuffer,
 * CursorPos indicates the location of the cursor upon exiting,
 * and ReturnCode contains the reason for exiting.
 BEGIN CASE
        CASE CODE = 0       * User pressed RETURN key
        CASE CODE = FK$FIN      * User pressed the defined FINISH key
        CASE CODE = FK$UP      * User pressed the defined UPCURSOR key
            CASE CODE = FK$DOWN   * User pressed the defined DOWNCURSOR key
            CASE 1               * Should never happen
```

```
END CASE
```

!ERRNO subroutine

Use the !ERRNO subroutine to return the current value of the operating system *errno* variable.

Syntax

```
CALL !ERRNO (variable)
```

variable is the name of a BASIC variable.

The !ERRNO subroutine returns the value of the system *errno* variable after the last call to a GCI subroutine in *variable*. If you call a system routine with the GCI, and the system call fails, you can use !ERRNO to determine what caused the failure. If no GCI routine was called prior to its execution, !ERRNO returns 0. The values of *errno* that apply to your system are listed in the system include file *errno.h*.

!FCMP subroutine

Use the !FCMP subroutine to compare the equality of two floating-point numeric values as follows:

If *number1* is less than *number2*, *result* is –1.

If *number1* is equal to *number2*, *result* is 0.

If *number1* is greater than *number2*, *result* is 1.

Syntax

```
CALL !FCMP (result , number1 ,  number2 )
```

!GET.KEY subroutine

Use the !GET.KEY subroutine to return the next key pressed at the keyboard. This can be either a printing character, the Return key, a function key as defined by the current terminal type, or a character sequence that begins with an escape or control character not defined as a function key.

Function keys can be automatically initialized by including the $INCLUDE UNIVERSE.INCLUDES GTI.FNKEYS.IH statement in the application program that uses the !GET.KEY subroutine.

Syntax

```
CALL !GET.KEY (string, code)
```

Qualifiers

Code	String value
string	Returns the character sequence of the next key pressed at the keyboard.

Code	String value	
code	Returns the string interpretation value:	
	Code	**String Value**
	0	A single character that is not part of any function key sequence. For example, if A is pressed, *code* = 0 and *string* = CHAR(65).
	>0	The character sequence associated with the function key defined by that number in the GTI.FNKEYS.IH include file. For example, on a VT100 terminal, pressing the key labeled --> (right cursor move) returns *code* = 5 and *string* = CHAR(27):CHAR(79):CHAR(67).
	<0	A character sequence starting with an escape or control character that does not match any sequence in either the *terminfo* entry or the CUSTOM.GCI.DEFS file.

Example

The following BASIC program waits for the user to enter input, then reports the type of input entered:

```
$INCLUDE GTI.FNKEYS.IH
        STRING = '  ' ; * initial states of call variables
    CODE = -999
    * Now ask for input until user hits a "Q"
    LOOP
    UNTIL STRING[1,1] = "q" OR STRING[1,1] = "Q"
                    PRINT 'Type a character or press a function key (q to quit):':
                CALL !GET.KEY(STRING, CODE)
                * Display meaning of CODE
                PRINT
                PRINT "CODE = ":CODE:
                BEGIN CASE
                            CASE CODE = 0
                                        PRINT "     (Normal character)"
                                CASE CODE > 0
                                        PRINT "     (Function key number)"
                            CASE 1; * otherwise
                                        PRINT "     (Unrecognized function key)"
                END CASE
              * Print whatever is in STRING, as decimal numbers:
                PRINT "STRING = ":
                FOR I = 1 TO LEN(STRING)
                            PRINT "CHAR(":SEQ(STRING[I,1]):") ":
                NEXT I
                PRINT
    REPEAT
    PRINT "End of run."
    RETURN
END
```

!GET.PARTNUM subroutine

Use the !GET.PARTNUM subroutine with distributed files to determine the number of the part file to which a given record ID belongs.

Syntax

```
CALL !GET.PARTNUM (file, record.ID, partnum, status)
```

file (input) is the file variable of the open distributed file.

record.ID (input) is the record ID.

partnum (output) is the part number of the part file of the distributed file to which the given record ID maps.

status (output) is 0 for a valid part number or an error number for an invalid part number. An insert file of equate tokens for the error numbers is available.

An insert file of equate names is provided to allow you to use mnemonics for the error numbers. The insert file is called INFO_ERRORS.INS.IBAS, and is located in the *INCLUDE* subdirectory. To use the insert file, specify $INCLUDE statement SYSCOM INFO_ERRORS.INS.IBAS when you compile the program.

Equate name	Description
IE$NOT.DISTFILE	The file specified by the file variable is not a distributed file.
IE$DIST.DICT.OPEN.FAIL	The program failed to open the file dictionary for the distributed file.
IE$DIST.ALG.READ.FAIL	The program failed to read the partitioning algorithm from the distributed file dictionary.
IE$NO.MAP.TO.PARTNUM	The record ID specified is not valid for this distributed file.

Use the !GET.PARTNUM subroutine to call the partitioning algorithm associated with a distributed file. If the part number returned by the partitioning algorithm is not valid, that is, not an integer greater than zero, !GET.PARTNUM returns a nonzero status code. If the part number returned by the partitioning algorithm is valid, !GET.PARTNUM returns a zero status code.

Note: !GET.PARTNUM does not check that the returned part number corresponds to one of the available part files of the currently opened file.

Example

In the following example, a distributed file SYS has been defined with parts and part numbers S1, 5, S2, 7, and S3, 3, respectively. The file uses the default SYSTEM partitioning algorithm.

```
PROMPT ''
 GET.PARTNUM = '!GET.PARTNUM'
 STATUS = 0
 PART.NUM = 0
 OPEN '', 'SYS' TO FVAR ELSE STOP 'NO OPEN SYS'
 PATHNAME.LIST = FILEINFO(FVAR, FINFO$PATHNAME)
 PARTNUM.LIST = FILEINFO(FVAR, FINFO$PARTNUM)
 LOOP
                PRINT 'ENTER Record ID : ':
            INPUT RECORD.ID
 WHILE RECORD.ID
            CALL @GET.PARTNUM(FVAR, RECORD.ID, PART.NUM, STATUS)
            LOCATE PART.NUM IN PARTNUM.LIST<1> SETTING PART.INDEX THEN
 PATHNAME = PATHNAME.LIST <PART.INDEX>
            END ELSE
                                       PATHNAME = ''
       END
                PRINT 'PART.NUM = ':PART.NUM:'  STATUS = ':STATUS :'
```

```
        PATHNAME = ': PATHNAME
  REPEAT
  END
```

!GET.PARTNUM returns part number 5 for input record ID 5-1, with status code 0, and part number 7 for input record ID 7-1, with status code 0, and part number 3 for input record ID 3-1, with status code 0. These part numbers are valid and correspond to available part files of file SYS.

!GET.PARTNUM returns part number 1200 for input record ID 1200-1, with status code 0. This part number is valid but does not correspond to an available part file of file SYS.

!GET.PARTNUM returns part number 0 for input record ID 5-1, with status code IE $NO.MAP.TO.PARTNUM, and part number 0 for input record ID A-1, with status code IE $NO.MAP.TO.PARTNUM, and part number 0 for input record ID 12-4, with status code IE $NO.MAP.TO.PARTNUM. These part numbers are not valid and do not correspond to available part files of the file SYS.

!GET.PATHNAME subroutine

Use the !GET.PATHNAME subroutine to return the directory name and file name parts of a path name.

Syntax

```
CALL !GET.PATHNAME (pathname, directoryname, filename, status)
```

pathname (input) is the path name from which the details are required.

directoryname (output) is the directory name portion of the path name, that is, the path name with the last entry name stripped off.

filename (output) is the file name portion of the path name.

status (output) is the returned status of the operation. A 0 indicates success, another number is an error code indicating that the supplied path name was not valid.

Example

If *pathname* is input as /usr/accounts/ledger, *directoryname* is returned as /usr/accounts, and *filename* is returned as ledger.

```
PATHNAME = "/usr/accounts/ledger "
 CALL !GET.PATHNAME(PATHNAME,DIR,FNAME,STATUS)
 IF STATUS = 0
 THEN
        PRINT "Directory portion = ":DIR
        PRINT "Entryname portion = ":FNAME
 END
```

!GETPU subroutine

Use the !GETPU subroutine to read individual parameters of any logical print channel.

Syntax

```
CALL !GETPU (key, print.channel, set.value, return.code)
```

key is a number indicating the parameter to be read.

print.channel is the logical print channel, designated by –1 through 255.

set.value is the value to which the parameter is currently set.

return.code is the code returned.

The !GETPU subroutine allows you to read individual parameters of logical print channels as designated by *print.channel.* Print channel 0 is the terminal unless a PRINTER ON statement has been executed to send output to the default printer. If you specify print channel –1, the output is directed to the terminal, regardless of the status of PRINTER ON or OFF. See the description of the !SETPU subroutine later in this appendix for a means of setting individual *print.channel* parameters.

Equate names for keys

An insert file of equate names is provided to allow you to use mnemonics rather than key numbers. The name of the insert file is GETPU.INS.IBAS. Use the $INCLUDE statement compiler directive to insert this file if you want to use equate names. The following list shows the equate names and keys for the parameters:

Mnemonic	Key	Parameter
PU$MODE	1	Printer mode.
PU$WIDTH	2	Device width (columns).
PU$LENGTH	3	Device length (lines).
PU$TOPMARGIN	4	Top margin (lines).
PU$BOTMARGIN	5	Bottom margin (lines).
PU$LEFTMARGIN	6	Left margin (columns, reset on printer close). Always returns 0.
PU$SPOOLFLAGS	7	Spool option flags.
PU$DEFERTIME	8	Spool defer time. This cannot be 0.
PU$FORM	9	Spool form (string).
PU$BANNER	10	Spool banner or hold filename (string).
PU$LOCATION	11	Spool location (string).
PU$COPIES	12	Spool copies. A single copy can be returned as 1 or 0.
PU$PAGING	14	Terminal paging (nonzero is on). This only works when PU$MODE is set to 1.
PU$PAGENUMBER	15	Returns the current page number.
PU$DISABLE	16	0 is returned if *print.channel* is enabled; and a 1 is returned if *print.channel* is disabled.
PU$CONNECT	17	Returns the number of a connected print channel or an empty string if no print channels are connected.
PU$NLSMAP	22	If NLS is enabled, returns the NLS map name associated with the specified print channel.
PU$LINESLEFT	1002	Lines left before new page needed. Returns erroneous values for the terminal if cursor addressing is used, if a line wider than the terminal is printed, or if terminal input has occurred.
PU$HEADERLINES	1003	Lines used by current header.
PU$FOOTERLINES	1004	Lines used by current footer.
PU$DATALINES	1005	Lines between current header and footer.

Mnemonic	Key	Parameter
PU$DATACOLUMNS	1006	Columns between left margin and device width.

The PU$SPOOLFLAGS key

The PU$SPOOLFLAGS key refers to a 32-bit option word that controls a number of print options. This is implemented as a 16-bit word and a 16-bit extension word. (Thus bit 21 refers to bit 5 of the extension word.) The bits are assigned as follows:

Bit	Description	
1	Uses FORTRAN-format mode. This allows the attaching of vertical format information to each line of the data file. The first character position of each line from the file does not appear in the printed output, and is interpreted as follows:	
	Character	**Meaning**
	0	Advances two lines.
	1	Ejects to the top of the next page.
	+	Overprints the last line.
	Space	Advances one line.
	–	Advances three lines (skip two lines). Any other character is interpreted as advance one line.
3	Generates line numbers at the left margin.	
4	Suppresses header page.	
5	Suppresses final page eject after printing.	
12	Spools the number of copies specified in an earlier !SETPU call.	
21	Places the job in the spool queue in the hold state.	
22	Retains jobs in the spool queue in the hold state after they have been printed.	
other	All the remaining bits are reserved.	

Equate names for return code

An insert file of equate names is provided to allow you to use mnemonics rather than key numbers. The name of the insert file is ERRD.INS.IBAS. Use the $INCLUDE statement to insert this file if you want to use equate names. The following list shows the codes returned in the argument *return.code*:

Code	Meaning
0	No error
E$BKEY	Bad key (*key* is out of range)
E$BPAR	Bad parameter (value of *new.value* is out of range)
E$BUNT	Bad unit number (value of *print.channel* is out of range)
E$NRIT	No write (attempt to set a read-only parameter)

Examples

In this example, the file containing the parameter key equate names is inserted with the $INCLUDE compiler directive. Later the top margin parameter for logical print channel 0 is interrogated. Print channel 0 is the terminal unless a prior PRINTER statement ON has been executed to direct output to the default printer. The top margin setting is returned in the argument TM.SETTING. Return codes are returned in the argument RETURN.CODE.

```
$INCLUDE UNIVERSE.INCLUDE GETPU.H
```

```
CALL !GETPU(PU$TOPMARGIN,0,TM.SETTING,RETURN.CODE)
```

The next example does the same as the previous example but uses the key 4 instead of the equate name PU$TOPMARGIN. Because the key number is used, it is not necessary for the insert file GETPU.H to be included.

```
CALL !GETPU(4,0,TM.SETTING,RETURN.CODE)
```

The next example returns the current deferred time on print channel 0 in the variable TIME.RET:

```
CALL !GETPU(PU$DEFERTIME,0,TIME.RET,RETURN.CODE)
```

!GET.USER.COUNTS subroutine

Use the !GET.USER.COUNTS subroutine to return a count of UniVerse and system users. If any value cannot be retrieved, a value of –1 is returned.

Syntax

```
CALL !GET.USER.COUNTS (uv.users, max.uv.users, os.users)
```

uv.users (output) is the current number of UniVerse users.

max.uv.users (output) is the maximum number of licensed UniVerse users allowed on your system.

os.users (output) is the current number of operating system users.

!GET.USERS subroutine

The !GET.USERS subroutine allows a BASIC program access to the system usage information.

Syntax

```
CALL !GET.USERS(uv.users,max.users,sys.users,user.info,code)
```

The *user.info* argument returns a dynamic array with a field for each user. Each field is separated by value marks into four values, containing the following information:

- The UniVerse user number
- The user ID
- The process ID
- The user type

The user type is a character string containing either Terminal or Phantom.

Example

The following example illustrates the use of the !GET.USERS subroutine.

```
0001:USERS = "!GET.USERS"
0002: CALL @USERS(UV.USERS,MAX.USERS,SYS.USERS,USER.INFO,CODE)
0003:CRT "UV.USERS = ":UV.USERS
0004:CRT "MAX.USERS = ":MAX.USERS
0005:CRT "SYS.USERS = ":SYS.USERS
0006:CRT "USER.INFO = ":USER.INFO
0007:CRT "CODE = ":CODE
```

```
0008:END
```

This program returns information similar to the following example:

```
UV.USERS = 1
MAX.USERS = 16
SYS.USERS = 1
USER.INFO = -916²NT AUTHORITY\system²916²Phantom¦1172²NORTHAMERICA\claireg²1172²
Terminal
CODE = 0
>ED &BP& TRY.GETUSERS
8 lines long.
```

!INLINE.PROMPTS subroutine

Use the !INLINE.PROMPTS subroutine to evaluate a string that contains in-line prompts.

Syntax

```
CALL !INLINE.PROMPTS (result , string )
```

In-line prompts have the following syntax:

<<{ *control* , }…*text* { , *option* }>>

result (output) is the variable that contains the result of the evaluation.

string (input) is the string containing an in-line prompt.

control specifies the characteristics of the prompt, and can be one of the following:

Prompt	Description
@(CLR)	Clears the terminal screen.
@(BELL)	Rings the terminal bell.
@(TOF)	Issues a formfeed character: in most circumstances this results in the cursor moving to the top left of the screen.
@ (*col* , *row*)	Prompts at the specified column and row number on the terminal.
A	Always prompts when the in-line prompt containing the control option is evaluated. If you do not specify this option, the input value from a previous execution of the prompt is used.
C*n*	Specifies that the *n*th word on the command line is used as the input value. (Word 1 is the verb in the sentence.)
F (*filename record.id* [,*fm* [,*vm* [,*sm*]]])	Takes the input value from the specified record in the specified file, and optionally, extracts a value (@VM), or subvalue (@SM), from the field (@FM). This option cannot be used with the file dictionary.
I*n*	Takes the *n*th word from the command line, but prompts if the word is not entered.
R (*string*)	Repeats the prompt until an empty string is entered. If *string* is specified, each response to the prompt is appended by *string* between each entry. If *string* is not specified, a space is used to separate the responses.

Prompt	Description
P	Saves the input from an in-line prompt. The input is then used for all in-line prompts with the same prompt text. This is done until the saved input is overwritten by a prompt with the same prompt text and with a control option of A, C, I, or S, or until control returns to the UniVerse prompt. The P option saves the input from an in-line prompt in the current paragraph, or in other paragraphs.
S*n*	Takes the *n*th word from the command (as in the I*n* control option), but uses the most recent command entered at the UniVerse system level to execute the paragraph, rather than an argument in the paragraph. This is useful where paragraphs are nested.
text	The prompt to be displayed.
option	A valid conversion code or pattern match. A valid conversion code is one that can be used with the ICONV function. Conversion codes must be enclosed in parentheses. A valid pattern match is one that can be used with the MATCHING keyword.

If the in-line prompt has a value, that value is substituted for the prompt. If the in-line prompt does not have a value, the prompt is displayed to request an input value when the function is executed. The value entered at the prompt is then substituted for the in-line prompt.

Note:

Once a value has been entered for a particular prompt, the prompt continues to have that value until a !CLEAR.PROMPTS subroutine is called, or control option A is specified. A !CLEAR.PROMPTS subroutine clears all the values that have been entered for in-line prompts.
You can enclose prompts within prompts.

Example

```
A = ""
 CALL !INLINE.PROMPTS(A,"You have requested the <<Filename>> file")
 PRINT "A"
```

The following output is displayed:

```
Filename=PERSONNEL
You have requested the PERSONNEL file
```

!INTS subroutine

Use the !INTS subroutine to retrieve the integer portion of elements in a dynamic array.

Syntax

```
CALL !INTS (result, dynamic.array)
```

result (output) contains a dynamic array that comprises the integer portions of the elements of *dynamic.array*.

dynamic.array (input) is the dynamic array to process.

The !INTS subroutine returns a dynamic array, each element of which contains the integer portion of the numeric value in the corresponding element of the input *dynamic.array.*

Example

```
A=33.0009:@VM:999.999:@FM:-4.66:@FM:88.3874
CALL !INTS(RESULT,A)
```

The following output is displayed:

```
33VM999FM-4FM88
```

!MAKE.PATHNAME subroutine

Use the !MAKE.PATHNAME subroutine to construct the full path name of a file.

Syntax

```
CALL !MAKE.PATHNAME (path1, path2, result, status)
```

The !MAKE.PATHNAME subroutine can be used to:

- Concatenate two strings to form a path name. The second string must be a relative path.
- Obtain the fully qualified path name of a file. Where only one of *path1* or *path2* is given, ! MAKE.PATHNAME returns the path name in its fully qualified state. In this case, any file name you specify does not have to be an existing file name.
- Return the current working directory. To do this, specify both *path1* and *path2* as empty strings.

path1 (input) is a file name or partial path name. If *path1* is an empty string, the current working directory is used.

path2 (input) is a relative path name. If *path2* is an empty string, the current working directory is used.

result (output) is the resulting path name.

status (output) is the returned status of the operation. 0 indicates success. Any other number indicates either of the following errors:

Status	Description
IE$NOTRELATIVE	*path2*was not a relative path name.
IE$PATHNOTFOUND	The path name could not be found when ! MAKE.PATHNAME tried to qualify it fully.

Example

In this example, the user's working directory is `/usr/accounts`:

```
ENT = "ledger"
 CALL !MAKE.PATHNAME(ENT,"",RESULT,STATUS)
 IF STATUS = 0
 THEN PRINT "Full name = ":RESULT
```

The following result is displayed:

```
Full name = /usr/accounts/ledger
```

!MATCHES subroutine

Use the !MATCHES subroutine to test whether each element of one dynamic array matches the patterns specified in the elements of the second dynamic array. Each element of *dynamic.array* is compared with the corresponding element of *match.pattern.* If the element in *dynamic.array* matches the pattern specified in *match.pattern*, 1 is returned in the corresponding element of *result*. If the element from *dynamic.array* is not matched by the specified pattern, 0 is returned.

Syntax

```
CALL !MATCHES (result , dynamic. array , match.pattern )
```

result (output) is a dynamic array containing the result of the comparison on each element in *dynamic array1*.

dynamic.array (input) is the dynamic array to be tested.

match.pattern (input) is a dynamic array containing the match patterns.

When *dynamic.array* and *match.pattern* do not contain the same number of elements, the behavior of !MATCHES is as follows:

- *result* always contains the same number of elements as the longer of *dynamic.array* or *match.pattern*.
- If there are more elements in *dynamic.array* than in *match.pattern*, the missing elements are treated as though they contained a pattern that matched an empty string.
- If there are more elements in *match.pattern* than in *dynamic.array*, the missing elements are treated as though they contained an empty string.

Examples

The following example returns the value of the dynamic array as `1VM1VM1`:

```
A='AAA4A4':@VM:2398:@VM:'TRAIN'
 B='6X':@VM:'4N':@VM:'5A'
 CALL !MATCHES(RESULT,A,B)
```

In the next example, there are missing elements in *match.pattern* that are treated as though they contain a pattern that matches an empty string. The result is `0VM0SM0FM1FM1`.

```
 R='AAA':@VM:222:@SM:'CCCC':@FM:33:@FM:'DDDDDD'
 S='4A':@FM:'2N':@FM:'6X'
 CALL !MATCHES(RESULT,R,S)
```

In the next example, the missing element in *match.pattern* is used as a test for an empty string in dynamic.array, and the result is `1VM1FM1`:

```
X='AAA':@VM:@FM:''
 Y='3A':@FM:'3A'
 CALL !MATCHES(RESULT,X,Y)
```

!MESSAGE subroutine

Use the !MESSAGE subroutine to send a message to another user on the system. !MESSAGE lets you change and report on the current user's message status.

Syntax

```
CALL !MESSAGE (key, username, usernum, message, status)
```

key (input) specifies the operation to be performed. You specify the option you require with the *key* argument, as follows:

Key	Description
IK$MSGACCEPT	Sets message status to accept.
IK$MSGREJECT	Sets message status to reject.
IK$MSGSEND	Sends message to user.
IK$MSGSENDNOW	Sends message to user now.
IK$MSGSTATUS	Displays message status of user.

username (input) is the name of the user, or the TTY name, for send or status operations.

usernum (input) is the number of the user for send/status operations.

message (input) is the message to be sent.

status (output) is the returned status of the operation as follows:

Status	Description
0	The operation was successful.
IE$NOSUPPORT	You specified an unsupported *key* option.
IE$KEY	You specified an invalid *key* option.
IE$PAR	The *username* or *message* you specified was not valid.
IE$UNKNOWN.USER	You tried to send a message to a user who is not logged in to the system.
IE$SEND.REQ.REC	The sender does not have the MESSAGERECEIVE option enabled.
IE$MSG.REJECTED	One or more users have the MESSAGEREJECT mode set.

Note: The value of message is ignored when key is set to IK$MSGACCEPT, IK$MSGREJECT, or IK$MSGSTATUS.

Example

```
CALL !MESSAGE (KEY,USERNAME,USERNUMBER,MESSAGE,CODE)
 IF CODE # 0
 THEN CALL !REPORT.ERROR ('MY.COMMAND','!MESSAGE',CODE)
```

!PACK.FNKEYS subroutine

The !PACK.FNKEYS subroutine converts a list of function key numbers into a bit string suitable for use with the !EDIT.INPUT subroutine. This bit string defines the keys which cause !EDIT.INPUT to exit, enabling the program to handle the specific keys itself.

Syntax

```
CALL !PACK.FNKEYS (trap.list, ftable)
```

Qualifiers

Qualifier	Description
trap.list	A list of function numbers delimited by field marks (CHAR(254)), defining the specific keys that are to be used as trap keys by the !EDIT.INPUT subroutine.
ftable	A bit-significant string of trap keys used in the *ftable* parameter of the !EDIT.INPUT subroutine. This string should not be changed in any way before calling the !EDIT.INPUT subroutine.

trap.list can be a list of function key numbers delimited by field marks (CHAR(254)). Alternatively, the mnemonic key name, listed below and in the UNIVERSE.INCLUDE file GTI.FNKEYS.IH, can be used:

Key name	Field	Description
FK$FIN	1	Finish
FK$HELP	2	Help
FK$BSP	3	Backspace[a]
FK$LEFT	4	Left arrow[a]
FK$RIGHT	5	Right arrow[a]
FK$UP	6	Up arrow
FK$DOWN	7	Down arrow
FK$LSCR	8	Left screen
FK$RSCR	9	Right screen
FK$USCR	10	Up screen, Previous page
FK$DSCR	11	Down screen, Next page
FK$BEGEND	12	Toggle begin/end line, or Begin line
FK$TOPBOT	13	Top/Bottom, or End line
FK$NEXTWD	14	Next word
FK$PREVWD	15	Previous word
FK$TAB	16	Tab
FK$BTAB	17	Backtab
FK$CTAB	18	Column tab
FK$INSCH	19	Insert character (space)[a]
FK$INSLIN	20	Insert line
FK$INSTXT	21	Insert text, Toggle insert/overlay mode[a]
FK$INSDOC	22	Insert document
FK$DELCH	23	Delete character[a]
FK$DELLIN	24	Delete line[a]
FK$DELTXT	25	Delete text
FK$SRCHNX	26	Search next
FK$SEARCH	27	Search
FK$REPLACE	28	Replace
FK$MOVE	29	Move text
FK$COPY	30	Copy text
FK$SAVE	31	Save text
FK$FMT	32	Call format line
FK$CONFMT	33	Confirm format line

Key name	Field	Description
FK$CONFMTNW	34	Confirm format line, no wrap
FK$OOPS	35	Oops
FK$GOTO	36	Goto
FK$CALC	37	Recalculate
FK$INDENT	38	Indent (set left margin)
FK$MARK	39	Mark
FK$ATT	40	Set attribute
FK$CENTER	41	Center
FK$HYPH	42	Hyphenate
FK$REPAGE	43	Repaginate
FK$ABBREV	44	Abbreviation
FK$SPELL	45	Check spelling
FK$FORM	46	Enter formula
FK$HOME	47	Home the cursor
FK$CMD	48	Enter command
FK$EDIT	49	Edit
FK$CANCEL	50	Abort/Cancel
FK$CLEOL	51	Clear to end of line[a]
FK$SCRWID	52	Toggle between 80 and 132 mode
FK$PERF	53	Invoke DSS PERFORM emulator
FK$INCLUDE	54	DSS Include scratchpad data
FK$EXPORT	55	DSS Export scratchpad data
FK$TWIDDLE	56	Twiddle character pair
FK$DELWD	57	Delete word
FK$SRCHPREV	58	Search previous
FK$LANGUAGE	59	Language
FK$REFRESH	60	Refresh
FK$UPPER	61	Uppercase
FK$LOWER	62	Lowercase
FK$CAPIT	63	Capitalize
FK$REPEAT	64	Repeat
FK$STAMP	65	Stamp
FK$SPOOL	66	Spool record
FK$GET	67	Get record
FK$WRITE	68	Write record
FK$EXECUTE	69	Execute macro
FK$NUMBER	70	Toggle line numbering
FK$DTAB	71	Clear tabs
FK$STOP	72	Stop (current activity)
FK$EXCHANGE	73	Exchange mark and cursor
FK$BOTTOM	74	Move bottom
FK$CASE	75	Toggle case sensitivity
FK$LISTB	76	List (buffers)

Key name	Field	Description
FK$LISTD	77	List (deletions)
FK$LISTA	78	List (selects)
FK$LISTC	79	List (commands)
FK$DISPLAY	80	Display (current select list)
FK$BLOCK	81	Block (replace)
FK$PREFIX	82	Prefix

Indicates supported functionality.

If *ftable* is returned as an empty string, an error in the *trap.list* array is detected, such as an invalid function number. Otherwise *ftable* is a bit-significant string which should not be changed in any way before its use with the !EDIT.INPUT subroutine subroutine.

Example

The following program sets up three trap keys using the !PACK.FNKEYS function, then uses the bit string within the !EDIT.INPUT subroutine:

```
$INCLUDE UNIVERSE.INCLUDE GTI.FNKEYS.IH
 * Set up trap keys of FINISH, UPCURSOR and DOWNCURSOR
 TRAP.LIST = FK$FIN:@FM:FK$UP:@FM:FK$DOWN
 CALL !PACK.FNKEYS(TRAP.LIST, Ftable)
 * Start editing in INPUT mode, displaying contents in window
 KEYS = IK$INS + IK$DIS
 * Window edit is at x=20, y=2, of length 10 characters;
 * the user can enter up to 30 characters of input into TextBuffer,
 * and the cursor is initially placed on the first character of the
 * window.
 TextBuffer=""
 CursorPos = 1
 CALL !EDIT.INPUT(KEYS,20,2,10,TextBuffer,CursorPos,30,Ftable,ReturnCode)
  * On exit, the user's input is within TextBuffer,
 * CursorPos indicates the location of the cursor upon exiting,
 * and ReturnCode contains the reason for exiting.
 BEGIN CASE
           CASE CODE = 0
                                * User pressed RETURN key
       CASE CODE = FK$FIN
                                * User pressed the defined FINISH key
       CASE CODE = FK$UP
                                * User pressed the defined UPCURSOR key
       CASE CODE = FK$DOWN
                                * User pressed the defined DOWNCURSOR key
       CASE 1             * Should never happen
 END CASE
```

!REPORT.ERROR subroutine

Use the !REPORT.ERROR subroutine to print explanatory text for a UniVerse or operating system error code.

Syntax

```
CALL !REPORT.ERROR (command, subroutine, code)
```

command is the name of the command that used the subroutine in which an error was reported.

subroutine is the name of the subroutine that returned the error code.

code is the error code.

The general format of the message printed by !REPORT.ERROR is as follows:

```
Error: Calling subroutine from command. system error code:
message.text.
```

system is the operating system, or UniVerse.

Text for values of *code* in the range 0 through 9999 is retrieved from the operating system. Text for values of *code* over 10,000 is retrieved from the SYS.MESSAGES file. If the code has no associated text, a message to that effect is displayed. Some UniVerse error messages allow text to be inserted in them. In this case, *code* can be a dynamic array of the error number, followed by one or more parameters to be inserted into the message text.

Examples

```
CALL !MESSAGE (KEY,USERNAME,USERNUMBER,MESSAGE,CODE)
 IF CODE # 0
 THEN CALL !REPORT.ERROR ('MY.COMMAND','!MESSAGE',CODE)
```

If *code* was IE$SEND.REQ.REC, !REPORT.ERROR would display the following:

```
Error calling "!MESSAGE" from "MY.COMMAND" UniVerse error 1914:
Warning: Sender requires "receive" enabled!
```

The next example shows an error message with additional text:

```
CALL !MESSAGE (KEY,USERNAME,USERNUMBER,MESSAGE,CODE)
 IF CODE # 0
 THEN CALL !REPORT.ERROR ('MY.COMMAND','!MESSAGE',CODE:@FM:USERNAME)
```

If *code* was IE$UNKNOWN.USER, and the user ID was joanna, !REPORT.ERROR would display the following:

```
Error calling "!MESSAGE" from "MY.COMMAND" UniVerse error 1757: joanna
is not logged on
```

!SET.PTR subroutine

Use the !SET.PTR subroutine to set options for a logical print channel. This subroutine provides the same functionality as the UniVerse SETPTR (UNIX) or SETPTR (Windows platforms) command.

Syntax

```
CALL !SET.PTR (print.channel, width, length, top.margin,
bottom.margin, mode, options)
```

print.channel is the logical printer number, –1 through 255. The default is 0.

width is the page width. The default is 132.

length is the page length. The default is 66.

top.margin is the number of lines left at the top of the page. The default is 3.

bottom.margin is the number of lines left at the bottom of the page. The default is 3.

mode is a number 1 through 5 that indicates the output medium, as follows:

- 1 - Line Printer Spooler Output (default).
- 2, 4, 5 - Assigned Device. To send output to an assigned device, you must first assign the device to a logical print channel, using the UniVerse `ASSIGN` command. The `ASSIGN` command issues an automatic `SETPTR` command using the default parameters, except for mode, which it sets to 2. Use !SET.PTR only if you have to change the default parameters.
- 3 - Hold File Output. Mode 3 directs all printer output to a file called &HOLD&. If a &HOLD& file does not exist in your account, !SET.PTR creates the file and its dictionary (D_&HOLD&). You must execute !SET.PTR with mode 3 before each report to create unique report names in &HOLD&. If the report exists with the same name, the new report overwrites.

options are any of the printer options that are valid for the `SETPTR` command. These must be separated by commas and enclosed by valid quotation marks.

If you want to leave a characteristic unchanged, supply an empty string argument and specify the option NODEFAULT. If you want the default to be selected, supply an empty string argument without specifying the NODEFAULT option.

Printing on the last line and printing a heading

If you print on the last line of the page or screen and use a HEADING statement to print a heading, your printout will have blank pages. The printer or terminal is set to advance to the top of the next page when the last line of the page or screen is printed. The HEADING statement is set to advance to the top of the next page to print the heading.

Example

The following example sets the options so that printing is deferred until 12:00, and the job is retained in the queue:

```
CALL !SET.PTR (0,80,60,3,3,1,'DEFER 12:00,RETAIN')
```

!SETPU subroutine

Use the !SETPU subroutine to set individual parameters of any logical print channel.

Syntax

```
CALL !SETPU (key, print.channel, new.value, return.code)
```

Unlike !SET.PTR subroutine, you can specify only individual parameters to change; you need not specify parameters you do not want to change. See the description of the !GETPU subroutine subroutine for a way to read individual *print.channel* parameters.

key is a number indicating the parameter to be set (see !GETPU subroutine, on page 566).

print.channel is the logical print channel, designated by –1 through 255.

new.value is the value to which you want to set the parameter.

return.code is the returned error code (see !GETPU subroutine, on page 566).

The !SETPU subroutine lets you change individual parameters of logical print channels as designated by *print.channel.* Print channel 0 is the terminal unless a PRINTER statement ON has been executed to send output to the default printer. If you specify print channel –1, the output is directed to the terminal, regardless of the status of PRINTER ON or OFF.

Equate names for keys

An insert file of equate names is provided to allow you to use mnemonics rather than key numbers. The name of the insert file is GETPU.INS.IBAS. Use the $INCLUDE compiler directive to insert this file if you want to use the equate names. For a description of the $INCLUDE statement compiler directive, see *UniVerse BASIC*. The following list shows the equate names and keys for the parameters:

Mnemonic	Key	Parameter
PU$MODE	1	Printer mode.
PU$WIDTH	2	Device width (columns).
PU$LENGTH	3	Device length (lines).
PU$TOPMARGIN	4	Top margin (lines).
PU$BOTMARGIN	5	Bottom margin (lines).
PU$SPOOLFLAGS	7	Spool option flags (see !GETPU subroutine, on page 566).
PU$DEFERTIME	8	Spool defer time. This cannot be 0.
PU$FORM	9	Spool form (string).
PU$BANNER	10	Spool banner or hold file name (string).
PU$LOCATION	11	Spool location (string).
PU$COPIES	12	Spool copies. A single copy can be returned as 1 or 0.
PU$PAGING	14	Terminal paging (nonzero is on). This only works when PU $MODE is set to 1.
PU$PAGENUMBER	15	Sets the next page number.

The PU$SPOOLFLAGS key

The PU$SPOOLFLAGS key refers to a 32-bit option word that controls a number of print options. This is implemented as a 16-bit word and a 16-bit extension word. (Thus bit 21 refers to bit 5 of the extension word.) The bits are assigned as follows:

Bit	Description	
1	Uses FORTRAN-format mode. This allows the attaching of vertical format information to each line of the data file. The first character position of each line from the file does not appear in the printed output, and is interpreted as follows:	
	Character	**Meaning**
	0	Advances two lines.
	1	Ejects to the top of the next page.
	+	Overprints the last line.
	Space	Advances one line.
	–	Advances three lines (skip two lines). Any other character is interpreted as advance one line.
3	Generates line numbers at the left margin.	
4	Suppresses header page.	
5	Suppresses final page eject after printing.	
12	Spools the number of copies specified in an earlier !SETPU call.	
21	Places the job in the spool queue in the hold state.	
22	Retains jobs in the spool queue in the hold state after they have been printed.	
other	All the remaining bits are reserved.	

Equate names for return code

An insert file of equate names is provided to allow you to use mnemonics rather than key numbers. The name of the insert file is ERRD.INS.IBAS. Use the $INCLUDE statement to insert this file if you want to use equate names. The following list shows the codes returned in the argument *return.code*:

Code	Meaning
0	No error
E$BKEY	Bad key (*key* is out of range)
E$BPAR	Bad parameter (value of *new.value* is out of range)
E$BUNT	Bad unit number (value of *print.channel* is out of range)
E$NRIT	No write (attempt to set a read-only parameter)

Printing on the last line and printing a heading

If you print on the last line of the page or screen and use a HEADING statement to print a heading, your printout will have blank pages. The printer or terminal is set to advance to the top of the next page or screen when the last line of the page or screen is printed. The HEADING statement is set to advance to the top of the next page to print the heading.

Examples

In the following example, the file containing the parameter key equate names is inserted with the $INCLUDE compiler directive. Later, the top margin parameter for logical print channel 0 is set to 10 lines. Return codes are returned in the argument RETURN.CODE.

```
$INCLUDE SYSCOM GETPU.INS.IBAS
 CALL !SETPU(PU$TOPMARGIN,0,10,RETURN.CODE)
```

The next example does the same as the previous example, but uses the key 4 instead of the equate name PU$TOPMARGIN. Because the key is used, it is not necessary for the insert file GETPU.INS.IBAS to be included.

```
CALL !SETPU(4,0,10,RETURN.CODE)
```

!TIMDAT subroutine

Use the !TIMDAT subroutine to return a dynamic array containing the time, date, and other related information. The !TIMDAT subroutine returns a 13-element dynamic array containing information shown in the following list.

Syntax

```
CALL !TIMDAT (variable)
```

variable is the name of the variable to which the dynamic array is to be assigned.

Field	Description
1	Month (two digits).
2	Day of month (two digits).
3	Year (two digits).
4	Minutes since midnight (integer).
5	Seconds into the minute (integer).

Field	Description
6	Ticks of last second since midnight (integer). Always returns 0. Tick refers to the unit of time your system uses to measure real time.
7	CPU seconds used since entering UniVerse.
8	Ticks of last second used since login (integer).
9	Disk I/O seconds used since entering UniVerse. Always returns –1.
10	Ticks of last disk I/O second used since login (integer). Always returns –1.
11	Number of ticks per second.
12	User number.
13	Login ID (user ID).

Use the following functions for alternative ways of obtaining time and date information:

Use this function...	To obtain this data...
DATE function	Data in fields 1, 2, and 3 of the dynamic array returned by the !TIMDAT subroutine
TIME function	Data in fields 4, 5, and 6 of the dynamic array returned by the !TIMDAT subroutine
@USERNO	User number
@LOGNAME	Login ID (user ID)

Example

```
CALL !TIMDAT(DYNARRAY)
 FOR X = 1 TO 13
    PRINT 'ELEMENT ':X:', DYNARRAY
 NEXT X
```

!USER.TYPE subroutine

Use the !USER.TYPE subroutine to return the user type of the current process and a flag to indicate if the user is a UniVerse Administrator.

Syntax

```
CALL !USER.TYPE (type, admin)
```

type is a value that indicates the type of process making the subroutine call. *type* can be either of the following:

Equate Name	Value	Meaning
U$NORM	1	Normal user
U$PH	65	Phantom

admin is a value that indicates if the user making the call is a UniVerse Administrator. Possible values of *admin* are 1, if the user is a UniVerse Administrator, and 0, if the user is not a UniVerse Administrator.

An insert file of equate names is provided for the !USER.TYPE values. To use the equate names, specify the directive $INCLUDE statement SYSCOM USER_TYPES.H when you compile your program. (For PI/open compatibility you can specify $INCLUDE SYSCOM USER_TYPES.INS.IBAS.)

Example

In this example, the !USER.TYPE subroutine is called to determine the type of user. If the user is a phantom, the program stops. If the user is not a phantom, the program sends a message to the terminal and continues processing.

```
ERROR.ACCOUNTS.FILE: CALL !USER.TYPE(TYPE, ADMIN)
 IF TYPE = U&PH THEN STOP
        ELSE PRINT 'Error on opening ACCOUNTS file'
```

!VOC.PATHNAME subroutine

Use the !VOC.PATHNAME subroutine to extract the path names for the data file or the file dictionary of a specified VOC entry.

Syntax

```
CALL !VOC.PATHNAME (data/dict, voc.entry, result, status)
```

data/dict (input) indicates the file dictionary or data file, as follows:

- IK$DICT or 'DICT' returns the path name of the file dictionary of the specified VOC entry.
- IK$DATA or ' ' returns the path name (or path names for distributed files) of the data file of the specified VOC entry.

voc.entry is the record ID in the VOC.

result (output) is the resulting path names.

status (output) is the returned status of the operation.

An insert file of equate names is provided for the *data/dict* values. To use the equate names, specify the directive $INCLUDE statement SYSCOM INFO_KEYS.H when you compile your program. (For PI/open compatibility you can specify $INCLUDE SYSCOM INFO_KEYS.INS.IBAS.)

The result of the operation is returned in the *status* argument, and has one of the following values:

Value	Result
0	The operation executed successfully.
IE$PAR	A bad parameter was used in *data/dict* or *voc.entry*.
IE$RNF	The VOC entry record cannot be found.

Example

```
CALL !VOC.PATHNAME (IK$DATA,"VOC",VOC.PATH,STATUS)
 IF STATUS = 0
 THEN PRINT "VOC PATHNAME = ":VOC.PATH
```

If the user's current working directory is /usr/account, the output is:

```
VOC PATHNAME =  /usr/accounts/VOC
```

Appendix G: Socket function error return codes

The following error return codes are used for all socket-related functions described below. Note that only numeric code should be used in UniVerse BASIC programs.

The following table describes each error code and its meaning.

Error code	Definition
0 -SCK_ENOERROR	No error.
1 - SCK_ENOINITIALISED	On Windows platforms, a successful WSAStartup() call must occur before using this function.
2 - SCK_ENETDOWN	The network subsystem has failed.
3 - SCK_EFAULT	The *addrlen* parameter is too small or *addr* is not a valid part of the user address space.
4 - SCK_ENOTCONN	The socket is not connected.
5 - SCK_EINTR	The (blocking) call was canceled. (through WSACancelBlocking-Call).
6 - SCK_EINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
7 - SCK_EINVAL	This can be caused by several conditions. The listen function was not invoked prior to accept, the socket has not been bound with bind, an unknown flag was specified, or MSG_OOB was specified for a socket with SO_OOBINLINE enabled or (for byte stream sockets only) len was zero or negative.
8 - SCK_EMFILE	The queue is nonempty upon entry to accept and there are no descriptors available.
9 - SCK_ENOBUFS	No buffer space is available.
10 - SCK_ENOTSOCK	The descriptor is not a socket.
11 - SCK_EOPNOTSUPP	The referenced socket is not a type that supports connection-oriented service.
12 - SCK_EWOULDBLOCK	The socket is marked as nonblocking and the requested operation would block.
13 - SCK_ENETRESET	The connection has been broken due to the keep-alive activity detecting a failure while the operation was in progress.
14 - SCK_ESHUTDOWN	The socket has been shut down.
15 - SCK_EMSGSIZE	(For recv()) The message was too large to fit into the specified buffer and was truncated, or (for send()) the socket is message oriented, and the message is larger than the maximum supported by the underlying transport.
16 - SCK_ETIMEDOUT	The virtual circuit was terminated due to a time-out or other failure. The application should close the socket as it is no longer usable.
17 - SCK_ECONNABORTED	The connection has been dropped, because of a network failure or because the system on the other end went down without notice.

Error code	Definition
18 - SCK_ECONNRESET	The virtual circuit was reset by the remote side executing a hard or abortive close. For UPD sockets, the remote host was unable to deliver a previously sent UDP datagram and responded with a “Port Unreachable” ICMP packet. The application should close the socket as it is no longer usable.
19 - SCK_EACCES	The requested address is a broadcast address, but the appropriate flag was not set. Call `setSocketOption`() with the BROADCAST parameter to allow the use of the broadcast address.
20 - SCK_EHOSTUNREACH	The remote host cannot be reached from this host at this time.
21 - SCK_ENOPROTOOPT	The option is unknown or unsupported for the specified provider or socket.
22 - SCK_ESYSNOTREADY	Indicates that the underlying network subsystem is not ready for network communication.
23 -SCK_EVER NOTSUPPORTED	The version of Windows Sockets support requested is not provided by this particular Windows Sockets implementation.
24 - SCK_EPROCLIM	Limit on the number of tasks supported by the Windows Sockets implementation has been reached.
25 - SCK_EAFNOSUPPORT	The specified address family is not supported.
26 - SCK_EPROTONOSUPPORT	The specified protocol is not supported.
27 - SCK_EPROTOTYPE	The specified protocol is the wrong type for this socket.
28 - SCK_ESOCKTNOSUPPORT	The specified socket type is not supported in this address family.
29 - SCK_EBADF	Descriptor socket is not valid.
30 - SCK_EHOST_NOT_FOUND	Authoritative Answer Host not found.
31 - SCK_ETRY_AGAIN	Nonauthoritative Host not found, or server failure.
32 - SCK_ENO_RECOVERY	A nonrecoverable error occurred.
33 - SCK_ENO_DATA	Valid name, no data record of requested type.
34 - SCK_EACCESS	Attempt to connect datagram socket to broadcast address failed because setSocketOption() BROADCAST is not enabled.
35 - SCK_EADDRINUSE	A process on the machine is already bound to the same fully-qualified address and the socket has not been marked to allow address reuse with REUSEADDR. (See the REUSEADDR socket option under `setSocketOption`()).
36 - SCK_EADDRNOTAVAIL	The specified address is not a valid address for this machine.
37 - SCK_EISCONN	The socket is already connected.
38 - SCK_EALREADY	A nonblocking connect call is in progress on the specified socket.
39 - SCK_ECONNREFUSED	The attempt to connect was forcefully rejected.
40 - SCK_EMALLOC	Memory allocation error.
41 - SCK_ENSLMAP	NLS map not found, or unmapped characters encountered.
42 - SCK_EUNKNOWN	Other unknown errors.
101	Invalid security context handle.
102	SSL/TLS handshake failure (unspecified, peer is not SSL aware).
103	Requires client authentication but does not have a certificate in context.

Error code	Definition
104	Unable to authenticate server.
105	Client authentication failure.
106	Peer not speaking SSL.
107	Encryption error.
108	Decryption error.