

# UniVerse

**BASIC**

**Part No. 70-9002-952**

**Ardent<sup>™</sup>**  
Software, Inc.

50 WASHINGTON ST.  
WESTBORD, MA 01581  
USA

# NOTICE

Ardent Software, Inc., makes no warranty of any kind with regard to the material contained in this manual, including but not limited to the implied warranties of merchantability and fitness for a particular purpose.

The information contained in this manual is subject to change without notice.

This manual contains proprietary information that is protected by copyright. All rights are reserved. It may not be photocopied, reproduced, or translated, in whole or in part, without the prior express written consent of Ardent Software, Inc.

Copyright © 1988–2000 Ardent Software, Inc. All rights reserved.

## Trademarks

UniVerse is a registered trademark of Ardent Software, Inc. Uni Call Interface, UniVerse Data Replication, UniVerse NLS, UniVerse ODBC, UniObjects, UV/Net, UV/Term, and Ardent are trademarks of Ardent Software, Inc.

Microsoft, Windows, and Windows NT are registered trademarks of Microsoft Corporation. Open Database Connectivity is a trademark of Microsoft Corporation.

UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company, Ltd.

All other company or product names mentioned are trademarks or registered trademarks of their respective trademark holders.

## Printing History

First Printing (70-9002-952) for Release 9.5.2, February 2000

## How to Order Technical Documents

To order copies of documents or to obtain a catalog and price list, contact your local Ardent subsidiary or distributor, or call our main office at (508) 366-3888.

## Customer Comments

We welcome your input. Please comment on this manual using the customer comment [form](#) provided in the back of the manual.

This manual is printed on recycled paper.



# UniVerse BASIC

## Table of Contents

### **Preface**

Organization of This Manual .....	xv
Documentation Conventions .....	xvi
UniVerse Documentation .....	xvii
Related Documentation .....	xix
Common APIs Documentation .....	xx

### **Chapter 1. Introduction to UniVerse BASIC**

BASIC Terminology .....	1-2
Subroutines .....	1-3
Source Syntax .....	1-3
Statement Types .....	1-4
Statement Labels .....	1-5
Spaces or Tabs .....	1-5
Newlines and Sequential File I/O .....	1-5
Special Characters .....	1-6
Storing Programs .....	1-7
Editing Programs .....	1-7
Editing Programs in UniVerse .....	1-7
Editing Programs Outside UniVerse .....	1-8
Getting Started .....	1-8

### **Chapter 2. Data Types, Variables, and Operators**

Types of Data .....	2-1
Character String Data .....	2-1
Numeric Data .....	2-2
Unknown Data: The Null Value .....	2-3

Constants .....	2-4
Variables .....	2-4
Array Variables .....	2-5
File Variables .....	2-9
Select List Variables .....	2-9
Expressions .....	2-9
Format Expressions .....	2-10
Operators .....	2-11
Arithmetic Operators .....	2-11
String Operators .....	2-13
Relational Operators .....	2-15
Pattern Matching Operators .....	2-16
IF Operator .....	2-17
Logical Operators .....	2-17
Assignment Operators .....	2-19
Dynamic Array Operations .....	2-19

### **Chapter 3. Compiling BASIC Programs**

The BASIC Command .....	3-1
Compiling Programs in the Background .....	3-1
BASIC Options .....	3-2
Compiler Directives .....	3-4
Including Other Programs .....	3-5
Defining and Removing Identifiers .....	3-6
Specifying Flavor Compatibility .....	3-6
Conditional Compilation .....	3-6
Warnings and Error Messages .....	3-9
Successful Compilation .....	3-9
The RUN Command .....	3-10
Cataloging a BASIC Program .....	3-10
Catalog Space .....	3-11
The CATALOG Command .....	3-12
Deleting Cataloged Programs .....	3-12
Catalog Shared Memory .....	3-13

## Chapter 4. Locks, Transactions, and Isolation Levels

Locks .....	4-1
Shared Record Lock .....	4-2
Update Record Lock .....	4-3
Shared File Lock .....	4-4
Intent File Lock .....	4-4
Exclusive File Lock .....	4-5
Deadlocks .....	4-5
Transactions .....	4-5
Active Transactions .....	4-6
Transactions and Data Visibility .....	4-6
Transaction Properties .....	4-7
Transactions and Locks .....	4-8
Transactions and Isolation Levels .....	4-8
Using Transactions in BASIC .....	4-9
@Variables .....	4-10
Transaction Restrictions .....	4-10
Isolation Levels .....	4-11
Isolation Level Types .....	4-11
Data Anomalies .....	4-12
Using the ISOMODE Configurable Parameter .....	4-13
Isolation Levels and Locks .....	4-14
Example .....	4-15

## Chapter 5. Debugging Tools

RAID .....	5-1
Invoking RAID from the Command Processor .....	5-2
Invoking RAID from a BASIC Program .....	5-3
Invoking RAID Using the Break Key .....	5-3
Referencing Variables Through RAID .....	5-3
RAID Commands .....	5-4
VLIST .....	5-12

## Chapter 6. BASIC Statements and Functions

! statement .....	6-2
#INCLUDE statement .....	6-3
\$* statement .....	6-4
\$CHAIN statement .....	6-5
\$COPYRIGHT statement .....	6-6
\$DEFINE statement .....	6-7
\$EJECT statement .....	6-9
\$IFDEF statement .....	6-10
\$IFNDEF statement .....	6-11
\$INCLUDE statement .....	6-12
\$INSERT statement .....	6-13
\$MAP statement .....	6-15
\$OPTIONS statement .....	6-16
\$PAGE statement .....	6-25
\$UNDEFINE statement .....	6-26
* statement .....	6-27
< > operator .....	6-28
@ function .....	6-29
[ ] operator .....	6-49
ABORT statement .....	6-52
ABS function .....	6-53
ABSS function .....	6-54
ACOS function .....	6-55
ADDS function .....	6-56
ALPHA function .....	6-57
ANDS function .....	6-58
ASCII function .....	6-59
ASIN function .....	6-60
ASSIGNED function .....	6-61
assignment statements .....	6-62
ATAN function .....	6-64
AUTHORIZATION statement .....	6-65

AUXMAP statement .....	6-67
BEGIN CASE statement .....	6-68
BEGIN TRANSACTION statement .....	6-69
BITAND function .....	6-70
BITNOT function .....	6-71
BITOR function .....	6-72
BITRESET function .....	6-73
BITSET function .....	6-74
BITTEST function .....	6-75
BITXOR function .....	6-76
BREAK statement .....	6-77
BSCAN statement .....	6-79
BYTE function .....	6-82
BYTELEN function .....	6-83
BYTETYPE function .....	6-84
BYTEVAL function .....	6-85
CALL statement .....	6-86
CASE statement .....	6-89
CATS function .....	6-92
CHAIN statement .....	6-93
CHANGE function .....	6-94
CHAR function .....	6-95
CHARS function .....	6-96
CHECKSUM function .....	6-97
CLEAR statement .....	6-98
CLEARDATA statement .....	6-99
CLEARFILE statement .....	6-100
CLEARPROMPTS statement .....	6-102
CLEARSELECT statement .....	6-103
CLOSE statement .....	6-105
CLOSESEQ statement .....	6-107
COL1 function .....	6-109
COL2 function .....	6-110

COMMIT statement .....	6-112
COMMON statement .....	6-114
COMPARE function .....	6-115
CONTINUE statement .....	6-117
CONVERT function .....	6-118
CONVERT statement .....	6-119
COS function .....	6-120
COSH function .....	6-121
COUNT function .....	6-122
COUNTS function .....	6-124
CREATE statement .....	6-126
CRT statement .....	6-127
DATA statement .....	6-128
DATE function .....	6-130
DCOUNT function .....	6-131
DEBUG statement .....	6-132
DEFFUN statement .....	6-134
DEL statement .....	6-136
DELETE function .....	6-138
DELETE statements .....	6-140
DELETelist statement .....	6-144
DELETEU statement .....	6-145
DIMENSION statement .....	6-146
DISPLAY statement .....	6-149
DIV function .....	6-150
DIVS function .....	6-151
DOWNCASE function .....	6-152
DQUOTE function .....	6-153
DTX function .....	6-154
EBCDIC function .....	6-155
ECHO statement .....	6-156
END statement .....	6-157
END CASE statement .....	6-158



END TRANSACTION statement .....	6-159
ENTER statement .....	6-160
EOF(ARG.) function .....	6-161
EQS function .....	6-162
EQUATE statement .....	6-163
EREPLACE function .....	6-165
ERRMSG statement .....	6-166
EXCHANGE function .....	6-168
EXECUTE statement .....	6-170
EXIT statement .....	6-173
EXP function .....	6-174
EXTRACT function .....	6-175
FADD function .....	6-177
FDIV function .....	6-178
FFIX function .....	6-179
FFLT function .....	6-180
FIELD function .....	6-181
FIELDS function .....	6-183
FIELDSTORE function .....	6-184
FILEINFO function .....	6-186
FILELOCK statement .....	6-192
FILEUNLOCK statement .....	6-195
FIND statement .....	6-197
FINDSTR statement .....	6-198
FIX function .....	6-199
FLUSH statement .....	6-200
FMT function .....	6-201
FMTDP function .....	6-206
FMTS function .....	6-207
FMTSDP function .....	6-208
FMUL function .....	6-209
FOLD function .....	6-210
FOLDDP function .....	6-211

FOOTING statement .....	6-212
FOR statement .....	6-216
FORMLIST statement .....	6-220
FSUB function .....	6-221
FUNCTION statement .....	6-222
GES function .....	6-224
GET statements .....	6-225
GETX statement .....	6-230
GET(ARG.) statement .....	6-231
GETLIST statement .....	6-233
GETLOCALE function .....	6-234
GETREM function .....	6-235
GOSUB statement .....	6-236
GOTO statement .....	6-238
GROUP function .....	6-239
GROUPSTORE statement .....	6-241
GTS function .....	6-243
HEADING statement .....	6-244
HUSH statement .....	6-249
ICHECK function .....	6-251
ICONV function .....	6-254
ICONVS function .....	6-256
IF statement .....	6-257
IFS function .....	6-260
ILPROMPT function .....	6-261
INCLUDE statement .....	6-264
INDEX function .....	6-265
INDEXS function .....	6-267
INDICES function .....	6-268
INMAT function .....	6-272
INPUT statement .....	6-273
INPUTCLEAR statement .....	6-278
INPUTDISP statement .....	6-279

INPUTDP statement .....	6-280
INPUTERR statement .....	6-281
INPUTIF statement .....	6-282
INPUTNULL statement .....	6-283
INPUTTRAP statement .....	6-284
INS statement .....	6-285
INSERT function .....	6-288
INT function .....	6-291
ISNULL function .....	6-292
ISNULLS function .....	6-293
ITYPE function .....	6-294
KEYEDIT statement .....	6-296
KEYEXIT statement .....	6-302
KEYIN function .....	6-304
KEYTRAP statement .....	6-305
LEFT function .....	6-307
LEN function .....	6-308
LENDP function .....	6-309
LENS function .....	6-310
LENSDP function .....	6-311
LES function .....	6-312
LET statement .....	6-313
LN function .....	6-314
LOCALEINFO function .....	6-315
LOCATE statement (IDEAL and REALITY syntax) .....	6-316
LOCATE statement (INFORMATION syntax) .....	6-320
LOCATE statement (PICK syntax) .....	6-324
LOCK statement .....	6-327
LOOP statement .....	6-329
LOWER function .....	6-332
LTS function .....	6-334
MAT statement .....	6-335
MATBUILD statement .....	6-337

MATCH operator .....	6-338
MATCHFIELD function .....	6-340
MATPARSE statement .....	6-342
MATREAD statements .....	6-345
MATREADL statement .....	6-350
MATREADU statement .....	6-351
MATWRITE statements .....	6-352
MATWRITEU statement .....	6-356
MAXIMUM function .....	6-357
MINIMUM function .....	6-358
MOD function .....	6-359
MODS function .....	6-360
MULS function .....	6-361
NAP statement .....	6-362
NEG function .....	6-363
NEGS function .....	6-364
NES function .....	6-365
NEXT statement .....	6-366
NOBUF statement .....	6-367
NOT function .....	6-368
NOTS function .....	6-369
NULL statement .....	6-370
NUM function .....	6-371
NUMS function .....	6-372
OCONV function .....	6-373
OCONVS function .....	6-376
ON statement .....	6-377
OPEN statement .....	6-380
OPENCHECK statement .....	6-384
OPENDEV statement .....	6-386
OPENPATH statement .....	6-389
OPENSEQ statement .....	6-392
ORS function .....	6-397

PAGE statement .....	6-398
PERFORM statement .....	6-399
PRECISION statement .....	6-401
PRINT statement .....	6-402
PRINTER statement .....	6-404
PRINTERR statement .....	6-406
PROCREAD statement .....	6-408
PROCWRITE statement .....	6-409
PROGRAM statement .....	6-410
PROMPT statement .....	6-411
PWR function .....	6-412
QUOTE function .....	6-413
RAISE function .....	6-414
RANDOMIZE statement .....	6-416
READ statements .....	6-417
READBLK statement .....	6-423
READL statement .....	6-425
READLIST statement .....	6-426
READNEXT statement .....	6-428
READSEQ statement .....	6-430
READT statement .....	6-432
READU statement .....	6-434
READV statement .....	6-435
READVL statement .....	6-436
READVU statement .....	6-437
REAL function .....	6-438
RECORDLOCK statements .....	6-439
RECORDLOCKED function .....	6-442
RELEASE statement .....	6-444
REM function .....	6-446
REM statement .....	6-447
REMOVE function .....	6-448
REMOVE statement .....	6-450

REPEAT statement .....	6-452
REPLACE function .....	6-453
RETURN statement .....	6-457
RETURN (value) statement .....	6-459
REUSE function .....	6-460
REVREMOVE statement .....	6-462
REWIND statement .....	6-464
RIGHT function .....	6-465
RND function .....	6-466
ROLLBACK statement .....	6-467
RPC.CALL function .....	6-469
RPC.CONNECT function .....	6-471
RPC.DISCONNECT function .....	6-473
SADD function .....	6-474
SCMP function .....	6-475
SDIV function .....	6-476
SEEK statement .....	6-477
SEEK(ARG.) statement .....	6-479
SELECT statements .....	6-481
SELECTE statement .....	6-484
SELECTINDEX statement .....	6-485
SELECTINFO function .....	6-487
SEND statement .....	6-488
SENTENCE function .....	6-489
SEQ function .....	6-490
SEQS function .....	6-491
SET TRANSACTION ISOLATION LEVEL statement .....	6-492
SETLOCALE function .....	6-494
SETREM statement .....	6-496
SIN function .....	6-497
SINH function .....	6-498
SLEEP statement .....	6-499
SMUL function .....	6-500

SOUNDEX function .....	6-501
SPACE function .....	6-502
SPACES function .....	6-503
SPLICE function .....	6-504
SQRT function .....	6-505
SQUOTE function .....	6-506
SSELECT statements .....	6-507
SSUB function .....	6-510
STATUS function .....	6-511
STATUS statement .....	6-516
STOP statement .....	6-520
STORAGE statement .....	6-522
STR function .....	6-523
STRS function .....	6-524
SUBR function .....	6-525
SUBROUTINE statement .....	6-527
SUBS function .....	6-528
SUBSTRINGS function .....	6-529
SUM function .....	6-530
SUMMATION function .....	6-532
SYSTEM function .....	6-533
TABSTOP statement .....	6-538
TAN function .....	6-539
TANH function .....	6-540
TERMINFO function .....	6-541
TIME function .....	6-560
TIMEDATE function .....	6-561
TIMEOUT statement .....	6-562
TPARM function .....	6-564
TPRINT statement .....	6-567
TRANS function .....	6-569
transaction statements .....	6-571
TRANSACTION ABORT statement .....	6-572

TRANSACTION COMMIT statement .....	6-574
TRANSACTION START statement .....	6-575
TRIM function .....	6-576
TRIMB function .....	6-578
TRIMBS function .....	6-579
TRIMF function .....	6-580
TRIMFS function .....	6-581
TRIMS function .....	6-582
TTYCTL statement .....	6-583
TTYGET statement .....	6-585
TTYSET statement .....	6-591
UNASSIGNED function .....	6-593
UNICHAR function .....	6-594
UNICHARS function .....	6-595
UNISEQ function .....	6-596
UNISEQS function .....	6-597
UNLOCK statement .....	6-598
UPCASE function .....	6-599
UPRINT statement .....	6-600
WEOF statement .....	6-601
WEOFSEQ statement .....	6-602
WRITE statements .....	6-604
WRITEBLK statement .....	6-610
WRITELIST statement .....	6-612
WRITESEQ statement .....	6-613
WRITESEQF statement .....	6-616
WRITET statement .....	6-618
WRITEU statement .....	6-620
WRITEV statement .....	6-621
WRITEVU statement .....	6-622
XLATE function .....	6-623
XTD function .....	6-625



## **Appendix A. Quick Reference**

Compiler Directives .....	A-1
Declarations .....	A-3
Assignments .....	A-3
Program Flow Control .....	A-3
File I/O .....	A-5
Sequential File I/O .....	A-7
Printer and Terminal I/O .....	A-8
Tape I/O .....	A-10
Select Lists .....	A-10
String Handling .....	A-11
Data Conversion and Formatting .....	A-14
NLS .....	A-15
Mathematical Functions .....	A-16
Relational Functions .....	A-19
System .....	A-20
Remote Procedure Calls .....	A-21
Miscellaneous .....	A-21

## **Appendix B. ASCII and Hex Equivalents**

## **Appendix C. Correlative and Conversion Codes**

A code: Algebraic Functions .....	C-4
BB and BX codes: Bit Conversion .....	C-8
C code: Concatenation .....	C-9
D code: Date Conversion .....	C-11
DI code: International Date Conversion .....	C-17
ECS code: Extended Character Set Conversion .....	C-18
F code: Mathematical Functions .....	C-19
G code: Group Extraction .....	C-22
L code: Length Function .....	C-23
MC codes: Masked Character Conversion .....	C-24
MD code: Masked Decimal Conversion .....	C-26
ML and MR codes: Formatting Numbers .....	C-29

MM code: Monetary Conversion .....	C-32
MP code: Packed Decimal Conversion .....	C-34
MT code: Time Conversion .....	C-35
MX, MO, MB, and MU0C codes: Radix Conversion .....	C-37
MY code: ASCII Conversion .....	C-39
NL code: Arabic Numeral Conversion .....	C-40
NLSmapname code: NLS Map Conversion .....	C-41
NR code: Roman Numeral Conversion .....	C-42
P code: Pattern Matching .....	C-43
Q code: Exponential Notation .....	C-44
R code: Range Function .....	C-46
S (Soundex) code .....	C-47
S (substitution) code .....	C-48
T code: Text Extraction .....	C-49
Tfile code: File Translation .....	C-50
TI code: International Time Conversion .....	C-52

## **Appendix D. BASIC Reserved Words**

## **Appendix E. @Variables**

## **Appendix F. BASIC Subroutines**

!ASYNC subroutine .....	F-4
!EDIT.INPUT subroutine .....	F-6
!ERRNO subroutine .....	F-13
!FCMP subroutine .....	F-14
!GET.KEY subroutine .....	F-15
!GET.PARTNUM subroutine .....	F-17
!GET.PATHNAME subroutine .....	F-19
!GETPU subroutine .....	F-20
!GET.USER.COUNTS subroutine .....	F-24
!INLINE.PROMPTS subroutine .....	F-25
!INTS subroutine .....	F-27
!MAKE.PATHNAME subroutine .....	F-28
!MATCHES subroutine .....	F-29

!MESSAGE subroutine .....	F-31
!PACK.FNKEYS subroutine .....	F-33
!REPORT.ERROR subroutine .....	F-38
!SET.PTR subroutine .....	F-40
!SETPU subroutine .....	F-42
!TIMDAT subroutine .....	F-46
!USER.TYPE subroutine .....	F-48
!VOC.PATHNAME subroutine .....	F-49

## **Index**



# Preface

This manual describes the UniVerse BASIC programming language. It is for experienced programmers and includes explanations of all BASIC statements and functions supported by UniVerse as well as information regarding the use of BASIC with UniVerse in the UNIX and Windows NT environments. If you have never used BASIC, read Chapter 1 and Chapter 2 before you begin. Before using any statement or function, thoroughly read its description in Chapter 6.

If you have previously used a BASIC programming language, you can skim through the first two chapters to determine the difference that may exist between UniVerse BASIC and the BASIC you have used in the past.

## Organization of This Manual

This manual contains the following:

Chapter 1 covers information you should know before you begin to use BASIC, such as initial procedures, terminology, and features that are unique to this implementation of BASIC.

Chapter 2 describes types of data, such as constants and variables, and types of operators.

Chapter 3 describes the UniVerse BASIC compiler. The discussion includes instructions on how to run the compiler, compiling options, warnings and error messages, and other related commands.

Chapter 4 describes how to use locks, transaction processing, and isolation levels to prevent data loss and other data conflicts.

Chapter 5 describes the tools available for debugging UniVerse BASIC programs. Included is an interactive debugger, RAID, and the program listing command, VLIST.

Chapter 6 contains statements and functions in alphabetical order. At the top of each page is the syntax for the statement or function, followed by a detailed description of its use, often including references to other statements or functions that can be used with it or are helpful to know about. Examples illustrate the application of the statement or function in a program.

Appendix A is a quick reference for BASIC statements and functions grouped according to use.

Appendix B is a table of ASCII character codes and equivalents and hexadecimal equivalents.

Appendix C describes the syntax and use of correlative and conversion codes.

Appendix D lists UniVerse BASIC reserved words.

Appendix E is a quick reference for UniVerse BASIC @variables.

Appendix F describes subroutines you can call from UniVerse BASIC programs.

## Documentation Conventions

This manual uses the following conventions:

Convention	Usage
<b>Bold</b>	In syntax, bold indicates commands, function names, and options. In text, bold indicates keys to press, function names, menu selections, and MS-DOS commands.
UPPERCASE	In syntax, uppercase indicates UniVerse commands, keywords, and options; BASIC statements and functions; and SQL statements and keywords. In text, uppercase also indicates UniVerse identifiers such as filenames, account names, schema names, and Windows NT filenames and pathnames.
<i>Italic</i>	In syntax, italic indicates information that you supply. In text, italic also indicates UNIX commands and options, filenames, and pathnames.
Courier	Courier indicates examples of source code and system output.
<b>Courier Bold</b>	In examples, courier bold indicates characters that the user types or keys the user presses (for example, <Return>).
[ ]	Brackets enclose optional items. Do not type the brackets unless indicated.
{ }	Braces enclose nonoptional items from which you must select at least one. Do not type the braces.
itemA   itemB	A vertical bar separating items indicates that you can choose only one item. Do not type the vertical bar.

Convention	Usage
...	Three periods indicate that more of the same type of item can optionally follow.
➤	A right arrow between menu options indicates you should choose each option in sequence. For example, “Choose <b>File ➤ Exit</b> ” means you should choose <b>File</b> from the menu bar, then choose <b>Exit</b> from the File pull-down menu.
⌘	Item mark. For example, the item mark (⌘) in the following string delimits elements 1 and 2, and elements 3 and 4: 1⌘2F3⌘4V5
F	Field mark. For example, the field mark (F) in the following string delimits elements FLD1 and VAL1: FLD1FVAL1VSUBV1SSUBV2
v	Value mark. For example, the value mark (v) in the following string delimits elements VAL1 and SUBV1: FLD1FVAL1VSUBV1SSUBV2
s	Subvalue mark. For example, the subvalue mark (s) in the following string delimits elements SUBV1 and SUBV2: FLD1FVAL1VSUBV1SSUBV2
τ	Text mark. For example, the text mark (τ) in the following string delimits elements 4 and 5: 1F2s3v4τ5

The following conventions are also used:

- Syntax definitions and examples are indented for ease in reading.
- All punctuation marks included in the syntax—for example, commas, parentheses, or quotation marks—are required unless otherwise indicated.
- Syntax lines that do not fit on one line in this manual are continued on subsequent lines. The continuation lines are indented. When entering syntax, type the entire syntax entry, including the continuation lines, on the same input line.

## UniVerse Documentation

UniVerse documentation includes the following:

**UniVerse BASIC:** Contains comprehensive information about the UniVerse BASIC language. It includes reference pages for all BASIC statements and functions. It is for experienced programmers.

**UniVerse BASIC SQL Client Interface Guide:** Describes how to use the BASIC SQL Client Interface (BCI), an interface to UniVerse and non-UniVerse databases from UniVerse BASIC. The BASIC SQL Client Interface uses ODBC-like function calls to execute SQL statements on local or remote database servers such as UniVerse, ORACLE, SYBASE, or INFORMIX. This book is for experienced SQL programmers.

**Administering UniVerse:** Describes tasks performed by UniVerse administrators, such as starting up and shutting down the system, system configuration and maintenance, system security, maintaining and transferring UniVerse accounts, maintaining peripherals, backing up and restoring files, and managing file and record locks, and network services. This book includes descriptions of how to use the UniVerse Admin program on a Windows client and how to use shell commands on UNIX systems to administer UniVerse.

**UniVerse Transaction Logging and Recovery:** Describes the UniVerse transaction logging subsystem, including both transaction and warmstart logging and recovery. This book is for system administrators.

**UniVerse System Description:** Provides detailed and advanced information about UniVerse features and capabilities for experienced users. This book describes how to use UniVerse commands, work in a UniVerse environment, create a UniVerse database, and maintain UniVerse files.

**UniVerse User Reference:** Contains reference pages for all UniVerse commands, keywords, and user records, allowing experienced users to refer to syntax details quickly.

**Guide to Retrieve:** Describes Retrieve, the UniVerse query language that lets users select, sort, process, and display data in UniVerse files. This book is for users who are familiar with UniVerse.

**Guide to ProVerb:** Describes ProVerb, a UniVerse processor used by application developers to execute prestored procedures called procs. This book describes tasks such as relational data testing, arithmetic processing, and transfers to subroutines. It also includes reference pages for all ProVerb commands.

**Guide to the UniVerse Editor:** Describes in detail how to use the Editor, allowing users to modify UniVerse files or programs. This book also includes reference pages for all UniVerse Editor commands.

**UniVerse NLS Guide:** Describes how to use and manage UniVerse's National Language Support (NLS). This book is for users, programmers, and administrators.



**UniVerse SQL Administration for DBAs:** Describes administrative tasks typically performed by DBAs, such as maintaining database integrity and security, and creating and modifying databases. This book is for database administrators (DBAs) who are familiar with UniVerse.

**UniVerse SQL User Guide:** Describes how to use SQL functionality in UniVerse applications. This book is for application developers who are familiar with UniVerse.

**UniVerse SQL Reference:** Contains reference pages for all SQL statements and keywords, allowing experienced SQL users to refer to syntax details quickly. It includes the complete UniVerse SQL grammar in Backus Naur Form (BNF).

**UniVerse Master Index:** A comprehensive index for UniVerse documentation.

**UniVerse Quick Reference:** A quick reference to all UniVerse commands and keywords. It also summarizes UniVerse SQL statements and keywords, all elements of the UniVerse BASIC language, Editor commands, ProVerb commands, file types, file dictionaries, and user exits.

## Related Documentation

The following documentation is also available:

**UniVerse GCI Guide:** Describes how to use the General Calling Interface (GCI) to call subroutines written in C, C++, or FORTRAN from BASIC programs. This book is for experienced programmers who are familiar with UniVerse.

**UniVerse ODBC Guide:** Describes how to install and configure a UniVerse ODBC server on a UniVerse host system. It also describes how to use UniVerse ODBC Config and how to install, configure, and use UniVerse ODBC drivers on client systems. This book is for experienced UniVerse developers who are familiar with SQL and ODBC.

**UV/Term Guide:** Provides step-by-step instructions for how to install the UV/Term software on a PC, how to define terminal parameters, and how to start up and use UV/Term. This book also describes the terminal emulator window and how to configure the graphical user interface.

**UV/Net II Guide:** Describes UV/Net II, the UniVerse transparent database networking facility that lets users access UniVerse files on remote systems. This book is for experienced UniVerse administrators.

**UniVerse Guide for Pick Users:** Describes UniVerse for new UniVerse users familiar with Pick-based systems.

***Moving to UniVerse from PI/open:*** Describes how to prepare the PI/open environment before converting PI/open applications to run under UniVerse. This book includes step-by-step procedures for converting INFO/BASIC programs, accounts, and files. This book is for experienced PI/open users and does not assume detailed knowledge of UniVerse.

## Common APIs Documentation

The following books document application programming interfaces (APIs) used for developing client applications that connect to UniVerse and UniData servers.

***Administrative Supplement for Common APIs:*** Introduces Ardent Software's five common APIs, and provides important information that developers using any of the common APIs will need. It includes information about the UniRPC, the UCI Config Editor, the *ud\_database* file, and device licensing.

***UCI Developer's Guide:*** Describes how to use UCI (Uni Call Interface), an interface to UniVerse databases from C-based client programs. UCI uses ODBC-like function calls to execute SQL statements on local or remote UniVerse servers. This book is for experienced SQL programmers.

***InterCall Developer's Guide:*** Describes how to use the InterCall API to access data on UniVerse and UniData systems from external programs. This book is for experienced programmers who are familiar with UniVerse or UniData.

***UniObjects Developer's Guide:*** Describes UniObjects, an interface to UniVerse and UniData systems from Visual Basic. This book is for experienced programmers and application developers who are familiar with UniVerse or UniData, and with Visual Basic, and who want to write Visual Basic programs that access these databases.

***UniObjects for Java Developer's Guide:*** Describes UniObjects for Java, an interface to UniVerse and UniData systems from Java. This book is for experienced programmers and application developers who are familiar with UniVerse or UniData, and with Java, and who want to write Java programs that access these databases.

***Using UniOLEDB:*** Describes how to use UniOLEDB, an interface to UniVerse and UniData systems for OLE DB consumers. This book is for experienced programmers and application developers who are familiar with UniVerse or UniData, and with OLE DB, and who want to write OLE DB programs that access these databases.

# 1

## Introduction to UniVerse BASIC

UniVerse BASIC is a business-oriented programming language designed to work efficiently with the UniVerse environment. It is easy for a beginning programmer to use yet powerful enough to meet the needs of an experienced programmer.

The power of UniVerse BASIC comes from statements and built-in functions that take advantage of the extensive database management capabilities of UniVerse. These benefits combined with other BASIC extensions result in a development tool well-suited for a wide range of applications.

The extensions in UniVerse BASIC include the following:

- Optional statement labels (that is, statement numbers)
- Statement labels of any length
- Multiple statements allowed on one line
- Computed GOTO statements
- Complex IF statements
- Multiline IF statements
- Priority CASE statement selection
- String handling with variable length strings up to  $2^{32}-1$  characters
- External subroutine calls
- Direct and indirect subroutine calls
- Magnetic tape input and output
- Retrieve data conversion capabilities
- UniVerse file access and update capabilities
- File-level and record-level locking capabilities
- Pattern matching
- Dynamic arrays

# BASIC Terminology

UniVerse BASIC programmers should understand the meanings of the following terms:

- BASIC program
- Source code
- Object code
- Variable
- Function
- Keyword

**BASIC Program.** A BASIC program is a set of statements directing the computer to perform a series of tasks in a specified order. A BASIC statement is made up of *keywords* and *variables*.

**Source Code.** Source code is the original form of the program written by the programmer.

**Object Code.** Object code is compiler output, which can be executed by the UniVerse RUN command or called as a subroutine.

**Variable.** A variable is a symbolic name assigned to one or more data values stored in memory. A variable's value can be numeric or character string data, the null value, or it can be defined by the programmer, or it can be the result of operations performed by the program. Variable names can be as long as the physical line, but only the first 64 characters are significant. Variable names begin with an alphabetic character and can include alphanumeric characters, periods ( . ), dollar signs ( \$ ), and percent signs ( % ). Upper- and lowercase letters are interpreted as different; that is, REC and Rec are different variables.

**Function.** A BASIC intrinsic function performs mathematical or string manipulations on its arguments. It is referenced by its keyword name and is followed by the required arguments enclosed in parentheses. Functions can be used in expressions; in addition, function arguments can be expressions that include functions. UniVerse BASIC contains both numeric and string functions.

- **Numeric functions.** BASIC can perform certain arithmetic or algebraic calculations, such as calculating the sine (SIN), cosine (COS), or tangent (TAN) of an angle passed as an argument.
- **String functions.** A string function operates on ASCII character strings. For example, the TRIM function deletes extra blank spaces and tabs from a character string, and the STR function generates a particular character string a specified number of times.

**Keyword.** A BASIC keyword is a word that has special significance in a BASIC program statement. The case of a keyword is ignored; for example, READU and readu are the same keyword. For a [list of keywords](#), see Appendix D.

## Subroutines

A *subroutine* is a set of instructions that perform a specific task. It is a small program that can be embedded in a program and accessed with a GOSUB statement, or it can be external to the program and accessed with a CALL statement. Common processes are often kept as external subroutines. This lets the programmer access them from many different programs without having to rewrite them.

When a [GOSUB](#) or [CALL](#) statement is encountered, program control branches to the referenced subroutine. An internal subroutine must begin with a statement label. An external subroutine must begin with a [SUBROUTINE](#) statement.

A [RETURN](#) statement can be used at the end of a subroutine to return program flow to the statement following the last referenced GOSUB or CALL statement. If there is no corresponding CALL or GOSUB statement, the program halts and returns to the UniVerse command level. If an external subroutine ends before a RETURN statement is encountered, a RETURN is provided automatically.

**Note:** If an [ABORT](#), [STOP](#), or [CHAIN](#) statement is encountered during subroutine execution, program execution aborts, stops, or chains to another BASIC program and control never returns to the calling program.

One or more arguments separated by commas can be passed to the subroutine as an *argument list*. An argument can be a constant, variable, array variable, or expression, each representing an actual value. The [SUBROUTINE argument list](#) must contain the same number of arguments so that the subroutine can reference the values being passed to it. Arguments are passed to subroutines by passing a pointer to the argument. Therefore, arguments can also be used to return values to the calling program.

## Source Syntax

A BASIC source line has the following syntax:

```
[ label ] statement [ ; statement ... ] <Return>
```

You can put more than one statement on a line. Separate the statements with semicolons.

A BASIC source line can begin with a statement label. It always ends with a carriage return (**Return**). It can contain up to 256 characters and can extend over more than one physical line.

## Statement Types

BASIC statements can be used for any of the following purposes:

- Input and output control
- Program control
- Assignment (assigning a value to a variable)
- Specification (specifying the value of a constant)
- Documentation

*Input statements* indicate where the computer can expect data to come from (for example, the keyboard, a particular file, and so on). *Output statements* control where the data is displayed or stored.

In general, BASIC statements are executed in the order in which they are entered. *Control statements* alter the sequence of execution by branching to a statement other than the next statement, by conditionally executing statements, or by passing control to a subroutine.

*Assignment statements* assign values to variables, and *specification statements* assign names to constants.

Program documentation is accomplished by including optional *comments* that explain or document various parts of the program. Comments are part of the source code only and are not executable. They do not affect the size of the object code. Comments must begin with one of the following:

```
REM      *      !      $*
```

Any text that appears between a comment symbol and a carriage return is treated as part of the comment. Comments cannot be embedded in a BASIC statement. If you want to put a comment on the same physical line as a statement, you must end the statement with a semicolon ( ; ), then add the comment, as in the following example:

```
IF X THEN
  A = B; REM correctly formatted comment statement
  B = C
END
```

You cannot put comments between multiple statements on one physical line. For example, in the second line of the following program the statement `B = C` is part of the comment and is not executed:

```
IF X THEN
  A = B; REM The rest of this line is a comment; B = C
END
```

However, you can put comments in the middle of a statement that occupies more than one physical line, as in the following example:

```
A = 1
B = 2
IF A =
  REM comment
  PRINT A
  REM comment
END ELSE PRINT B
```

## Statement Labels

A statement label is a unique identifier for a program line. A statement label consists of a string of characters followed by a colon. The colon is optional when the statement label is completely numeric. Like variable names, alphanumeric statement labels begin with an alphabetic character and can include periods (.), dollar signs (\$), and percent signs (%). Upper- and lowercase letters are interpreted as different; that is, `ABC` and `Abc` are different labels. Statement labels, like variable names, can be as long as the length of the physical line, but only the first 64 characters are significant. A statement label can be put either in front of a BASIC statement or on its own line. The label must be first on the line—that is, the label cannot begin with a space.

## Spaces or Tabs

In a program line, spaces or tabs that are not part of a data item are ignored. Therefore you can use spaces or tabs to improve the program's appearance and readability.

## Newlines and Sequential File I/O

UniVerse BASIC uses the term *newline* to indicate the character or character sequence that defines where a line ends in a record in a type 1 or type 19 file. The newline differs according to the operating system you are using. On UNIX file

systems, a newline consists of a single LINEFEED character. On Windows NT file systems, a newline consists of the character sequence RETURN + LINEFEED.

UniVerse BASIC handles this difference transparently in nearly every case, but in a few instances the operating system differences become apparent. If you want your program to work on different operating systems, watch sequential file I/O (that is, writing to or reading from type 1 and type 19 files, line by line or in blocks of data). In particular, be aware of the potential differences that occur:

- When moving a pointer through a file
- When reading or writing blocks of data of a specified length

## Special Characters

The UniVerse BASIC character set comprises alphabetic, numeric, and special characters. The alphabetic characters are the upper- and lowercase letters of the alphabet. The numeric characters are the digits 0 through 9. The special characters are as follows. Most of the special characters are not permitted in a numeric constant or a variable name.

	Space
	Tab
=	Equal sign or assignment symbol
+	Plus sign
-	Minus sign
*	Asterisk, multiplication symbol, or nonexecutable comment
**	Exponentiation
/	Slash or division symbol
^	Up-arrow or exponentiation symbol
(	Left parenthesis
)	Right parenthesis
#	Number (pound or hash) sign or not equal to
\$	Dollar sign
!	Exclamation point or nonexecutable comment
[	Left bracket
]	Right bracket
,	Comma (not permitted in numeric data)
.	Period or decimal point



'	Single quotation mark or apostrophe
;	Semicolon
:	Colon or concatenation
&	Ampersand (and)
<	Less than (left angle bracket)
>	Greater than (right angle bracket)
@	At sign
_	Underscore

## Storing Programs

BASIC programs are stored as records in type 1 or type 19 files. The program file must exist before you invoke an editor to create a new record to hold your program. Record IDs must follow the conventions for type 1 and type 19 files.

## Editing Programs

You can use the UniVerse Editor or any suitable editor, such as *vi* on UNIX or *edit* on Windows NT, to write your programs. You can edit programs in the UniVerse environment or at the operating system level.

### Editing Programs in UniVerse

On UNIX systems you can invoke *vi* from the UniVerse system prompt using this syntax:

**VI** *pathname*

*pathname* is the relative or absolute pathname of the program you want to edit. For example, the program PAYROLL is stored as a record in the file BP. To edit it with *vi*, enter:

```
>VI BP/PAYROLL
```

If you want to use *vi*, or any other editor, directly from UniVerse, you can create a VOC entry that invokes your chosen editor. For example, this VOC entry calls *edit* from UniVerse on Windows NT:

```
EDIT
001 V
002 \win25\edit.com
003 PR
```

## Editing Programs Outside UniVerse

When you invoke an editor at the operating system level, remember that the UniVerse file holding the programs is implemented as a directory at the operating system level. For example, the YEAR.END program is stored as a record in the BP file in UniVerse. Its operating system pathname is BP\YEAR.END on Windows NT systems and BP/YEAR.END on UNIX systems.

## Getting Started

To create and use a BASIC program, follow these steps:

1. Use the **CREATE.FILE** command to create a type 1 or type 19 UniVerse file to store your BASIC program source. The **RUN** command uses the filename BP if a filename is not specified, so many people use BP as the name of their general BASIC program file.
2. Use the UniVerse **Editor** or some other editor to create the source for your BASIC program as a record in the file you created in step 1.
3. Once you have created the record containing your BASIC program source statements, use the **BASIC** command to compile your program. The BASIC command creates a file to contain the object code output by the compiler. You do not have to know the name of the object file because the program is always referred to by the source filename.
4. If the BASIC compiler detects any errors, use the Editor to correct the source code and recompile using the BASIC command.
5. When your program compiles without any errors, execute it using the RUN command. Use the **RAID** command to debug your program.

# 2

## Data Types, Variables, and Operators

This chapter gives an overview of the fundamental components of the UniVerse BASIC language. It describes types of data, constants, variables, and how data is combined with arithmetic, string, relational, and logical operators to form expressions.

### Types of Data

Although many program languages distinguish different types of data, the UniVerse BASIC compiler does not. All data is stored internally as character strings, and data typing is done contextually at run time. There are three main types of data: character string, numeric, and unknown (that is, the null value).

### Character String Data

Character string data is represented internally as a sequence of ASCII characters. Character strings can represent either numeric or nonnumeric data. Their length is limited only by the amount of available memory. Numeric and nonnumeric data can be mixed in the same character string (for example, in an address).

In NLS mode all data is held in the UniVerse internal character set. In all UniVerse I/O operations, data is converted automatically by applying the map specified for a file or a device. One character can be more than one byte long and can occupy zero or more positions on the screen. UniVerse BASIC provides functions so that programs can determine what these characteristics are. For more information about [character sets](#), see *UniVerse NLS Guide*.

## Character String Constants

In BASIC source code, character string constants are a sequence of ASCII characters enclosed in single or double quotation marks, or backslashes ( \ ). These marks are not part of the character string value. The length of character string constants is limited to the length of a statement.

Some examples of character string constants are the following:

```
"Emily Daniels"  
'$42,368.99'  
'Number of Employees'  
"34 Cairo Lane"  
\"Fred's Place" isn't open\
```

The beginning and terminating marks enclosing character string data must match. In other words, if you begin a string with a single quotation mark, you must end the string with a single quotation mark.

If you use either a double or a single quotation mark within the character string, you must use the opposite kind to begin and end the string. For example, this string should be written:

```
"It's a lovely day."
```

And this string should be written:

```
'Double quotation marks (") enclosing this string would be  
wrong.'
```

The empty string is a special instance of character string data. It is a character string of zero length. Two adjacent double or single quotation marks, or backslashes, specify an empty string:

```
'' or "" or \\
```

In your source code you can use any ASCII character in character string constants except ASCII character 0 (NUL), which the compiler interprets as an end-of-string character, and ASCII character 10 (linefeed), which separates the logical lines of a program. Use CHAR(0) and CHAR(10) to embed these characters in a string constant.

## Numeric Data

All numeric data is represented internally either as floating-point numbers with the full range of values supported by the system's floating-point implementation, or as integers. On most systems the range is from  $10^{-307}$  through  $10^{+307}$  with 15 decimal digits of precision.

## Numeric Constants

Numeric constants can be represented in either fixed-point or floating-point form. Commas and spaces are not allowed in numeric constants.

**Fixed-Point Constants.** Fixed-point form consists of a sequence of digits, optionally containing a decimal point and optionally preceded by a plus ( + ) or minus ( - ) sign. Some examples of valid fixed-point constants are:

```
12
-132.4
+10428
```

**Floating-Point Constants.** Floating-point form, which is similar to scientific notation, consists of a sequence of digits, optionally preceded by a plus ( + ) or minus ( - ) sign representing the mantissa. The sequence of digits is followed by the letter E and digits, optionally preceded by a minus sign, representing the power of 10 exponent. The exponent must be in the range of -307 through +307. Some examples of valid floating-point constants are:

```
1.2E3
-7.3E42
-1732E-4
```

Use the [PRECISION](#) statement to set the maximum number of fractional digits that can result from converting numbers to strings.

## Unknown Data: The Null Value

The null value has a special run-time data type in UniVerse BASIC. It was added to UniVerse BASIC for compatibility with UniVerse SQL. The null value represents data whose value is unknown.

**Note:** Do not confuse the null value with the empty string. The empty string is a character string of zero length which is known to have no value. Unlike null, whose value is defined as unknown, the value of the empty string is known. You cannot use the empty string to represent the null value, nor can you use the null value to represent “no value.”

Like all other data in UniVerse BASIC, the null value is represented internally as a character string. The string is made up of the single byte CHAR(128). At run time when explicit or implicit dynamic array extractions are executed on this character, it is assigned the data type “null.” UniVerse BASIC programs can reference the null value using the system variable @NULL. They can test whether a value is the null value using the [ISNULL](#) and [ISNULLS](#) functions.

There is no printable representation of the null value. In this manual the symbol  $\lambda$  (lambda) is sometimes used to denote the null value.

Here is an example of the difference between an empty string and the null value. If you concatenate a string value with an empty string, the string value is returned, but if you concatenate a string value with the null value, null is returned.

```
A = @NULL
B = " "
C = "JONES"
X = C:B
Y = C:A
```

The resulting value of X is "JONES", but the value of Y is the null value. When you concatenate known data with unknown data, the result is unknown.

Programmers should also note the difference between the null value—a special constant whose type is “null”—and the stored representation of the null value—the special character CHAR(128) whose type is “string.” BASIC programs can reference the stored representation of null using the system variable @NULL.STR instead of @NULL.

## Constants

Constants are data that do not change in value, data type, or length during program execution. Constants can be character strings or numeric strings (in either integer or floating-point form). A character string of no characters—the empty string—can also be a constant.

## Variables

Variables are symbolic names that represent stored data values. The value of a variable can be:

- Unassigned
- A string, which can be an alphanumeric character string, a number, or a dynamic array
- A number, which can be fixed-point (an integer) or floating-point
- The null value
- A dimensioned array (that is, a vector or matrix)

- A subroutine name
- A file
- A select list

The value of a variable can be explicitly assigned by the programmer, or it can be the result of operations performed by the program during execution. Variables can change in value during program execution. At the start of program execution, all variables are unassigned. Any attempt to use an unassigned variable produces an error message.

A variable name must begin with an alphabetic character. It can also include one or more digits, letters, periods, dollar signs, or percent signs. Spaces and tabs are not allowed. A variable name can be any length up to the length of the physical line, but only the first 64 characters are significant. A variable name cannot be any of the [reserved words](#) listed in Appendix D. In UniVerse, upper- and lowercase characters in a variable name are interpreted differently.

UniVerse BASIC also provides a set of system variables called [@variables](#). Many of these are read-only variables. Read-only @variables cannot be changed by the programmer.

Most variables in BASIC remain available only while the current program or subroutine is running. Unnamed common variables, however, remain available until the program returns to the system prompt. Named common variables and @variables remain available until the user logs out of UniVerse. See the [COMMON](#) statement for information about named and unnamed common variables.

In NLS mode you can include characters outside the ASCII character set only as constants defined by the [\\$DEFINE](#) and [EQUATE](#) statements, or as comments. Everything else, including variable names, must use the ASCII character set. For more information about [character sets](#), see *UniVerse NLS Guide*.

## Array Variables

An array is a variable that represents more than one data value. There are two types of array: dimensioned and dynamic. Dimensioned arrays can be either standard or fixed. Fixed arrays are provided in PICK, IN2, and REALITY flavor accounts for compatibility with other Pick systems.

### Dimensioned Arrays

Each value in a dimensioned array is called an *element* of the array. Dimensioned arrays can be one- or two-dimensional.

A one-dimensional array is called a *vector*. Its elements are arranged sequentially in memory. An element of a vector is specified by the variable name followed by the index of the element enclosed in parentheses. The index of the first element is 1. The index can be a constant or an expression. Two examples of valid vector element specifiers are:

```
A(1)
COST(35)
```

A two-dimensional array is called a *matrix*. The elements of the first row are arranged sequentially in memory, followed by the elements of the second row, and so on. An element of a matrix is specified by the variable name followed by two indices enclosed in parentheses. The indices represent the row and column position of the element. The indices of the first element are (1,1). Indices can be constants or expressions. The indices used to specify the elements of a matrix that has four columns and three rows are illustrated by the following:

```
1,1      1,2      1,3      1,4
2,1      2,2      2,3      2,4
3,1      3,2      3,3      3,4
```

Two examples of valid matrix element specifiers are:

```
OBJ(3,1)
WIDGET(7,17)
```

Vectors are treated as matrices with a second dimension of 1. COST(35) and COST(35,1) are equivalent specifications and can be used interchangeably.

Both vectors and matrices have a special *zero element* that is used in [MATPARSE](#), [MATREAD](#), and [MATWRITE](#) statements. The zero element of a vector is specified by *vector.name*(0), and the zero element of a matrix is specified by *matrix.name*(0,0). Zero elements are used to store fields that do not fit in the dimensioned elements on MATREAD or MATPARSE statements.

Dimensioned arrays are allocated either at compile time or at run time, depending on the flavor of the account. Arrays allocated at run time are called *standard* arrays. Arrays allocated at compile time are called *fixed* arrays. Standard arrays are redimensionable; fixed arrays are not redimensionable and do not have a zero element. All arrays are standard unless the program is compiled in a PICK, IN2, or REALITY flavor account, in which case they are fixed arrays. To use fixed arrays in PIOPEN, INFORMATION and IDEAL flavor accounts, use the STATIC.DIM option of the [\\$OPTIONS](#) statement. To use standard arrays in PICK, IN2, and REALITY flavor accounts, use \$OPTIONS –STATIC.DIM.



## Dynamic Arrays

Dynamic arrays map the structure of UniVerse file records to character string data. Any character string can be a *dynamic array*. A dynamic array is a character string containing elements that are substrings separated by delimiters. At the highest level these elements are fields separated by field marks ( `F` ) (ASCII 254). Each field can contain values separated by value marks ( `v` ) (ASCII 253). Each value can contain subvalues separated by subvalue marks ( `s` ) (ASCII 252).

A common use of dynamic arrays is to store data that is either read in from or written out to a UniVerse file record. However, UniVerse BASIC includes facilities for manipulating dynamic array elements that make dynamic arrays a powerful data type for processing hierarchical information independently of UniVerse files.

The number of fields, values, or subvalues in a dynamic array is limited only by the amount of available memory. Fields, values, and subvalues containing the empty string are represented by two consecutive field marks, value marks, or subvalue marks, respectively.

The following character string is a dynamic array with two fields:

```
TOMSDICKSHARRYVBETTYSSUESMARYFJONESVSMITH
```

The two fields are:

```
TOMSDICKSHARRYVBETTYSSUESMARY
```

and:

```
JONESVSMITH
```

Conceptually, this dynamic array has an infinite number of fields, all of which are empty except the first two. References made to the third or fourth field, for example, return an empty string.

The first field has two values:

```
TOMSDICKSHARRY
```

and:

```
BETTYSSUESMARY
```

The first value has three subvalues: TOM, DICK, and HARRY. The second value also has three subvalues: BETTY, SUE, and MARY.

The second field has two values: JONES and SMITH. Each value has one subvalue: JONES and SMITH.

The following character string:

```
NAME AND ADDRESS
```

can be considered a dynamic array containing one field, which has one value, which has one subvalue, all of which are: NAME AND ADDRESS.

The following character string can be considered a dynamic array containing two fields:

```
JONESVSMITHVBROWN$1.23VV$2.75
```

The first field has three values: JONES, SMITH, and BROWN. The second field has three values: \$1.23, an empty string, and \$2.75

Intrinsic functions and operators allow individual subvalues, values, and fields to be accessed, changed, added, and removed.

You can create a dynamic array in two ways: by treating it as a concatenation of its fields, values, and subvalues; or by enclosing the elements of the dynamic array in angle brackets, using the syntax:

```
array.name < field# , value# , subvalue# >
```

For example, to create the dynamic array A as:

```
JONESVSMITHF1.23S20V2.50S10
```

you can say:

```
A="JONES":@VM:"SMITH":@FM:1.23:@SM:20:@VM:2.50:@SM:10
```

or you can say:

```
A = ""  
A<1,1> = "JONES"  
A<1,2> = "SMITH"  
A<2,1,1> = 1.23  
A<2,1,2> = 20  
A<2,2,1> = 2.50  
A<2,2,2> = 10
```

The example has two fields. The first field has two values, and the second field has two values. The first value of the second field has two subvalues, and the second value of the second field also has two subvalues.

You must use the following statements to declare that the first field contains the two values JONES and SMITH:

```
A = ""  
A<1,1> = "JONES"  
A<1,2> = "SMITH"
```

The statement:

```
A = " "  
A<1> = "JONES"
```

declares that the first field contains only JONES with no other values or subvalues. Similarly, the statement:

```
A<2,1> = 1.23
```

declares that the first value of the second field is 1.23 with no subvalues. The statements:

```
A<2,2,1> = 2.50  
A<2,2,2> = 10
```

declare that the second value of the second field has two subvalues, 2.50 and 10, respectively.

## File Variables

A file variable is created by a form of the **OPEN** statement. Once opened, a file variable is used in I/O statements to access the file. There are two types of file variable: hashed file variable and sequential file variable. File variables can be scalars or elements of a dimensioned array.

## Select List Variables

Select list variables are created by a form of the **SELECT** statement. A select list variable contains a select list and can be used only in **READNEXT** statements. Unlike other variables, a select list variable cannot be an element of a dimensioned array.

## Expressions

An expression is part of a BASIC statement. It can comprise:

- A string or numeric constant
- A variable
- An intrinsic function
- A user-defined function
- A combination of constants, variables, operators, functions, and other expressions

## Format Expressions

A format expression formats variables for output. It specifies the size of the field in which data is displayed or printed, the justification (left, right, or text), the number of digits to the right of the decimal point to display, and so on. Format expressions work like the [FMT](#) function. The syntax is:

*variable format*

*format* is a valid string expression that evaluates to:

[ *width* ] [ *background* ] *justification* [ *edit* ] [ *mask* ]

Either *width* or *mask* can specify the size of the display field.

*background* specifies the character used to pad the field (**Space** is the default padding character).

You must specify *justification* as left, right, or text (text left-justifies output, but breaks lines on spaces when possible).

*edit* specifies how to format numeric data for output, including such things as the number of digits to display to the right of the decimal point, the descaling factor, whether to round or truncate data, and how to indicate positive and negative currency, handle leading zeros, and so on.

*mask* is a pattern that specifies how to output data.

If a format expression is applied to the null value, the result is the same as formatting an empty string. This is because the null value has no printable representation.

You can use the STATUS function to determine the result of the format operation. The STATUS function returns the following after a format operation:

- 0 The format operation is successful.
- 1 The variable is invalid.
- 2 The format expression is invalid.

In NLS mode, the [FMT](#) function formats an expression in characters; the [FMTDP](#) function formats it in display positions. The effect of the format mask can be different if the data contains double-width or multibyte characters. For more information about [display length](#), see *UniVerse NLS Guide*.

# Operators

Operators perform mathematical, string, and logical operations on values. Operands are expressions on which operations are performed. BASIC operators are divided into the following categories:

- Arithmetic
- String
- Relational
- Pattern matching
- IF operator
- Logical
- Assignment
- Dynamic array

## Arithmetic Operators

Arithmetic operators combine operands comprising one or more variables, constants, or intrinsic functions. Resulting arithmetic expressions can be combined with other expressions almost indefinitely. The syntax of arithmetic expressions is:

*expression operator expression*

Table 2-1 lists the arithmetic operators used in BASIC, in order of evaluation.

**Table 2-1. Arithmetic Operators**

Operator	Operation	Sample Expression
–	Negation	–X
^	Exponentiation	X ^ Y
**		X ** Y
*	Multiplication	X * Y
/	Division	X / Y
+	Addition	X + Y
–	Subtraction	X – Y

You can use parentheses to change the order of evaluation. Operations on expressions enclosed in parentheses are performed before those outside parentheses.

The following expression is evaluated as  $112 + 6 + 2$ , or 120:

$(14 * 8) + 12 / 2 + 2$

On the other hand, the next expression is evaluated as  $14 * 20 / 4$ , or  $280 / 4$ , or 70:

```
14 * (8 + 12) / (2 + 2)
```

The result of any arithmetic operation involving the null value is the null value. Since the null value is unknown, the result of combining it with anything must also be unknown. So in the following example, B is the null value:

```
A = @NULL
B = 3 + A
```

The values of arithmetic expressions are internally maintained with the full floating-point accuracy of the system.

If a character string variable containing only numeric characters is used in an arithmetic expression, the character string is treated as a numeric variable. That is, the numeric string is converted to its equivalent internal number and then evaluated numerically in the arithmetic expression. For example, the following expression is evaluated as 77:

```
55 + "22"
```

If a character string variable containing nonnumeric characters is used in an arithmetic expression, a warning message appears, and the string is treated as zero. For example, the following expression is evaluated as 85, and a message warns that the data is nonnumeric:

```
"5XYZ" + 85
```

A BASIC program compiled in an INFORMATION or a PIOPEN flavor account has arithmetic instructions capable of operating on multivalued data. The following statement in a program compiled in an INFORMATION or a PIOPEN flavor account is valid:

```
C = (23:@VM:46) * REUSE(2)
```

In a BASIC program compiled in an IDEAL, PICK, PIOPEN, REALITY, or IN2 flavor account, arithmetic can be performed on string data only if the string can be interpreted as a single-valued number. The previous statement successfully compiles in PICK, PIOPEN, IN2, REALITY, and IDEAL flavor accounts but causes a run-time error. The [REUSE](#) function converts 2 to a string which then has to be converted back to a number for the arithmetic operation. This is harmless. The multivalued string cannot be converted to a number and causes a *nonnumeric data* warning.

The IDEAL flavor uses single-valued arithmetic because of the performance penalty incurred by multivalued arithmetic. To perform multivalued arithmetic in

IDEAL, PICK, PIOPEN, IN2, and REALITY flavor accounts, use the VEC.MATH option of the [\\$OPTIONS](#) statement.

## String Operators

The concatenation operator ( : or CAT) links string expressions to form compound string expressions, as follows:

```
'HELLO. ' : 'MY NAME IS ' : X : ". WHAT'S YOURS?"
```

or:

```
'HELLO. ' CAT 'MY NAME IS ' CAT X CAT ". WHAT'S YOURS?"
```

If, for instance, the current value of X is JANE, these string expressions both have the following value:

```
"HELLO. MY NAME IS JANE. WHAT'S YOURS?"
```

Multiple concatenation operations are performed from left to right. Parenthetical expressions are evaluated before operations outside the parentheses.

With the exception of the null value, all operands in concatenated expressions are considered to be string values, regardless of whether they are string or numeric expressions. However, the precedence of arithmetic operators is higher than the concatenation operator. For example:

```
"THERE ARE " : "2" + "2" : "3" : " WINDOWS."
```

has the value:

```
"THERE ARE 43 WINDOWS."
```

The result of any string operation involving the null value is the null value. Since the null value represents an unknown value, the results of operations on that value are also unknown. But if the null value is referenced as a character string containing only the null value (that is, as the string CHAR(128) ), it is treated as character string data. For example, the following expression evaluates to null:

```
"A" : @NULL
```

But this expression evaluates to "A<CHAR128>":

```
"A" : @NULL.STR
```

## Substring Operator

A *substring* is a subset of contiguous characters of a character string. For example, JAMES is a substring of the string JAMES JONES. JAMES JON is also a substring of JAMES JONES.

You can specify a substring as a variable name or an array element specifier, followed by two values separated by a comma and enclosed in square brackets. The two values specify the starting character position and the length of the substring. The syntax is:

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

The bold brackets are part of the syntax and must be typed.

If *start* is 0 or a negative number, the starting position is assumed to be 1. If *start* is omitted, 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. So the following substring specification:

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

returns the substring:

```
67890
```

The following example:

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

displays the following output:

```
DHHH
```

Another syntax for removing substrings from a string, similar to the previous syntax, is:

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

The bold brackets are part of the syntax and must be typed. Use this syntax to return the substring that is located between the stated number of occurrences of the specified delimiter. *fields* specifies the number of successive fields after the specified occurrence of the delimiter that are to be returned with the substring. The delimiter is part of the returned value when successive fields are returned. This syntax performs the same function as the **FIELD** function.



All substring syntaxes can be used with the assignment operator ( = ) to replace the value normally returned by the [ ] operator with the value assigned to the variable. For example:

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

returns the following:

```
A= 121212
```

Assigning the three-argument syntax of the [ ] operator provides the same functionality as the [FIELDSTORE](#) function.

## Relational Operators

Relational operators compare numeric, character string, or logical data. The result of the comparison, either true ( 1 ) or false ( 0 ), can be used to make a decision regarding program flow (see the [IF](#) statement). Table 2-2 lists the relational operators.

**Table 2-2. Relational Operators**

Operator	Relation	Example
EQ or =	Equality	X = Y
NE or #	Inequality	X # Y
>< or <>	Inequality	X <> Y
LT or <	Less than	X < Y
GT or >	Greater than	X > Y
LE or <= or =< or #>	Less than or equal to	X <= Y
GE or >= or => or #<	Greater than or equal to	X >= Y

When arithmetic and relational operators are both used in an expression, the arithmetic operations are performed first. For example, the expression:

```
X + Y < ( T - 1 ) / Z
```

is true if the value of X plus Y is less than the value of T minus 1 divided by Z.

String comparisons are made by comparing the ASCII values of single characters from each string. The string with the higher numeric ASCII code equivalent is

considered to be greater. If all the ASCII codes are the same, the strings are considered equal.

If the two strings have different lengths, but the shorter string is otherwise identical to the beginning of the longer string, the longer string is considered greater.

**Note:** An empty string is always compared as a character string. It does not equal numeric zero.

A space is evaluated as less than zero. Leading and trailing spaces are significant. If two strings can be converted to numeric, then the comparison is always made numerically.

Some examples of true comparisons are:

```
"AA" < "AB"  
"FILENAME" = "FILENAME"  
"X&" > "X#"  
"CL " > "CL"  
"kg" > "KG"  
"SMYTH" < "SMYTHE"  
B$ < "9/14/93" (where B$ = "8/14/93")
```

The results of any comparison involving the null value cannot be determined—that is, the result of using a relational operator to compare any value to the null value is unknown. You cannot test for the null value using the = (equal) operator, because the null value is not equal to any value, including itself. The only way to test for the null value is to use the function [ISNULL](#) or [ISNULLS](#).

## Pattern Matching Operators

The pattern matching operator, [MATCH](#), and its synonym, [MATCHES](#), compare a string expression to a pattern. The syntax for a pattern match expression is:

```
string MATCH[ES] pattern
```

The pattern is a general description of the format of the string. It can consist of text or the special characters X, A, and N preceded by an integer used as a repeating factor. X stands for any characters, A stands for any alphabetic characters, and N stands for any numeric characters. For example, 3N is the pattern for character strings made up of three numeric characters. If the repeating factor is zero, any number of characters will match the string. For example, 0A is the pattern for any number of alphabetic characters, including none. If an NLS locale is defined, its associated definitions of *alphabetic* and *numeric* determine the pattern matching.

An empty string matches the following patterns: "0A", "0X", "0N", "...", "", "", or \\.

BASIC uses characters rather than bytes to determine string length. In NLS mode, MATCHES works in the same way for multibyte and single-byte character sets. For more information about NLS and [character sets](#), see *UniVerse NLS Guide*.

## IF Operator

The **IF** operator lets you indicate a value conditional upon the truth of another value. The IF operator has the following syntax:

*variable* = IF *expression* THEN *expression* ELSE *expression*

*variable* is assigned the value of the THEN expression if the IF expression is true, otherwise it is assigned the value of the ELSE expression. The IF operator is similar to the IF statement, but it can sometimes be more efficient.

## Logical Operators

Numeric data, string data, and the null value can function as logical data. Numeric and string data can have a logical value of true or false. The numeric value 0 (zero), is false; all other numeric values are true. Character string data other than an empty string is true; an empty string is false. The null value is neither true nor false. It has the special logical value of null.

Logical operators perform tests on logical expressions. Logical expressions that evaluate to 0 or an empty string are false. Logical expressions that evaluate to null are null. Expressions that evaluate to any other value are true.

The logical operators in UniVerse BASIC are:

- AND (or the equivalent &)
- OR (or the equivalent !)
- NOT

The NOT function inverts a logical value.

The operands of the logical operators are considered to be logical data types. Tables 2-3, 2-4, and 2-5 show logical operation results.

**Table 2-3. The AND Operator**

AND	TRUE	NULL	FALSE
TRUE	TRUE	NULL	FALSE
NULL	NULL	NULL	FALSE
FALSE	FALSE	FALSE	FALSE

**Table 2-4. The OR Operator**

OR	TRUE	NULL	FALSE
TRUE	TRUE	TRUE	TRUE
NULL	TRUE	NULL	NULL
FALSE	TRUE	NULL	FALSE

**Table 2-5. The NOT Operator**

NOT	
TRUE	FALSE
NULL	NULL
FALSE	TRUE

Arithmetic and relational operations take precedence over logical operations. UniVerse logical operations are evaluated from left to right (AND statements do not take precedence over OR statements).

**Note:** The logical value NULL takes the action of false, because the condition is not known to be true.

## Assignment Operators

Assignment operators are used in UniVerse BASIC assignment statements to assign values to variables. Table 2-6 shows the operators and their uses.

**Table 2-6. Assignment Operators**

Operator	Syntax	Description
=	<i>variable</i> = <i>expression</i>	Assigns the value of <i>expression</i> to <i>variable</i> without any other operation specified.
+=	<i>variable</i> += <i>expression</i>	Adds the value of <i>expression</i> to the previous value of <i>variable</i> and assigns the sum to <i>variable</i> .
--	<i>variable</i> -= <i>expression</i>	Subtracts the value of <i>expression</i> from the previous value of <i>variable</i> and assigns the difference to <i>variable</i> .
:=	<i>variable</i> := <i>expression</i>	Concatenates the previous value of <i>variable</i> and the value of <i>expression</i> to form a new value for <i>variable</i> .

Table 2-7 shows some examples of assignment statements.

**Table 2-7. Examples of Assignment Statements**

Example	Interpretation
X = 5	This statement assigns the value 5 to the <i>variable</i> X.
X += 5	This statement is equivalent to X=X+5. It adds 5 to the value of the <i>variable</i> X, changing the value of X to 10 if it was originally 5.
X -= 3	This statement is equivalent to X=X-3. It subtracts 3 from the value of the <i>variable</i> X, changing the value of X to 2 if it was originally 5.
X := Y	This statement is equivalent to X=X:Y. If the value of X is 'CON', and the value of Y is 'CATENATE', then the new value of the <i>variable</i> X is 'CONCATENATE'.

## Dynamic Array Operations

UniVerse BASIC provides a number of special functions that are designed to perform common operations on dynamic arrays.

## Vector Functions

Vector functions process lists of data rather than single values. By using the VEC.MATH (or V) option of the [SOPTIONS](#) statement, the arithmetic operators ( +, -, \*, / ) can also operate on dynamic arrays as lists of data.

The operations performed by vector functions or operators are essentially the same as those performed by some standard functions or operators. The difference is that standard functions process all variables as single-valued variables, treating delimiter characters as part of the data. On the other hand, vector functions recognize delimiter characters and process each field, value, and subvalue individually. In fact, vector functions process single-valued variables as if they were dynamic arrays with only the first value defined.

Vector functions have been implemented as subroutines for compatibility with existing UniVerse BASIC programs. Each subroutine is assigned a name made up of the function's name preceded by a hyphen. For example, the name of the subroutine that performs the [ADDS](#) function is -ADDS. Because the subroutines are cataloged globally, they can be accessed using the method described in the [CALL](#) statement.

The first column of Table 2-8 shows the functions for manipulating dynamic arrays that are available with UniVerse BASIC. The second column shows the corresponding instructions to use for single-valued variables. In this table, *m1* and *m2* represent dynamic arrays; *s1* and *s2* represent single-valued variables; *p1*, *p2*, and so on, represent single-valued parameters. The value of the function is the resulting dynamic array.

**Table 2-8. Vector Functions**

Vector Function	Corresponding Instruction for Single-Valued Field
<a href="#">ADDS</a> ( <i>m1</i> , <i>m2</i> )	<i>s1</i> + <i>s2</i>
<a href="#">ANDS</a> ( <i>m1</i> , <i>m2</i> )	<i>s1</i> AND <i>s2</i>
<a href="#">CATS</a> ( <i>m1</i> , <i>m2</i> )	<i>s1</i> : <i>s2</i>
<a href="#">CHARS</a> ( <i>m1</i> )	CHAR ( <i>s1</i> )
<a href="#">COUNTS</a> ( <i>m1</i> , <i>p1</i> )	COUNT ( <i>s1</i> , <i>p1</i> )
<a href="#">DIVS</a> ( <i>m1</i> , <i>m2</i> )	<i>s1</i> / <i>s2</i>
<a href="#">EQS</a> ( <i>m1</i> , <i>m2</i> )	<i>s1</i> EQ <i>s2</i>
<a href="#">ISNULLS</a> ( <i>m1</i> )	ISNULL ( <i>s1</i> )
<a href="#">NES</a> ( <i>m1</i> , <i>m2</i> )	<i>s1</i> NE <i>s2</i>

**Table 2-8. Vector Functions (Continued)**

<b>Vector Function</b>	<b>Corresponding Instruction for Single-Valued Field</b>
<b>LES</b> ( <i>m1, m2</i> )	<i>s1</i> LE <i>s2</i>
<b>LTS</b> ( <i>m1, m2</i> )	<i>s1</i> LT <i>s2</i>
<b>GES</b> ( <i>m1, m2</i> )	<i>s1</i> GE <i>s2</i>
<b>GTS</b> ( <i>m1, m2</i> )	<i>s1</i> GT <i>s2</i>
<b>NOTS</b> ( <i>m1</i> )	NOT ( <i>s1</i> )
<b>FIELDS</b> ( <i>m1, p1, p2, p3</i> )	FIELD ( <i>s1, p1, p2, p3</i> )
<b>FMTS</b> ( <i>m1, p1</i> )	FMT ( <i>s1, p1</i> )
<b>ICONVS</b> ( <i>m1, p1</i> )	ICONV ( <i>s1, p1</i> )
<b>IFS</b> ( <i>m1, m2, m3</i> )	IF <i>s1</i> THEN <i>s2</i> ELSE <i>s3</i>
<b>INDEXS</b> ( <i>m1, p1, p2</i> )	INDEX ( <i>s1, p1, p2</i> )
<b>LENS</b> ( <i>m1</i> )	LEN ( <i>s1</i> )
<b>MODS</b> ( <i>m1, m2</i> )	MOD ( <i>s1, s2</i> )
<b>MULS</b> ( <i>m1, m1</i> )	<i>s1</i> * <i>s2</i>
<b>NUMS</b> ( <i>m1</i> )	NUM ( <i>s1</i> )
<b>OCONVS</b> ( <i>m1, p1</i> )	OCONV ( <i>s1, p1</i> )
<b>ORS</b> ( <i>m1, m2</i> )	<i>s1</i> OR <i>s2</i>
<b>SEQS</b> ( <i>m1</i> )	SEQ ( <i>s1</i> )
<b>STRS</b> ( <i>m1, p1</i> )	STR ( <i>s1, p1</i> )
<b>SPACES</b> ( <i>m1</i> )	SPACE ( <i>s1</i> )
<b>SPLICE</b> ( <i>m1, p1, m2</i> )	<i>s1</i> : <i>p1</i> : <i>s2</i>
<b>SUBSTRINGS</b> ( <i>m1, p1, p2</i> )	<i>s1</i> [ <i>p1, p2</i> ]
<b>SUBS</b> ( <i>m1, m1</i> )	<i>s1</i> – <i>s2</i>
<b>TRIMS</b> ( <i>m1</i> )	TRIM ( <i>s1</i> )

When a function or operator processes two dynamic arrays, it processes the lists in parallel. In other words, the first value of field A and the first value of field B are processed together, then the second value of field A and the second value of field B are processed together, and so on.

Consider the following example:

```
A = 123V456S7890S2468V10F3691V33S12
B = 13V57S912F1234V8
$OPTIONS VEC.MATH
X = A + B
```

First, the function processing isolates the first field of each dynamic array, which can be written as:

```
A <1> = 123V456S7890S2468V10
B <1> = 13V57S912
```

Then the first values of the first fields are isolated:

```
A <1, 1> = 123
B <1, 1> = 13
```

Then the first subvalues of the first values of the first fields are isolated and added:

```
A <1, 1, 1> = 123
B <1, 1, 1> = 13
```

This produces the first subvalue of the first value of the first field of the result:

```
X <1, 1, 1> = 136
```

Since there are no more subvalues in the first value of either first field, the second values of the first fields are isolated:

```
A <1, 2> = 456S7890S2468
B <1, 2> = 57S912
```

The first subvalues of the second values of the first fields are isolated and added:

```
A <1, 2, 1> = 456
B <1, 2, 1> = 57
```

This produces the first subvalue of the second value of the first field of the result:

```
X <1, 2, 1> = 513
```

Next the subvalues:

```
A <1, 2, 2> = 7890
B <1, 2, 2> = 912
```

are isolated and added to produce:

```
X <1, 2, 2> = 8802
```



Then the subvalues:

A <1, 2, 3> = 2468

B <1, 2, 3> = ""

are isolated and added to produce:

X <1, 2, 3> = 2468

Since B <1, 2, 3> does not exist, it is equal to an empty string. In arithmetic expressions an empty string equals zero.

Since there are no more subvalues in either second value of the first fields, these values are isolated:

A <1, 3> = 10

B <1, 3> = ""

Then the subvalues:

A <1, 3, 1> = 10

B <1, 3, 1> = ""

are isolated and added to produce:

X <1, 3, 1> = 10

Since there are no more subvalues or values in either first field, the second fields of each dynamic array are isolated and the process repeats down to the subvalue levels. The second fields can be written as follows:

A <2> = 3691V33S12

B <2> = 1234V8

Then the first values of the second fields are isolated:

A <2, 1> = 3691

B <2, 1> = 1234

Then the first subvalues of the first values of the second fields are isolated and added:

A <2, 1, 1> = 3691

B <2, 1, 1> = 1234

This produces the first subvalue of the first value of the second field of the result:

X <2, 1, 1> = 4925

Then the second values of the second fields are isolated:

A <2, 2> = 33S12

B <2, 2> = 8

Then the first subvalues of the second values of the second fields are isolated and added:

```
A <2, 2, 1> = 33  
B <2, 2, 1> = 8
```

This produces the first subvalue of the second value of the second field of the result:

```
X <2, 2, 1> = 41
```

Then the second subvalues of the second values of the second fields are isolated and added:

```
A <2, 2, 2> = 12  
B <2, 2, 2> = ""
```

This produces the second subvalue of the second value of the second field of the result:

```
X <2, 2, 2> = 12
```

Since there are no more elements in either dynamic array, the result is:

```
X <1, 1, 1> = 136  
X <1, 2, 1> = 513  
X <1, 2, 2> = 8802  
X <1, 2, 3> = 2468  
X <1, 3, 1> = 10  
X <2, 1, 1> = 4925  
X <2, 2, 1> = 41  
X <1, 2, 2> = 12
```

These elements are put into the resultant dynamic array, separated by the delimiter mark corresponding to the highest levels that are different (for example, X<1,1,1> and X<1,2,1> have different value levels, so they are separated by a value mark). This yields the following:

```
X = 136V513S8802S2468V10F4925V41S12
```

## REUSE Function

If two dynamic arrays are processed by the vector functions described in the preceding section, and they contain unequal numbers of fields, values, or subvalues, then zeros or empty strings are added to the shorter list until the two lists are equal.

When you use the [REUSE](#) function, the last value in the shorter list is reused until all the elements in the longer list are exhausted or until the next higher delimiter is encountered.

## Dynamic Array Operations and the Null Value

In all dynamic array operations an array reference to a null value treats the null value as an unknown structure of the least bounding delimiter level. For example, the **extract operator** (**< >**) extracts the requested data element from a dynamic array. The result of extracting any element from the null value itself is also the null value. If the requested dynamic array element is the stored representation of the null value (CHAR(128)), the null value is returned.

Consider the following three cases:

```
X is λ
Y = ^128
Z = ^128vA
```

X is the null value, Y is a dynamic array containing only the character used to represent the null value (CHAR(128)), and Z is a dynamic array containing two values, CHAR(128) and A, separated by a value mark.

If you extract all or part of the dynamic array from X, you get the null value in all cases:

```
X<1> is λ
X<2> is λ
X<1,2> is λ
```

But if you extract all or part of the dynamic array from Y or Z, you get the null value only when the extract operator specifically references that element of the array:

```
Y<1> is λ
Y<2> = " "
Y<1,2> is λ

Z<1> = ^128vA
Z<2> = " "
Z<1,1> is λ
Z<1,2> = A
```

When the dynamic array extraction finds a string made up of only CHAR(128) between two system delimiters or between a system delimiter and an end-of-string character, CHAR(128) is converted to the null value before any other operation is performed.

See the **EXTRACT**, **INSERT**, **REPLACE**, **DELETE**, **REMOVE**, and **REUSE** functions, and the **INS**, **DEL**, and **REMOVE** statements for details about how BASIC dynamic array operations handle the null value.



# 3

## Compiling BASIC Programs

Before you can run a BASIC program, you must compile it with the UniVerse BASIC compiler. The compiler takes your source code as input and produces executable object code.

Use the UniVerse command [CREATE.FILE](#) to create a type 1 or type 19 file in which to store the source code of your BASIC programs. You can create and edit the source code with an operating system editor (such as *vi*), the UniVerse Editor, or a combination of the two.

### The BASIC Command

To compile a BASIC program, enter the [BASIC](#) command at the system prompt in the following syntax:

```
BASIC filename [ program | * ] ... [ options ]
```

*filename* is the name of the type 1 or type 19 file containing the BASIC programs to be compiled. You can compile more than one program at a time if you put all the programs in the same file.

### Compiling Programs in the Background

Use the [PHANTOM](#) command to compile BASIC programs in the background. The output from PHANTOM processes is stored in the file named &PH&. For example, the command:

```
>PHANTOM BASIC BP *
```

compiles all the programs in BP and puts the output in a record named `BASIC_tttt_ddd` in the &PH& file (*tttt* and *ddd* are a time and date stamp).

## BASIC Options

You can use the following options with the BASIC command:

<b>+<i>\$option</i></b>	Turns on the specified \$OPTIONS option, or defines a UniVerse flavor.
<b>–<i>\$option</i></b>	Turns off the specified \$OPTIONS option or UniVerse flavor.
<b>–I</b>	Suppresses execution of <b>RAID</b> or <b>VLIST</b> on a compiler or a BASIC program.
<b>–LIST or –L</b>	Generates a listing of the program.
<b>–XREF or –X</b>	Generates a cross-reference table of statement labels and variable names used in the program.
<b>–SPOOL or –S</b>	Generates a listing of the program and spools it directly to the printer rather than to a file.
<b>–T</b>	Suppresses the symbol and line number tables that are usually appended to the end of the object file, for run-time error messages.

A listing produced with either the **–LIST** or the **–XREF** option is saved in a file whose name is made up of the source filename and a suffixed **.L** . The record ID of the program listing in the listing file (*filename.L*) is the same as the record ID in the program file (*filename*).

### The +\$ Option

The **+\$** option specifies the \$OPTIONS options you want to turn on, or the flavor of UniVerse you want the program to use. See the **\$OPTIONS** statement for the list of options and UniVerse flavors. You must specify all options you want to turn on before the options you want to turn off.

### The –\$ Option

The **–\$** option specifies the \$OPTIONS options, or the UniVerse flavor, you want to turn off. See the **\$OPTIONS** statement for the list of options and UniVerse flavors. You must specify all options you want to turn off after the options you want to turn on.

### The –I Option

The **–I** option inhibits the execution of **RAID** or **VLIST** on your BASIC program. This lets you bypass subroutines already debugged and provides security to your subroutines.

## The -LIST Option

Listings produced with the -LIST option contain all source lines and all lines inserted with the **SINSERT** or **SINCLUDE** statements. The listing is saved in a file whose name is made up of the source filename and a suffixed .L. The record ID of the program listing in the listing file is the same as the record ID in the program file.

## The -XREF Option

The -XREF option produces an alphabetical cross-reference listing of every label and variable name used in the program. The listing is saved in a file whose name is made up of the source filename and a suffixed .L. The record ID of the program listing in the listing file is the same as the record ID in the program file.

Consider the following example:

```
>BASIC BP DATE.INT -XREF
Compiling: Source = 'PB/DATE.INT', Object = 'BP.O/DATE.NT'

Compilation Complete.
>ED BP.L DATE.INT
13 lines long.

----: P
0001: BP.L/DATE.INT Source Listing
0002:
0003:
0004: Cross Reference Listing
0005:
0006: Variable..... Type..... References.....
.....
0007:
0008: DATE                Local Scalar    0003=    0004
0009:
0010: * Definition of symbol
0011: = Assignment of variable
0012: ! Dimension
0013: @ Argument to CALL
```

The listing shows three columns: Variable, Type, and References. Variable is the name of the variable or symbol. Type is one of the following symbol types:

Local Scalar

Local Array

Common Scalar

Common Array

Argument	Variable used in SUBROUTINE statement
Array Arg	Variable used in MAT clause
@variable	One of the system @variables
Label	A program label, as in GOTO FOO
Named Common	Name of a named common segment
Predefined EQU	Predefined equate like @FM, @VM, etc.
Equate	User-defined equate

References shows the numbers of all lines in the program that refer to the symbol. Each line number can have a symbol after it to indicate what the line contains:

*	Definition of symbol
=	Assignment of variable
!	Dimension of array
@	Argument to CALL statement

### The –SPOOL Option

The –SPOOL option lets you direct output to a printer rather than to a file. Program listings can be spooled for printing with this option.

The –SPOOL option is useful when the listing is very long and you need to look at different parts of the program simultaneously.

### The –T Option

The –T option suppresses the table of symbols and the table of line numbers that are appended to the end of object files. These tables are used for handling run-time error messages. Suppressing them results in somewhat smaller object files, but run-time error messages do not know the line number or variable involved in the error.

## Compiler Directives

Compiler directives are BASIC statements that direct the behavior of the compiler. Functions performed by compiler directives include: inserting source code from one program into another program during compilation, setting compile-time compatibility with another UniVerse flavor, and specifying a condition for compiling certain parts of a program. Most compiler directive statements are prefixed by a dollar sign ( \$ ).



## Including Other Programs

Two statements, **\$INCLUDE** (synonyms are **#INCLUDE** and **INCLUDE**) and **\$CHAIN**, instruct the compiler to include the source code of another program in the program currently being compiled. **\$INCLUDE** inserts other code in your program during compilation, returning afterward to compile the next statement in your program. **\$CHAIN** also inserts the code of another program, but after doing so it does not continue reading from the original file. Any program statements following a **\$CHAIN** statement are not compiled.

The syntax for both statements is as follows:

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

```
$CHAIN [ filename ] program
```

If you do not specify *filename*, the included program must be in the same file as the program you are compiling.

If *program* is in a different file, you must specify *filename* in the **\$INCLUDE** statement. *filename* must be defined in the VOC file.

The **\$INSERT** statement is included for compatibility with Prime INFORMATION programs. **\$INSERT** is used, like **\$INCLUDE**, to insert other code in your program during compilation, returning afterward to compile the next statement in your program.

The syntax for the **\$INSERT** statement is as follows:

```
$INSERT primos.pathname
```

The PRIMOS pathname is converted to a valid filename using the following conversion rules:

/	converts to	?\<
?	converts to	??
An ASCII NUL	converts to	?0
An initial .	converts to	?.

Any leading **\*>** is ignored. If a full pathname is specified, the **>** between directory names changes to a **/** to yield the following:

```
[ pathname/ ] program
```

**\$INSERT** uses the transformed argument directly as a filename of the file containing the source to be inserted. It does not use the VOC file.

## Defining and Removing Identifiers

You can define and remove identifiers with the `$DEFINE` and `$UNDEFINE` statements. `$DEFINE` defines an identifier that controls program compilation. You can also use it to replace the text of an identifier. `$UNDEFINE` removes the definition of an identifier. You can use the identifier to control conditional compilation.

Use the `$UNDEFINE` statement to remove an identifier defined by a previous `$DEFINE` statement from the symbol table. You can also specify conditional compilation with the `$UNDEFINE` statement.

## Specifying Flavor Compatibility

A `$OPTIONS` statement is a compiler directive used to specify compile-time emulation of any UniVerse flavor. By default the settings are the same as the flavor of the account. A program can also specify individual options, overriding the usual setting. This does not allow object code that has been compiled in one flavor to execute in another. It allows only the emulation of capabilities of one flavor from within another flavor.

## 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
- `$IFDEF`
- `$IFNDEF`

Conditional compilation with the modified `IF` statement is useful for customizing large programs that are to be used by more than one kind of user. It can also reduce the size of the object code and increase program efficiency.

You can use the compiler directives `$IFDEF` and `$IFNDEF` to control whether or not sections of a program are compiled. Both of these compiler directives test a given identifier to see if it is currently defined (that is, has appeared in a `$DEFINE` compiler directive and has not been undefined). If the identifier that appears in a `$IFDEF` is defined, all the program source lines appearing between the `$IFDEF` compiler directive and the closing `$ENDIF` compiler directive are compiled. If the identifier is not defined, all the lines between the `$IFDEF` compiler directive and the `$ENDIF` compiler directive are ignored.

The `$IFNDEF` compiler directive is the complement to the `$IFDEF` compiler directive. The lines following the `$IFNDEF` compiler directive are included in the compilation if the identifier is *not* defined. If the identifier is defined, all lines

between the \$IFDEF compiler directive and the \$ENDIF compiler directive are ignored. \$IFDEF and \$IFDEF compiler directives can be nested up to 10 deep.

## IF Statements

The syntax of the conditional compilation statement is the same as that of the **IF** statement with the exception of the test expression, which must be one of the following: \$TRUE, \$T, \$FALSE, or \$F. The syntaxes are as follows:

```
IF $TRUE THEN statements ELSE statements
```

```
IF $T THEN statements ELSE statements
```

```
IF $FALSE THEN statements ELSE statements
```

```
IF $F THEN statements ELSE statements
```

The conditional compilation statement can specify a variable name rather than one of the test specifications listed previously. If it does, an **EQUATE** statement equating the variable to the test specification must appear at the beginning of the source code. For example:

```
EQUATE USER.A LIT "$T"  
IF USER.A THEN statements ELSE statements
```

Consider a program that contains debugging statements in its source code. Using the conditional compilation statement, you could create two versions of the object code from the same source: a test version that contains the debugging statements, and a release version without the debugging statements. The following steps produce the two required versions of the program:

1. Include a conditional debugging statement throughout the source code:

```
IF TEST PRINT X,Y
```

2. Put the following statement in the source code before any conditional statements:

```
EQUATE TEST LIT "$TRUE"
```

3. Compile the source code to produce object code that contains the debugging statements (the test version).

4. Change the EQUATE statement to:

```
EQUATE TEST LIT "$FALSE"
```

5. Compile the source code to produce object code that does not contain the debugging statements (the release version).

## The \$IFDEF Compiler Directive

**\$IFDEF** uses the \$IF...\$ELSE...\$ENDIF variation of conditional syntax. \$IFDEF tests for the definition of a compile-time symbol. If it is defined and the \$ELSE clause is omitted, the statements between \$IFDEF and \$ENDIF are compiled, otherwise they are ignored. If the \$ELSE clause is included, only the statements between \$IFDEF and \$ELSE are compiled.

If the compile-time symbol is not defined and the \$ELSE clause is included, only the statements between \$ELSE and \$ENDIF are compiled.

The \$ELSE compiler directive introduces the alternative clause of an \$IFDEF compiler directive. The \$ENDIF compiler directive marks the end of a conditional compilation block.

In the following example, *identifier* is not defined so the statements following the \$ELSE compiler directive are compiled. All the statements up to the \$ENDIF compiler directive are compiled.

```
$DEFINE identifier
.
.
.
$UNDEFINE identifier
$IFDEF identifier
    [ statements ]
$ELSE
    [ statements ]
$ENDIF
```

## The \$IFNDEF Compiler Directive

**\$IFNDEF** tests for the definition of a compile-time symbol. If it is defined and the \$ELSE clause is omitted, the statements between \$IFNDEF and \$ENDIF are ignored, otherwise they are compiled. If the \$ELSE clause is included, only the statements between \$ELSE and \$ENDIF are compiled.

If the compile-time symbol is not defined and the \$ELSE clause is included, only the statements between \$IFNDEF and \$ELSE are compiled.

The \$ELSE compiler directive introduces the alternative clause of an \$IFNDEF compiler directive. The \$ENDIF compiler directive marks the end of a conditional compilation block.

In the following example the `$IFDEF` compiler directive determines that *identifier* is defined so that the compiler is forced to compile the statements following the `$ELSE` compiler directive:

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

## Warnings and Error Messages

As the compiler attempts to compile a program, various warnings and error messages may appear, disclosing problems that exist in the source code (e.g., statement format errors). When an error occurs, compilation aborts. All errors must be corrected before compilation is successful. Warning messages do not stop compilation.

During compilation, the compiler displays an asterisk on the screen for every 10 lines of source code successfully compiled. You can monitor the progress of the compilation process by counting the number of asterisks on the screen at any time. If an error is encountered, a question mark ( ? ) rather than the asterisk ( \* ) is displayed for those 10 lines.

## Successful Compilation

When all errors in the source code are corrected, the compiler successfully completes the compilation by producing an object code record. The object code record is stored in a file whose name is made up of the source filename suffixed with `.O` (*sourcename.O*). The object code record ID is the same as the source file record ID (program name).

For example, if source code record MAIN is stored in a file called BP, executing the following compile statement:

```
>BASIC BP MAIN
```

compiles the source code in record MAIN in file BP, producing object code that is stored in record MAIN in file BP.O . The compiler creates the object code file if it does not exist.

# The RUN Command

After you have successfully compiled your program, you can run it from the UniVerse system level with the RUN command. Enter the RUN command at the system prompt. The syntax is as follows:

**RUN** [ *filename* ] *program* [ *options* ]

The RUN command appends .O to *filename* and executes the record containing object code in *filename.O* . If *filename* is omitted, the default file, BP, is assumed.

*program* is the name of the source code of the program. *options* can be one or more of the following:

NO.WARN	Suppresses all warning (nonfatal) messages.
NO.PAGE	Turns off automatic paging. Programs that position the cursor with @ functions do not need to disable pagination.
LPTR	Spools program output to the printer rather than to the terminal screen.
KEEP.COMMON	If the program is executed from within a chain, links the unnamed common.
TRAP	Causes the program to enter the interactive debugger, RAID, whenever a nonfatal error occurs.

For example, the following command executes the record MAIN in the BP.O file:

```
>RUN BP MAIN
```

Run-time error messages are printed on the terminal screen as they are encountered, unless the NO.WARN keyword was specified.

**Note:** Cataloged programs are considered executable files (that is, the RUN command is not required). To run a cataloged program, enter its [catalog](#) name at the system prompt.

## Cataloging a BASIC Program

You must catalog a BASIC program in order to:

- Use the program as a subroutine in an I-descriptor
- Execute the program without using the RUN command

## Catalog Space

There are three ways to catalog a program: locally, normally (standard), and globally. Each has different implications. There is no one best way of cataloging.

### Local Cataloging

Local cataloging creates a VOC entry for the program. This entry is a verb that points to the file and record containing the object code for the cataloged program. A locally cataloged program can be accessed only from the account in which it was cataloged, unless you copy the VOC entry for the catalog name to another account.

Since cataloging a program locally only creates a VOC entry pointing to the object file, you need not recatalog the program every time you recompile it.

### Normal Cataloging

Normal cataloging copies the specified object record to the system catalog space and gives it a name of the form:

*\*account\*catalog.name*

Normal cataloging also creates a VOC entry for the program. This entry is a verb that contains the name *\*account\*catalog* in field 2. A normally cataloged program can be accessed only from the account in which it was cataloged, unless you copy the VOC entry for the catalog name to another account or specify the full catalog name, including the account prefix.

Since cataloging a program normally copies the object code to the system catalog space, you must recatalog the program every time you recompile it.

### Global Cataloging

Global cataloging copies the specified object record into the system catalog space and gives it a name in one of the following forms:

*\*catalog.name*  
*–catalog.name*  
*\$catalog.name*  
*!catalog.name*

VOC entries are not created for globally cataloged programs. They are available to all accounts on the system as soon as they are cataloged. The UniVerse command processor looks in the system catalog space for verbs or external subroutines that have an initial \*. The run machine looks in the system catalog space for verbs or subroutines whose names begin with \*, –, \$, or !.

Because cataloging a program globally copies the object code to the system catalog space, you must recatalog the program every time you recompile it.

**Note:** Because the command processor interprets any line beginning with an asterisk and containing blanks as a comment, you cannot use command parameters when you invoke a globally cataloged program. That is, you can use the following command to run the globally catalog program \*GLOBAL, but you cannot include arguments in the command line:

```
>*GLOBAL
```

## The CATALOG Command

The CATALOG command is used to catalog a compiled BASIC program, either locally or in the system catalog space. The syntax of the CATALOG command is as follows:

```
CATALOG [ filename ] [ [ catalog.name ] program.name | * ] [ options ]
```

If you simply enter CATALOG at the system prompt, you are prompted for the argument values, one at a time:

```
>CATALOG
Catalog name or LOCAL =LOCAL
File name              =BP
Program name           =MONTHLY.SALES
```

If you press **Return** at any of the prompts, CATALOG terminates without cataloging anything. FORCE or NOXREF cannot be specified at a prompt. You can specify the LOCAL keyword in response to the prompt for *catalog.name*.

If you do not specify *catalog.name*, CATALOG uses the program name as the catalog name.

## Deleting Cataloged Programs

There are two commands for removing a program from the shared catalog space: DELETE.CATALOG and DECATALOG.

### DELETE.CATALOG

The DELETE.CATALOG command removes locally, normally, or globally cataloged programs. It has the following syntax:

```
DELETE.CATALOG catalog.name
```



*catalog.name* is used to determine if the program is either globally or normally cataloged. If it is, the program is removed from the system catalog. If the program is not in the system catalog, the VOC file is searched for a local catalog entry. If the program is locally cataloged, only the VOC entry is deleted, not the object code.

If a program is running when you try to delete it, the deletion does not take effect until the program terminates.

## DECATALOG

The DECATALOG command removes a locally cataloged program. It deletes the object code and removes the catalog entry from the user's VOC file. It has the following syntax:

```
DECATALOG [ filename [ [ program ] ]
```

This command deletes the object code of *program* from the “.O” portion of *file-name*. Use an asterisk ( *\** ) in place of *program* to indicate all records in the file. This command can also be executed after building a select list of programs to be decataloged.

## Catalog Shared Memory

UniVerse lets you load BASIC programs from the system catalog into shared memory and run them from there. This reduces the amount of memory needed for multiple users to run the same program at the same time. The program also starts a little faster since it is already in memory and does not have to be read from a disk file.

For example, if 21 users are running the same BASIC program at the same time without catalog shared memory, and the program code requires 50 kilobytes of memory, the total amount of memory used by everyone running that program is 21\*50, or 1050, kilobytes. On the other hand, if the program is loaded into catalog shared memory, all 21 users can run one copy of the program, which uses only 50 kilobytes of memory. In this example, catalog shared memory saves 1000 kilobytes, or one megabyte, of memory.

Before users can have access to programs running from catalog shared memory, the system administrator must explicitly choose the programs and load them into memory.



# 4

## Locks, Transactions, and Isolation Levels

This chapter describes the UniVerse BASIC mechanisms that prevent lost updates and other problems caused by data conflicts among concurrent users:

- Locks
- Transactions
- Isolation levels

### Locks

UniVerse locks control access to records and files among concurrent users. To provide this control, UniVerse supports the following two levels of lock granularity:

- Fine granularity (record locks)
- Coarse granularity (file locks)

The level at which you acquire a lock is known as *granularity*. Record locks affect a smaller component (the record) and provide a fine level of granularity, whereas file locks affect a larger component (the file) and provide a coarse level of granularity.

Lock *compatibility* determines what your process can access when other processes have locks on records or files. Record locks allow more compatibility because they coexist with other record locks, thus allowing more transactions to take place concurrently. However, these “finer-grained” locks provide a lower isolation level. File locks enforce a higher isolation level, providing more concurrency control but less compatibility.

Lock compatibility decreases and isolation level increases as strength and granularity increase. This can increase the possibility of deadlocks at high isolation levels. Within each granularity level, the strength of the lock can vary. UniVerse supports the following locks in order of increasing strength:

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

The locks become less compatible as the granularity, strength, and number of locks increase. Therefore the number of lock conflicts increase, and fewer users can access the records and files concurrently. To maximize concurrency, you should acquire the minimum lock required to perform a BASIC statement for the shortest period of time. The lock can always be promoted to a lock of greater strength or escalated to a coarser level of granularity if needed.

## Shared Record Lock

This lock is also called a READL lock, and is displayed as RL in the [LIST.READU](#) output. The shared record lock affects other users as follows:

Lets other users acquire:	Prevents other users from acquiring:	Is ignored if you already own:
Shared record lock	Update record lock	Shared record lock
Shared file lock	Exclusive file lock	Update record lock
Intent file lock		Shared file lock Intent file lock Exclusive file lock

The shared record lock can be promoted or escalated as follows:

Promoted to...	If...
Update record lock	No shared record locks are owned by another user No shared file locks are owned by another user No intent file locks are owned by another user
Escalated to...	If...
Shared file lock	No intent file locks are owned by another user No update record locks are owned by another user
Intent file lock	No intent file locks are owned by another user All update record locks are owned by you

Escalated to...	If...
Exclusive file lock	No intent file locks are owned by another user All shared and update record locks are owned by you

In UniVerse BASIC a shared record lock can be acquired with a [MATREADL](#), [READL](#), [READVL](#), or [RECORDLOCKL](#) statement, and released with a [CLOSE](#), [RELEASE](#), or [STOP](#) statement.

## Update Record Lock

This lock is also called a READU lock, and is displayed as RU in the [LIST.READU](#) output. The update record lock affects other users as follows:

Lets other users acquire:	Prevents other users from acquiring:	Is ignored if you already own:
No locks	Shared record lock Update record lock Shared file lock Intent file lock Exclusive file lock	Update record lock Exclusive file lock

**Note:** An update record lock you own is incompatible with a shared file lock you own. Be sure to use a LOCKED clause to avoid deadlocks.

The update record lock can be escalated as follows:

Escalated to...	If...
Intent file lock	All update record locks are owned by you
Exclusive file lock	All shared and update record locks are owned by you

In UniVerse BASIC an update record lock can be acquired or escalated from a shared record lock with a [MATREADU](#), [READU](#), [READVU](#), or [RECORDLOCKU](#) statement, and released with a [CLOSE](#), [DELETE](#), [MATWRITE](#), [RELEASE](#), [STOP](#), [WRITE](#), or [WRITEV](#) statement.

## Shared File Lock

This lock is displayed as FS in the [LIST.READU](#) output. The shared file lock affects other users as follows:

Lets other users acquire:	Prevents other users from acquiring:	Is ignored if you already own:
Shared record lock	Update record lock	Shared file lock
Shared file lock	Intent file lock Exclusive file lock	Intent file lock Exclusive file lock

**Note:** A shared file lock you own is incompatible with an update record lock you own. Be sure to use a LOCKED clause to avoid deadlocks.

The shared file lock can be promoted as follows:

Promoted to...	If...
Intent file lock	No shared file locks are owned by another user
Exclusive file lock	No shared file or record locks are owned by another user

In UniVerse BASIC a shared file lock can be acquired or promoted with a [FILE-LOCK](#) statement and released with a [CLOSE](#), [FILEUNLOCK](#), [RELEASE](#), or [STOP](#) statement.

## Intent File Lock

This lock is displayed as IX in the [LIST.READU](#) output. The intent file lock affects other users as follows:

Lets other users acquire:	Prevents other users from acquiring:	Is ignored if you already own:
Shared record lock	Update record lock Shared file lock Intent file lock Exclusive file lock	Intent file lock Exclusive file lock

The intent file lock can be promoted as follows:

Promoted to...	If...
Exclusive file lock	No shared record locks are owned by another user

In UniVerse BASIC, an intent file lock can be acquired or promoted from a shared file lock with a [FILELOCK](#) statement, and released with a [CLOSE](#), [FILEUNLOCK](#), [RELEASE](#), or [STOP](#) statement.

### Exclusive File Lock

This lock is displayed as FX in the [LIST.READU](#) output. The exclusive file lock affects other users as follows:

Lets other users acquire:	Prevents other users from acquiring:	Is ignored if you already own:
No locks	Shared record lock Update record lock Shared file lock Intent file lock Exclusive file lock	Exclusive file lock

In UniVerse BASIC an exclusive file lock can be acquired from a shared file lock with a [FILELOCK](#) statement and released with a [CLOSE](#), [FILEUNLOCK](#), [RELEASE](#), or [STOP](#) statement.

### Deadlocks

Deadlocks occur when two users who acquire locks incrementally try to acquire a lock that the other user owns, and the existing lock is incompatible with the requested lock. The following situations can lead to deadlocks:

- Lock promotion from a shared record or shared file lock to a stronger lock
- Lock escalation to file locks when two users try to escalate at the same time

You can configure UniVerse to automatically identify and resolve deadlocks as they occur through the Deadlock Daemon Administration menu, or you can manually fix a deadlock by selecting and aborting one of the deadlocked user processes. You use the deadlock daemon *uvdlockd* to identify and resolve deadlocks. For more information, see *Administering UniVerse*.

### Transactions

A *transaction* is a group of logically related operations on the database. In a transaction either the entire sequence of operations or nothing at all is applied to the database. For example, a banking transaction that involves the transfer of funds from one account to another involves two logically related operations: a with-

drawal and a deposit. Both operations, or neither, must be performed if the accounts concerned are to remain reconciled.

## Active Transactions

UniVerse supports *nested* transactions. Any transaction can include:

- Read and write operations
- Other transactions or subtransactions that can contain other operations or other transactions

When a transaction begins, it is active. If a second transaction begins before the first transaction is committed or rolled back, the new (child) transaction becomes the active transaction while the first (parent) transaction continues to exist but inactively. The child transaction remains active until:

- It is committed or rolled back, when the parent transaction becomes active again
- Another transaction (child) begins and becomes the active transaction

Only one transaction can be active at any time, although many transactions can exist concurrently. Only one transaction can exist at each transaction nesting level. The top-level transaction is at nesting level 1. When no transactions exist, the nesting level is 0.

## Transactions and Data Visibility

Transactions let you safeguard database files by caching database operations in the active transaction until the top-level transaction is committed. At this point the cached operations are applied to the database files, and other users can see the result of the transaction.

When a read operation for a record occurs, the most recent image of the record is returned to the user. This image is retrieved from the active transaction cache. If it is not found there, the parent transactions are then searched. Finally, if it is not found in the parent transactions, the image is retrieved from the database file.

When a child transaction is committed, the operations are adopted by the parent transaction. When it is rolled back, the operations are discarded and do not affect the database or the parent transaction.



## Transaction Properties

Each transaction must possess properties commonly referred to as the ACID properties:

- Atomicity
- Consistency
- Isolation
- Durability

In nested transactions, child transactions exhibit atomicity, consistency, and isolation properties. They do not exhibit the durability property since the parent adopts its operation when it is committed. The operations affect the database only when the top-level transaction is committed. Therefore, even though a child transaction is committed, its operations and locks can be lost if the parent transaction is rolled back.

### Atomicity

Either all the actions of a transaction occur successfully or the transaction is nullified by rolling back all operations. The transaction system ensures that all operations performed by a successfully committed transaction are reflected in the database, and the effects of a failed transaction are completely undone.

### Consistency

A transaction moves the database from one valid state to another valid state, and if the transaction is prematurely terminated, the database is returned to its previous valid state.

### Isolation

The actions carried out by a transaction cannot become visible to another transaction until the transaction is committed. Also, a transaction should not be affected by the actions of other concurrent transactions. UniVerse provides different isolation levels among concurrently executing transactions.

### Durability

Once a transaction completes successfully, its effects cannot be altered without running a compensating [transaction](#). The changes made by a successful transaction survive subsequent failures of the system.

## Serializability

In addition to the ACID properties, SQL standards stipulate that transactions be serializable. *Serializability* means that the effects of a set of concurrent transactions should produce the same results as though the individual transactions were executed in a serial order, and as if each transaction had exclusive use of the system. In UniVerse, serializability can be achieved by using isolation level 4 for all transactions.

## Transactions and Locks

Locks acquired either before a transaction exists or outside the active transaction are inherited by the active transaction. Locks acquired or promoted within a transaction are not released. Instead they adhere to the following behavior:

- Locks acquired or promoted within a child transaction are adopted by the parent transaction when the child is committed.
- Locks acquired within a child transaction are released when the child transaction is rolled back.
- Locks promoted within a child transaction are demoted to the level they were before the start of the child transaction when the child is rolled back.
- All locks acquired, promoted, or adopted from child transactions are released when the top-level transaction is committed or is rolled back.

## Transactions and Isolation Levels

If you do not specify an isolation level for the top-level transaction, the UniVerse default isolation level 0 is used. You can change this default by using the **BASIC SET TRANSACTION ISOLATION LEVEL** statement.

If an isolation level is not specified for a child transaction, the isolation level is inherited from the parent transaction. A specified isolation level should be the same or higher than the parent's isolation level. Isolation levels that are lower than the parent transaction's isolation level are ignored, and the parent's isolation level is used. This occurs because a child transaction's operations must take place under the protection of the higher isolation level so that when it merges with the parent transaction, its data is still valid.

## Using Transactions in BASIC

The compiler enforces the creation of well-formed transactions. A well-formed transaction occurs when data is locked before it is accessed. A BASIC transaction must include the following three statements:

- **BEGIN TRANSACTION**
- At least one **COMMIT** or **ROLLBACK** statement
- **END TRANSACTION**

If one of these statements is omitted or out of order, the program does not compile.

The run machine also enforces the use of well-formed transactions. A transaction starts when **BEGIN TRANSACTION** is executed and ends when **COMMIT** or **ROLLBACK** is executed. Program execution then continues at the statement following the next **END TRANSACTION** statement.

If UniVerse is running with transaction logging active, a fatal run-time error can occur if the log daemon is not running. If transaction logging is not active on your system, no transactions are logged even if the log daemon is running. To activate transaction logging, set the **TXMODE** configurable parameter to 1; to deactivate transaction logging, set **TXMODE** to 0. For more information about [transaction logging](#), see *UniVerse Transaction Logging and Recovery*.

The following example shows transactions in a BASIC program:

```
BEGIN TRANSACTION ISOLATION LEVEL 1
* acquire locks and execute database operations
  BEGIN TRANSACTION ISOLATION LEVEL 4
  * acquire locks and execute database operations
    BEGIN TRANSACTION ISOLATION LEVEL 3
    COMMIT
    END TRANSACTION
    BEGIN TRANSACTION ISOLATION LEVEL 0
    * acquire locks and execute database operations
    ROLLBACK
    END TRANSACTION
  COMMIT
END TRANSACTION
COMMIT
END TRANSACTION
```

## @Variables

You can use the following [@variables](#) to track transaction activity:

- @ISOLATION
- @TRANSACTION
- @TRANSACTION.ID
- @TRANSACTION.LEVEL

@ISOLATION indicates the current transaction isolation level (0, 1, 2, 3, or 4) for the active transaction, or the current default isolation level if no transaction exists.

@TRANSACTION is a numeric value that indicates transaction activity. Any nonzero value indicates that a transaction is active. 0 indicates that no active transaction exists.

@TRANSACTION.ID indicates the transaction number of the active transaction. An empty string indicates that no transaction exists.

@TRANSACTION.LEVEL indicates the transaction nesting level of the active transaction. 0 indicates that no transaction exists.

## Transaction Restrictions

Other than memory and disk space, there is no restriction on the number of nesting levels in transactions. However, it is important to remember that having many levels in a transaction affects system performance.

If in a transaction you try to write to a remote file over UV/Net, the WRITE statement fails, the transaction is rolled back, the program terminates with a run-time error message.

You cannot use the following statements while a transaction is active. Doing so causes a fatal error.

- [CLEARFILE](#)
- [SET TRANSACTION ISOLATION LEVEL](#)

You cannot use the EXECUTE or PERFORM statements in a transaction to execute most UniVerse commands and SQL statements. However, you can use [EXECUTE](#)

and **PERFORM** to execute the following UniVerse commands and SQL statements within a transaction:

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	
GET.LIST	SAVE.LIST	SORT.LABEL	

If a BASIC statement in a transaction has an ON ERROR clause and a fatal error occurs, the ON ERROR clause is ignored.

## Isolation Levels

Setting a transaction's isolation level helps avoid various data anomalies. UniVerse BASIC lets you set different isolation levels depending on which data anomalies you want to avoid. Your transaction runs at the specified isolation level because the transaction subsystem verifies that you have acquired the required locks for that isolation level. If you have not done so, the program fails.

You can specify isolation levels with the following BASIC statements:

- **BEGIN TRANSACTION**
- **SET TRANSACTION ISOLATION LEVEL**

You can use the LOGIN entry in the VOC file to set the isolation level for a session. Do this by including a SET.SQL command that sets the isolation level. This sets the default isolation level for all transactions that occur during that session, including UniVerse commands and SQL statements. For example, the program might include the statement SET.SQL ISOLATION 2 to set the isolation level to 2 each time a user logs in to the account. This affects all SQL statements and BASIC transactions that occur during this session.

## Isolation Level Types

UniVerse BASIC provides the following types of isolation level:

Level	Type
0	NO.ISOLATION
1	READ.UNCOMMITTED

Level	Type
2	READ.COMMITTED
3	REPEATABLE.READ
4	SERIALIZABLE

Each level provides a different degree of protection against the anomalies described in the following section. Only level 4 provides true serializability. However, due to performance issues, for a large system we recommend that you use level 2 rather than 3 or 4 for most programs. UniVerse provides a default of 0 for backward compatibility.

## Data Anomalies

Isolation levels provide protection against the following data anomalies or conflicts that can occur when two processes concurrently access the data:

- Lost updates occur when two processes try to update an object at the same time. For example, Process A reads a record. Process B reads the same record, adds 10, and rewrites it. Process A adds 20 to the record that it previously read, and rewrites the record. Thus, Process B's update is lost.
- Dirty reads occur when one process modifies a record and a second process reads the record before the first is committed. If the first process terminates and is rolled back, the second process has read data that does not exist.
- Nonrepeatable reads occur when a process is unable to ensure repeatable reads. For example, this can occur if a transaction reads a record, another transaction updates it, then the first transaction rereads it, and gets a different value the second time.
- Phantom writes occur when a transaction selects a set of records based on selection criteria and another process writes a record that meets those criteria. The first process repeats the same selection and gets a different set of records.

Table 4-1 lists the data anomalies and the isolation levels at which they can occur.

**Table 4-1. Levels at Which Anomalies Can Occur**

Anomaly	Level 0	Level 1	Level 2	Level 3	Level 4
Lost update	No <sup>1</sup>	No	No	No	No
Dirty read	Yes	Yes	No	No	No

**Table 4-1. Levels at Which Anomalies Can Occur (Continued)**

Anomaly	Level 0	Level 1	Level 2	Level 3	Level 4
Nonrepeatable read	Yes	Yes	Yes	No	No
Phantom write	Yes	Yes	Yes	Yes	No

1. Lost updates cannot occur if ISOMODE is set to 2 or 1. If ISOMODE is 0, it is possible for a process running at isolation level 0 to cause a lost update in your process.

## Using the ISOMODE Configurable Parameter

The ISOMODE parameter controls the minimum locking requirements for each UniVerse system. By enforcing a minimum level of locking, the transaction management subsystem guarantees that no transaction suffers a lost update due to the actions of another transaction. Protection against lost updates is an important property of serializability. You can set ISOMODE to one of the following settings:

Setting	Description
0	Provides backward compatibility. Transactions are not required to use well-formed writes.
1	Enforces well-formed writes in BASIC transactions. This is the default.
2	Enforces well-formed writes in BASIC programs, whether or not they are in a transaction.

The default ISOMODE setting 1 ensures that BASIC transactions obey the locking rules for isolation level 1, as described in Table 4-2. This means a record cannot be written or deleted in a transaction unless the record or file is locked for update. A write or delete of a locked record is known as a well-formed write.

ISOMODE setting 0 provides compatibility with earlier UniVerse releases that did not enforce the requirement for well-formed writes in transactions. Since transactions should always use well-formed writes, we recommend that you modify any transactions that do not follow this rule as soon as possible, so that you can set ISOMODE to 1.

Setting ISOMODE to 2 enforces all writes and deletes in BASIC to be well-formed. This mode is available so that when converting an application to use transactions, you can determine whether any programs have not yet been converted. You should not use ISOMODE 2 permanently since many UniVerse system programs are not (and need not be) transactional.

## Isolation Levels and Locks

In transactions you must consider the level of isolation you need to perform the task, since UniVerse uses locks to ensure that the isolation levels are achieved. As the isolation level increases, the granularity of the locks required becomes coarser. Therefore the compatibility of locks as well as the number of concurrent users accessing the records and files decreases.

The BASIC run machine checks that the user has acquired the necessary locks to perform a BASIC statement. If the minimum locks for the current isolation level are not held by the user, a fatal error results.

The minimum locks required in a transaction to achieve isolation levels that ensure successful file operation are listed in Table 4-2.

**Table 4-2. Isolation Level Locking Requirements**

Operation	Isolation Level	Minimum Lock
Read	0 NO.ISOLATION	None
	1 READ.UNCOMMITTED	None
	2 READ.COMMITTED	Shared record lock (RL)
	3 REPEATABLE.READ	Shared record lock (RL)
	4 SERIALIZABLE	Shared file lock (FS)
Delete or Write	0 NO.ISOLATION <sup>1</sup>	None, or update record lock (RU) <sup>1</sup>
	1 READ.UNCOMMITTED	Update record lock (RU)
	2 READ.COMMITTED	Update record lock (RU)
	3 REPEATABLE.READ	Update record lock (RU)
	4 SERIALIZABLE	Update record lock (RU) and intent file lock (IX)
Select	4 SERIALIZABLE	Intent file lock (IX)

1. Different ISOMODE settings affect the locking rules for isolation level 0.

In the SQL environment UniVerse automatically acquires the locks it needs to perform SQL DML (data manipulation language) statements. Lock escalation from record locks to file locks occurs if the number of record locks acquired or



promoted within a file in a transaction associated with the SQL DML statements exceeds the value of the configurable parameter [MAXRLOCK](#) (the default value is 100). This ensures that the record lock tables do not fill to capacity with large multirow statements.

## Example

The following example illustrates how isolation levels affect transactions. A transaction running at isolation level 2 deletes records for Customer 100 from the file CUST. The transaction scans the file ORDERS for all orders placed by this customer and deletes each order. The part of the transaction that deletes the orders does not want to lock the ORDERS file unnecessarily.

The following program illustrates how lock escalation takes place:

```
OPEN "CUST" TO CUST ELSE
    STOP "Cannot open CUST file"
END
OPEN "ORDERS" TO ORDERS ELSE
    CLOSE CUST
    STOP "Cannot open ORDERS file"
END
LOCK.COUNT = 0

** escalate record locks into file locks
** when 10 records have been locked
LOCK.ESCALATE = 10
BEGIN TRANSACTION ISOLATION LEVEL 2
    READU CUST.REC FROM CUST,100 THEN
        SELECT ORDERS
GET.NEXT.RECORD:
    LOOP
        WHILE READNEXT ORDERS.NO DO
            ** if lock escalation limit has not been met
            ** obtain a shared record lock for the order
            IF LOCK.COUNT < LOCK.ESCALATE THEN
                READL ORDERS.REC FROM ORDERS,ORDERS.NO ELSE
                    GOTO GET.NEXT.RECORD:
            END
            LOCK.COUNT = LOCK.COUNT + 1
        END ELSE
            ** if lock escalation limit has been reached
            ** obtain intent file lock since the file
            ** needs to be updated
            IF LOCK.COUNT = LOCK.ESCALATE THEN
                FILELOCK ORDERS,"INTENT"
            END
            READ ORDERS.REC FROM ORDERS,ORDERS.NO ELSE
```

```

        GOTO GET.NEXT.RECORD:
    END
END
IF ORDERS.REC<1> = 100 THEN
    IF LOCK.COUNT < LOCK.ESCALATE THEN
        ** promote shared record lock to
        ** an exclusive record lock
        READU ORDERS.REC FROM ORDERS,ORDERS.NO THEN NULL
    END ELSE
        ** promote intent file lock to
        ** an exclusive file lock
        IF LOCK.COUNT = LOCK.ESCALATE THEN
            FILELOCK ORDERS,"EXCLUSIVE"
        END
    END
    DELETE ORDERS,ORDERS.NO
END
REPEAT
    DELETE CUST,100
END
COMMIT
END TRANSACTION
CLOSE CUST
CLOSE ORDERS
END

```

# 5

## Debugging Tools

UniVerse provides two debugging tools: RAID and VLIST. RAID is an interactive debugger. VLIST is a diagnostic tool that lists source code followed by object code, as well as statistics about your program.

**Note:** You cannot run RAID or VLIST on programs compiled with the `-I` option.

### RAID

You can use RAID with your UniVerse BASIC programs. RAID is both an object code and a source code debugger—a powerful tool for detecting errors in UniVerse BASIC code. RAID lets you do the following:

- Set and delete breakpoints. You can suspend execution of the program at specified lines by setting breakpoints in the source code. Once RAID suspends program execution, you can examine the source code, change or display variable values, set additional breakpoints, or delete breakpoints.
- Set watchpoints. You can keep track of changes to variable values by setting watchpoints in the source code. When a variable's value changes, RAID can print both the old and new values and the source instruction that caused the change.
- Step through and display source code, line by line or in segments.
- Examine object addresses.
- Display and modify variables.
- Display all the elements of an array in succession.

You can invoke RAID from the command processor, from within a BASIC program, or by pressing the **Break** key while your BASIC program is executing.

## Invoking RAID from the Command Processor

To invoke RAID from the command processor, enter the RAID command instead of the RUN command. The syntax for invoking RAID from the command processor is as follows:

**RAID** [ *filename* ] *program* [ *options* ]

*filename* is the name of the file in which the source code is stored. RAID appends “.O” to *filename* in order to locate and operate on the object code. If you do not specify *filename*, RAID assumes the BP file by default.

*program* is the name of the record containing the source code of the program.

*options* can be one or more of the following:

NO.WARN	Suppresses all warning (nonfatal) error messages. If you do not specify NO.WARN, run-time error messages are printed on the terminal screen as they are encountered.
NO.PAGE	Turns off automatic paging. Programs that position the cursor with @ functions need not disable pagination.
LPTR	Spools program output to the printer rather than to the terminal.
KEEP.COMMON	Maintains the value of variables in unnamed common if a CHAIN statement passes control to another BASIC program.
TRAP	Causes RAID to be reentered whenever a nonfatal error occurs.

Use RAID the same way you use RUN. This causes RAID to be invoked just before program execution. For example, the following command executes the file BP.O/MAIN using the RAID debugger:

```
>RAID BP MAIN
```

When you invoke RAID from the command processor, RAID displays the first executable source code instruction, followed by a double colon ( :: ). Enter a RAID command at the :: prompt. To run the program, enter **R** at the :: prompt. To quit RAID, enter **Q**. RAID commands are discussed in detail in [“RAID Commands”](#) on page 5-4.

## Invoking RAID from a BASIC Program

To invoke RAID from a program, include the **DEBUG** statement in the program. The syntax is as follows:

```
DEBUG
```

The **DEBUG** statement takes no arguments. When the run machine encounters this statement, the program stops execution, displays a double colon ( :: ), and prompts you to enter a RAID command.

You can also enter the debugger while a BASIC program is running by pressing the **Break** key and then selecting the break option D.

## Invoking RAID Using the Break Key

To invoke RAID using the **Break** or **Intr** key, press the **Break** key during execution and then select the break option D.

## Referencing Variables Through RAID

Enter variable names as they appear in the BASIC source program. They are case-sensitive, so “A” is not the same variable as “a”.

In addition to regular variable names, you can reference some special “register” variables with RAID. UniVerse BASIC object code is executed by the run machine. When you assign a new variable, the run machine allocates memory for that variable and creates a pointer to that memory location. There are special variables that the run machine uses to hold temporary information. These are the “registers” that can be referenced by \$R0 through \$Rn, and a matrix address variable referenced by \$MATRIX. An arbitrary number of these registers is available, depending on the needs of your program. The appropriate amount is always made available. You never have more than you need.

**Note:** Unreferenced variables are not carried in the symbol table of BASIC object code. Therefore, RAID can only display the contents of variables referenced in the current subroutine. RAID ignores all unreferenced variables, and treats them as unknown symbols.

Registers hold intermediate values in a program. For example, for the following statement the sum of 3 and 4 is evaluated and placed in \$R0:

```
A=B : 3+4
```

The object code evaluates the statement as:

```
A=B:$R0
```

The \$MATRIX variable is sometimes used as a pointer to a specific element of an array. It avoids the need to locate the element more than once in the same statement. For example, in the REMOVE statement, the following statement allows for successive system-delimited substrings in the third element of array B to be put in variable A and a delimiter code setting put in variable C:

```
REMOVE A FROM B(3) SETTING C
```

The reference to the third element of array B is put in the \$MATRIX pointer. The object code evaluates the statement as follows:

```
REMOVE A FROM $MATRIX SETTING C
```

## RAID Commands

You can enter any RAID command from the double colon ( :: ) prompt. RAID commands have the following general syntax:

*position command qualifier*

<i>position</i>	Tells where and how often to execute the RAID command in the program. You can provide one of the following:
<i>line</i>	The decimal number of a line of the source code.
<i>address</i>	The hexadecimal address of an object code instruction, indicated by a leading 0X.
<i>procedure</i>	The name of a procedure in the source code.
<i>variable</i>	The name of a variable in the source code. You must specify the variable exactly as it appears in the source code. Variable names are case-sensitive, so “A” is not the same as “a”. Subscript <i>variable</i> to indicate an element of an array. For example, A[1,2].
<i>n</i>	Indicates the number of times to execute the command.
<i>qualifier</i>	Can be either of the following:
<i>string</i>	A string of characters to search for or to replace the value of a variable.
<i>*</i>	Indicates a special form of the specified command.

Table 5-1 summarizes the RAID commands.

**Table 5-1. RAID Commands**

Command	Description
<i>line</i>	Displays the specified line of the source code.
<i>/[string]</i>	Searches the source code for <i>string</i> .
B	Sets a RAID breakpoint.
C	Continues program execution.
D	Deletes a RAID breakpoint.
G	Goes to a line or address, and continues program execution.
H	Displays statistics for the program.
I	Displays and executes the next object code instruction.
L	Displays the next line to be executed.
M	Sets watchpoints.
Q	Quits RAID.
R	Runs the program.
S	Steps through the BASIC source code.
T	Displays the call stack trace.
V	Enters verbose mode for the M command.
V*	Prints the compiler version that generated the object code.
W	Displays the current window.
X	Displays the current object code instruction and address.
X*	Displays local run machine registers and variables.
Z	Displays the next 10 lines of source code.
\$	Turns on instruction counting.
#	Turns on program timing.
+	Increments the current line.
-	Decrements the current line.
.	Displays object code instruction and address before execution.
<i>variable/</i>	Prints the value of <i>variable</i> .
<i>variable!string</i>	Changes the value of <i>variable</i> to <i>string</i> .

## **line: Displaying Source Code Lines**

*line* displays a line of the source code specified by its line number. Note that this command displays but does not change the current executable line.

## **/ : Searching for a Substring**

Use a slash followed by a string to search the source code for the next occurrence of the substring *string*. The syntax is as follows:

`/[string]`

The search begins at the line following the current line. If *string* is found, RAID sets the current line to the line containing *string* and displays that line. If you issue the `/` command without specifying *string*, RAID searches for the last-searched string. This is an empty string if you have not yet specified a string during this RAID session. If RAID encounters the end of file without finding *string*, it starts at the first line and continues the search until it finds *string* or returns to the current line.

## **B: Setting Breakpoints**

Use the B command to set or list RAID breakpoints. There are two syntaxes:

`[ address | line | procedure [ : line ] ] :B`

`B*`

You can set a RAID breakpoint at the current line, an object code address, a BASIC source line number, the beginning of a specified procedure, or a BASIC source line number within a specified procedure. RAID recognizes lines in called subroutines. RAID executes the program up to the breakpoint and then stops and issues the `::` prompt. At that point you can issue another RAID command.

The following example sets breakpoints at line 30 and line 60 of the source code, then runs the program. The program stops executing and displays the RAID prompt when it reaches line 30, and again when it reaches line 60.

```
:: 30B
:: 60B
::R
```

The `B*` command lists all currently active breakpoints.

## **C: Continuing Program Execution**

Use the C command to continue program execution until RAID encounters a breakpoint, or until completion. The C command turns off verbose mode (use the



V command to enter verbose mode). If the TRAP command line option is used, RAID is entered at every nonfatal error.

## **D: Deleting Breakpoints**

Use the D command to delete RAID breakpoints. There are two syntaxes:

`[ address | line ] D`

`D*`

You can delete a RAID breakpoint at the current line, an object code address, or a BASIC source line number. If you use the \* option, this command deletes all breakpoints. Some BASIC statements produce multiple run machine statements. If you delete a breakpoint by line number, RAID tries to match the starting address of the BASIC source number. A breakpoint set at anything other than the starting address of a BASIC source line must be deleted by its address.

## **G: Continuing Program Execution from a Specified Place**

Use the G command to go to the line or address specified and to execute the program from that point. The syntax is as follows:

`address | line G`

## **H: Displaying Program Information**

Use the H command to display the version of BASIC used to compile the program, the number of constants used, the number of local variables used, the number of subroutine arguments passed, the object code size, and what procedure RAID was in when the program failed (main program versus subroutine).

## **I: Executing the Next Object Code Instruction**

Use the I command to display and execute the next object code instruction. The syntax is as follows:

`[ n ] I`

If you use the *n* option, RAID displays and executes the next *n* instructions.

## **L: Displaying the Next Line**

Use the L command to display the next line to be executed.

## M: Setting Watchpoints

Use the M command to set watchpoints. The syntaxes are as follows:

*variable* M [ ; [ *variable* ] M ... ]

*variable* =VALUE M

*variable* is a variable found in the symbol table.

VALUE is the value that you want to break.

The second syntax lets you set a watchpoint for a variable set to a specific value.

A watchpoint condition occurs when RAID monitors a variable until the variable's value changes. The program then suspends operation and displays the variable's old value, its new value, and the source instruction that caused the change to occur. If no change occurs, no display appears. This command accepts up to eight variables. To continue monitoring a variable after RAID has already displayed a change to that variable's value, you need only enter M again, not *variable* M.

## Q: Quitting RAID

Use the Q command to quit RAID.

## R: Running the Program

Use the R command to run the program until RAID encounters a breakpoint, or until completion. The R command is the same as the C command. The R command turns off verbose mode (use the V command to enter verbose mode). If you specify the TRAP command line option, RAID is entered at every nonfatal error.

## S: Stepping Through the Source Code

Use the S command to execute the current line and display the next line of source code. Use multiple S commands to step through the program. The syntax is as follows:

[ *n* ] S [ \* ]

If the line includes a subroutine call, RAID steps into the subroutine. If you use the *n* option, RAID steps through the next *n* lines. If you use the \* option, RAID steps around any subroutine call, essentially treating the entire subroutine as a single line. That is, the S\* command instructs RAID to display and execute a

source line. If the line includes a subroutine call, RAID executes the subroutine and displays the first source line occurring after the subroutine returns.

### **T: Displaying the Call Stack Trace**

Use the T command to display the call stack trace. It displays the names of the routines that have been called up to the current subroutine.

### **V: Entering Verbose Mode**

Use the V command to enter verbose mode for the M command. In verbose mode RAID displays every source code line until a watchpointed variable's value changes. The program then breaks and displays the variable's old value, its new value, and the source code line that caused the change. To use this command, you must follow it with an M command:

```
: :V  
: : variable M
```

The verbose mode remains on until turned off. Use the C, R, or S command to turn off this mode.

### **V\*: Printing the Compiler Version**

Use the V\* command to print the version of the compiler that generated the BASIC object code.

### **W: Displaying the Current Window**

Use the W command to display the current window. The syntax is as follows:

```
[ line ] W
```

A window comprises a group of 10 lines of source code, centered around the current line. For example, if the current line is 4 when you issue the W command, RAID displays the first 10 lines. The W command by itself does not change the current line. If you use the line option, RAID changes the current line to *line* and displays a window centered around that line. For example, if the current line is 14, RAID displays the lines 9–18. Note that this command affects only the current display lines; it does not affect the executable lines. That is, if you are stepping through the code using the S command, RAID considers the current display line to be the last line displayed before issuing the first S command.

### **X: Displaying the Current Object Code Instruction**

Use the X command to display but not execute the current object code instruction and address.

## X\*: Displaying the Local Run Machine Registers

Use the X command to display the contents of the local run machine registers (if any exist) and list run machine variables. The syntaxes are as follows:

X [ \* ]

X!

The second syntax lets you set a variable to the empty string. The value of X! is displayed with a carriage return immediately following.

RAID displays the contents. The run machine variables are the following:

Inmat	Value set by the <a href="#">INMAT</a> function
Col1	Value set by the <a href="#">COL1</a> function
Col2	Value set by the <a href="#">COL2</a> function
Tab	Value set by the <a href="#">TABSTOP</a> statement
Precision	Value set by the <a href="#">PRECISION</a> statement
Printer	Printer channel, set by the <a href="#">PRINTER</a> statement
Psw	Value set of the last internal comparison
Lsw	Value set of the last lock test
Status	Value set by the <a href="#">STATUS</a> function

## Z: Displaying Source Code

Use the Z command to display the next 10 lines of source code and establish the last line displayed as the current line. The syntax is as follows:

[ *line* ] Z

For example, if the current line is 4 when you issue the Z command, RAID displays lines 4–13, and line 13 becomes the current line.

The current window changes each time this command is used, since the last line printed becomes the current line. For example, if the current line is 14 when you issue the Z command, RAID displays lines 14–23. The current line becomes 23.

If you use the line option, the current line becomes *line*, and RAID displays a window with the specified line first. Regardless of the syntax used, the last line printed becomes the current line once you issue this command. Note that this command affects only the current display lines. It does not affect the executable lines.

## **\$ : Turning On Instruction Counting**

Use the \$ command to turn on instruction counting. RAID records the object code instructions used by the program and the number of times each instruction was executed into a record in your &UFD& file called *profile*. The instruction counting stops when RAID encounters the next break point, a **DEBUG** statement, **Ctrl-C**, or the end of the program.

## **# : Turning On Program Timing**

Use the # command to turn on program timing. RAID records the program name, the number of times it is executed, and the total elapsed time spent during execution into a record of your &UFD& file called *timings*. The program timing stops when RAID encounters the next break point, a **DEBUG** statement, **Ctrl-C**, or the end of the program.

## **+ : Incrementing the Current Line Number**

Use the + command to increment and display the current object code line number. The syntax is as follows:

[ *n* ] +

If you use the *n* option, RAID adds *n* to the current line. However, the command only displays valid lines. For example, if you start at line 4 and use the command 3+, the command moves to line 7 only if line 7 is a valid line for the code. If line 7 is invalid, RAID moves to the first valid line after line 7. Note that this command does not affect the currently executed line. That is, if you are stepping through the code using the S command, RAID considers the current line to be the line you displayed before issuing the first S command.

## **– : Decrementing the Current Line Number**

Use the – command to decrement and display the current line number. The syntax is as follows:

[ *n* ] –

If you use the *n* option, RAID subtracts *n* from the current line. However, the command only displays valid lines. For example, if you start at line 14 and use the command 3–, the command moves to line 11 only if line 11 is a valid address for the code. If line 11 is invalid, RAID moves backward to the first valid address before address 11.

## **. : Displaying the Next Instruction To Be Executed**

Use the . (period) command to display the next object code instruction and address to be executed.

## ***variable/* : Printing the Value of a Variable**

Use a forward slash (/) after a variable name to print the value of the variable. The syntax is as follows:

*variable/*

When you enter the *variable/* command, RAID displays the variable's value, and indicates whether it is a string, a number, or the null value. To print an array element, you must subscript the variable. If you enter just an array name, RAID displays the length of the X and Y coordinates. Display matrix values by explicitly displaying the first element, then press **Return** to display subsequent elements. For example, after displaying an array element, pressing **Return** displays successive elements in the array (i.e., 1,1 1,2 1,3 2,1 2,2 2,3, etc.). After the last element in the array displays, the indices wrap to display 0,0 and then 1,1.

## **! : Changing the Value of a Variable**

Use the ! (exclamation point) command to change the value of *variable* to *string*. The syntax is as follows:

*variable!string*

To change an array element, you must subscript the variable. This option is not available for debugging programs in shared memory.

## **VLIST**

Use VLIST to display a listing of the object code of a BASIC program. The syntax for VLIST is as follows:

**VLIST** [*filename*] *program* [**R**]

- |                 |   |
|-----------------|---|
| <i>filename</i> | The name of the file containing the source code of the BASIC program. The default filename is BP.         |
| <i>program</i>  | The name of the program to list.  |
| <b>R</b>        | Displays internal reference numbers for variables and constants rather than source code names and values. |

VLIST displays each line of source code followed by the lines of object code it generated. VLIST also displays program statistics.

```
>VLIST BP TO.LIST
Main Program "BP.O/TO.LIST"
Compiler Version: 7.3.1.1
Object Level      : 5
Machine Type      : 1
Local Variables   : 1
Subroutine args    : 0
Unnamed Common    : 0
Named Common Seg: 0
Object Size       : 34
Source lines      : 4

0001: FOR I = 1 TO 10
0001 0000 : 0F8 move          0 => I
0001 0006 : 098 forincr      I 10 1 0020:

0002: PRINT I
0002 0014 : 130 printcrLf    I

0003: NEXT I
0003 001A : 0C2 jump         0006:

0004: END
0004 0020 : 190 stop
```

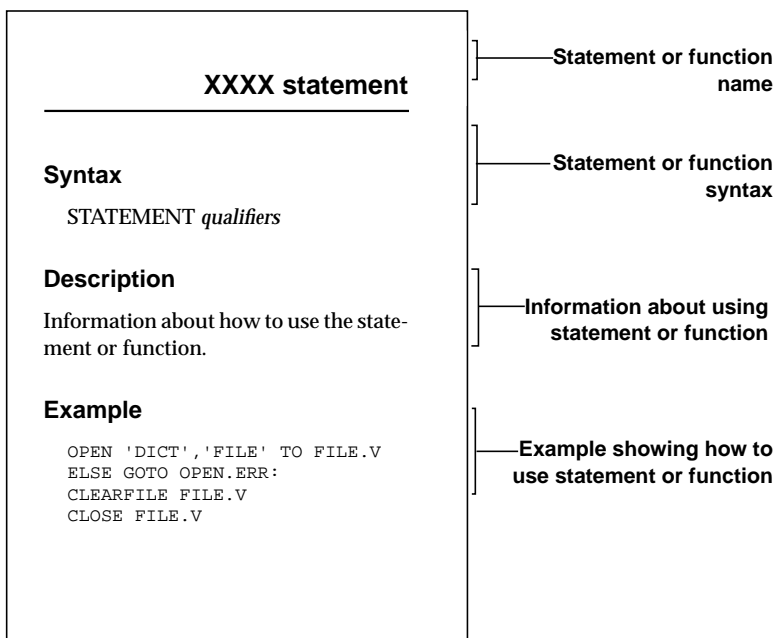




# 6

## BASIC Statements and Functions

This chapter describes the UniVerse BASIC statements and functions. Each statement and function is listed on a separate page. The sample shows a typical statement or function reference page.



## ! statement

---

### Syntax

`! [comment.text]`

### Description

Use the ! 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 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.

### 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
```

### Syntax

#INCLUDE [*filename*] *program*

#INCLUDE *program* FROM *filename*

### Description

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 filename must be specified in the #INCLUDE statement, followed by the name of the program. The filename 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 page). 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
```

## \$\* statement

---

### Syntax

`$* [comment.text]`

### Description

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

A comment must be a separate 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 \$\*.

### 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
```

### Syntax

`$CHAIN [filename] program`

### Description

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](#), #INCLUDE, and INCLUDE statements 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 filename, 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.

### Example

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

When this program is compiled, the \$CHAIN statement inserts code from the program END (see the example on the [END](#) statement page). 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

---

### Syntax

`$COPYRIGHT "copyright.notice"`

### Description

Use the \$COPYRIGHT statement to specify copyright information in 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

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

### Description

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

- Defining an identifier
- Supplying replacement text for an identifier

*identifier* is the symbol to be defined. It can be any valid identifier.

*replacement.text* is 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* may cause unexpected program behavior.

The [\\$UNDEFINE](#) statement removes the definition of an identifier.

## \$DEFINE statement

---

### Conditional Compilation

You can use \$DEFINE with the [\\$IFDEF](#) 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.

### 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**

---

### **Syntax**

`$EJECT`

### **Description**

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

This statement is a synonym for the \$PAGE statement.

## \$IFDEF statement

---

### Syntax

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

### Description

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.

\$IFDEF and [\\$IFNDEF](#) statements can be nested up to 10 deep.

### 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
```

### Syntax

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

### Description

Use the \$IFDEF statement to test for the definition of a compile-time symbol. The \$IFDEF statement complements the [\\$IFDEF](#) statement.

If *identifier* is currently not 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 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.

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

### Example

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

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

## \$INCLUDE statement

---

### Syntax

`$INCLUDE [filename] program`

`$INCLUDE program FROM filename`

### Description

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 filename must be specified in the \$INCLUDE statement, followed by the name of the program. The filename 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 page). 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
```

### Syntax

`$INSERT primos.pathname`

### Description

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 \$INSERT and [\\$INCLUDE](#) (and its synonyms #INCLUDE and INCLUDE) is that \$INSERT takes a PRIMOS pathname as an argument, whereas \$INCLUDE takes a UniVerse filename and record ID. The PRIMOS pathname is converted to a pathname; any leading \*> is ignored.

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

If *primos.pathname* is the name of the program only, it is interpreted as a relative pathname. 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 pathname using the following 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 pathname, the > between directory names changes to a / to yield:

`[pathname/] program`

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

## \$INSERT statement

---

### Example

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

When this program is compiled, the \$INSERT statement inserts code from the program END (see the example on the [END](#) statement page). 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
```

### Syntax

\$MAP *mapname*

### Description

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.

*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 *UniVerse 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

---

### Syntax

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

### Description

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.

Use the following keywords to specify *flavor*:

Keyword	Flavor
PICK	Generic Pick emulation
INFORMATION	Prime INFORMATION emulation
REALITY	REALITY emulation
IN2	Intertechnique emulation
DEFAULT	IDEAL UniVerse
PIOPEN	PI/open emulation

For instance, the following statement instructs the compiler to treat all 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 statement

---

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

**Options for the \$OPTIONS Statement**

Option Name	Option Letter	Description
CASE	<i>none</i>	Differentiates between uppercase and lowercase identifiers and keywords.
COMP.PRECISION	<i>none</i>	Rounds the number at the current precision value in any comparison.
COUNT.OVLP	O	For <b>INDEX</b> and <b>COUNT</b> functions, the count overlaps.
END.WARN	R	Prints a warning message if there is no final END statement.
EXEC.EQ.PERF	P	Compiles <b>EXECUTE</b> as <b>PERFORM</b> .
EXTRA.DELIM	W	For <b>INSERT</b> and <b>REPLACE</b> functions, 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	<i>none</i>	Lets output conversion codes be used as format masks (see the <b>FMT</b> function).
FSELECT	<i>none</i>	Makes the <b>SELECT</b> statement 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	<i>none</i>	Specifies the PIOPEN flavor for the I and P options to the <b>HEADING</b> and <b>FOOTING</b> keywords. This is the default for the PIOPEN flavor.

## \$OPTIONS statement

---

Options for the \$OPTIONS Statement (Continued)

Option Name	Option Letter	Description
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	<b>HEADING</b> statement causes initial page eject.
IN2.SUBSTR	T	Uses IN2 definitions for BASIC substring handling ( <i>string[n,m]</i> ). 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	<b>ABORT</b> syntax follows Prime INFORMATION instead of PICK.
INFO.CONVERT	<i>none</i>	Specifies that the FMT, ICONV, and OCONV functions perform PI/open style conversions.
INFO.ENTER	<i>none</i>	Specifies the PIOPEN flavor of the ENTER statement.
INFO.INCLUDE	<i>none</i>	Processes any PRIMOS pathnames specified with the <b>\$INSERT</b> statement.
INFO.LOCATE	L	<b>LOCATE</b> syntax follows Prime INFORMATION instead of REALITY. The Pick format of the LOCATE statement is always supported in all flavors.
INFO.MARKS	<i>none</i>	Specifies that the LOWER, RAISE, and REMOVE functions use a smaller range of delimiters for PI/open compatibility.
INFO.MOD	<i>none</i>	Specifies the PIOPEN flavor for the MOD function. This is the default for the PIOPEN flavor.

## \$OPTIONS statement

---

Options for the \$OPTIONS Statement (Continued)

Option Name	Option Letter	Description
INPUTAT	<i>none</i>	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 <a href="#">INPUT</a> statement.
INT.PRECISION	<i>none</i>	Rounds the integer at the current precision value in an <a href="#">INT</a> function.
LOCATE.R83	<i>none</i>	A <a href="#">LOCATE</a> statement returns an “AR” or “DR” sequence value compatible with Pick, Prime INFORMATION, and PI/open systems.
NO.CASE	<i>none</i>	Does not differentiate between uppercase and lowercase in identifiers or keywords. This is the default for the PIOPEN flavor.
NO.RESELECT	U	For <a href="#">SELECT</a> and <a href="#">SSELECT</a> statements, active select list 0 remains active; another selection or sort is not performed. The next <a href="#">READ-NEXT</a> statement uses select list 0.
ONGO.RANGE	G	If the value used in an <a href="#">ON...GOTO</a> or <a href="#">ON...GOSUB</a> is out of range, executes the next statement rather than the first or last branch.
PCLOSE.ALL	Z	The <a href="#">PRINTER CLOSE</a> statement closes all print channels.
PERF.EQ.EXEC	C	<a href="#">PERFORM</a> compiles as <a href="#">EXECUTE</a> .
PIOPEN.EXECUTE	<i>none</i>	EXECUTE behaves similarly to the way it does on PI/open systems.
PIOPEN.INCLUDE	<i>none</i>	Processes any PRIMOS pathnames specified with the <a href="#">\$INSERT</a> and <a href="#">\$INCLUDE</a> statements.

## \$OPTIONS statement

---

Options for the \$OPTIONS Statement (Continued)

Option Name	Option Letter	Description
PIOPEN.MATREAD	<i>none</i>	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	<i>none</i>	In the <a href="#">SELECTINDEX</a> statement, removes multiple occurrences of the same record ID in an index with a multivalued field.
RADIANS	<i>none</i>	Calculates trigonometric operations using radians instead of degrees.
RAW.OUTPUT	<i>none</i>	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 a <a href="#">READ</a> , <a href="#">READU</a> , <a href="#">READV</a> , <a href="#">READVL</a> , or <a href="#">READVU</a> fails, 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 ( <i>string[n,m]</i> ). If <i>m</i> or <i>n</i> 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 href="#">READNEXT</a> returns an exploded select list.
SEQ.255	N	<a href="#">SEQ</a> (" ") = 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 href="#">STOP</a> and <a href="#">ABORT</a> to use the ERRMSG file to produce error messages instead of using the specified text.
SUPP.DATA.ECHO	I	Causes input statements to suppress echo from data.

## \$OPTIONS statement

### Options for the \$OPTIONS Statement (Continued)

Option Name	Option Letter	Description
TIME.MILLISECOND	<i>none</i>	Causes the <a href="#">SYSTEM</a> (12) function to return the current system time in milliseconds, and the <a href="#">TIME</a> function to return the current system time in seconds.
ULT.FORMAT	<i>none</i>	Format operations are compatible with Ult/ix. For example, FMT("", "MR2") returns an empty string, not 0.00.
USE.ERRMSG	B	<a href="#">PRINTERR</a> prints error messages from ERRMSG.
VAR.SELECT	S	<a href="#">SELECT TO variable</a> creates a local select variable instead of using numbered select lists, and <a href="#">READLIST</a> 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 single-valued arithmetic.
WIDE.IF	<i>none</i>	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:

### Override Versions for Statements

Statement	Equal to...
ABORTE	<a href="#">ABORT</a> with \$OPTIONS STOP.MSG
ABORTM	ABORT with \$OPTIONS –STOP.MSG
HEADINGE	<a href="#">HEADING</a> with \$OPTIONS HEADER.EJECT
HEADINGN	HEADING with \$OPTIONS –HEADER.EJECT
SELECTV	<a href="#">SELECT</a> with \$OPTIONS VAR.SELECT

## \$OPTIONS statement

---

### Override Versions for Statements

Statement	Equal to...
SELECTN	SELECT with \$OPTIONS –VAR.SELECT
STOPE	<b>STOP</b> with \$OPTIONS STOP.MSG
STOPM	STOP with \$OPTIONS –STOP.MSG

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

### Default Settings of \$OPTIONS Options

	IDEAL	PICK	INFO.	REALITY	IN2	PIOPEN
CASE			✓			
COMP.PRECISION						
COUNT.OVLP		✓		✓	✓	
END.WARN			✓	✓		✓
EXEC.EQ.PERF			✓			✓
EXTRA.DELIM			✓		✓	✓
FOR.INCR.BEF	✓	✓		✓	✓	
FORMAT.OCONV				✓		
FSELECT						
HEADER.BRK						✓
HEADER.DATE			✓			✓
HEADER.EJECT			✓			✓
IN2.SUBSTR			✓		✓	✓
INFO.ABORT						✓
INFO.CONVERT						
INFO.ENTER						✓
INFO.LOCATE			✓			✓
INFO.MARKS						✓
INFO.MOD						✓
INPUTAT						✓
INPUT.ELSE		✓	✓			

## \$OPTIONS statement

**Default Settings of \$OPTIONS Options (Continued)**

	IDEAL	PICK	INFO.	REALITY	IN2	PIOPEN
INT.PRECISION						
LOCATE.R83						
NO.CASE						✓
NO.RESELECT		✓	✓		✓	✓
NO.SMA.COMMON						
ONGO.RANGE		✓			✓	
PCLOSE.ALL		✓		✓	✓	
PERF.EQ.EXEC				✓		✓
PIOPEN.EXECUTE						
PIOPEN.INCLUDE						✓
PIOPEN.MATREAD						
PIOPEN.SELIDX						✓
RADIANS					✓	
RAW.OUTPUT						
READ.RETAIN		✓		✓	✓	
REAL.SUBSTR			✓	✓		✓
RNEXT.EXPL			✓			
SEQ.255		✓		✓	✓	
STATIC.DIM		✓		✓	✓	
STOP.MSG		✓		✓	✓	
SUPP.DATA.ECHO		✓		✓	✓	
ULT.FORMAT						
USE.ERRMSG				✓		
VAR.SELECT		✓		✓	✓	
VEC.MATH			✓			✓
WIDE.IF	✓	✓	✓	✓	✓	

### Example

```
>ED BP OPT
4 lines long.
```

## \$OPTIONS statement

---

```
----: 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 [SEJECT](#) statement.

## \$UNDEFINE statement

---

### Syntax

\$UNDEFINE *identifier*

### Description

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.

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

You can use \$UNDEFINE with the [\\$IFDEF](#) 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.

## Syntax

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

## Description

Use the **\*** 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 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.

## 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.

```
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
```

## < > operator

---

### Syntax

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

### Description

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

*variable* specifies the dynamic array containing the data to be changed.

*field#*, *value#*, and *subvalue#* are 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.

Angle brackets to the right of an assignment operator indicate that an EXTRACT function is to be performed. For examples, see the [EXTRACT](#) function.

### Syntax

`@ (column [,row])`

`@ (-code [,arg])`

### Description

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.

*column* defines a screen column position.

*row* defines a screen row position.

*-code* is 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])`.

## @ function

---

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 [@ \(IT\\$VIDEO\)](#) function 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:

**Terminal Control Codes**

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	

---

**Terminal Control Codes (Continued)**

---

<b>Integer</b>	<b>Equate Name</b>	<b>Function</b>	<b>Argument</b>
-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
-23	IT\$AUXON	Auxiliary port on	
-24	IT\$AUXOFF	Auxiliary port off	
-25	IT\$STRON	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	

---

## @ function

---

### Terminal Control Codes (Continued)

Integer	Equate Name	Function	Argument
-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	The required graphics character
-40	IT\$ELINEGRFX	End line graphics	
-41	IT\$LINEGRFXCH	Line graphics character	
-42	IT\$DMI	Disable manual input	
-43	IT\$EMI	Enable manual input	
-44	IT\$BSCN	Blank screen	
-45	IT\$UBS	Unblank screen	Number of lines to scroll
-48	IT\$SU	Scroll up	
-49	IT\$SD	Scroll down	
-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	
-58	IT\$SBOLD	Start bold	



## Terminal Control Codes (Continued)

Integer	Equate Name	Function	Argument
-59	IT\$EBOLD	End bold	
-60	IT\$SECUR	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 Ardent	
-101 through -128	IT\$USERFIRST	Available for general use	
	IT\$USERLAST		

## @ function

---

### **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.

## @ function

---

### 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.

## @ function

---

### 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

Value	Name	Description
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

## @ function

---

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 back-



ground color to be chosen. (See “Set Foreground Color [@\(IT\\$FCOLOR\)](#)” on page 6-39 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:

<b>Value</b>	<b>Token</b>	<b>Description</b>
0	IT\$GRFX.CROSS	Cross piece
1	IT\$GRFX.H.LINE	Horizontal line
2	IT\$GRFX.V.LINE	Vertical line
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.

## @ function

---

### **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 labelled 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.

## @ function

---

### **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 Ardent @(-80) to @(-100)**

These functions are reserved for Ardent.

### **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

## @ function

---

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"
```

## @ function

---

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
```



### Syntax

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

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

### Description

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.

*expression* evaluates to any character string.

*start* is 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:

$$\text{string.length} - \text{substring.length} + 1$$

This lets you specify a substring consisting of the last *n* characters of a string without having to calculate the string length.

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.

*length* is an expression that evaluates to the length of the substring.

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.

*delimiter* can be 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

## [ ] operator

---

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= 121212
```

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

---

## Syntax

ABORT [*expression ...*]

ABORTE [*expression ...*]

ABORTM [*expression ...*]

## Description

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 BASIC program following an [EXECUTE](#) statement, and so on), whereas ABORT terminates all calling environments as well as the BASIC program. You can use it as part of an IF...THEN statement to terminate processing if certain conditions exist.

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 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.

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?NO  
Program "TEST": Line 3, Abort.
```

### Syntax

`ABS (expression)`

### Description

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.

### Example

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

This is the program output:

```
57
```

## ABSS function

---

### Syntax

`ABSS (dynamic.array)`

### Description

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.

### 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
```

### Syntax

`ACOS (expression)`

### Description

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.

### Example

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

This is the program output:

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

## ADDS function

---

### Syntax

ADDS (*array1*, *array2*)

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

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

### Description

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*.

### 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
```



### Syntax

ALPHA (*expression*)

### Description

Use the ALPHA function to determine whether *expression* is an alphabetic or nonalphabetic 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 *UniVerse NLS Guide*.

### 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
```

## ANDS function

---

### Syntax

ANDS (*array1*, *array2*)

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

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

### Description

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.

### 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
```

### Syntax

ASCII (*expression*)

### Description

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.

### 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

---

### Syntax

`ASIN (expression)`

### Description

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.

### Example

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

This is the program output:

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

### Syntax

ASSIGNED (*variable*)

### Description

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.

### 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

---

## Syntax

*variable* = *expression*

*variable* += *expression*

*variable* -= *expression*

*variable* := *expression*

## Description

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, a [READ](#) statement, 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).

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
```

## assignment statements

---

This is the program output:

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

# ATAN function

---

## Syntax

ATAN (*expression*)

## Description

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.

## 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)
```

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

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

This is the program output:

```
135      89.5756
1        45
```



## AUTHORIZATION statement

---

### Syntax

AUTHORIZATION "*username*"

### Description

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

*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 UV/Net, *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 subsequent **EXECUTE** and **PERFORM** statements 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.

## AUTHORIZATION statement

---

### 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
```

### Syntax

AUXMAP { ON | OFF | *expression* }

### Description

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

AUXMAP ON causes subsequent [PRINT](#) statements 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 [CRT](#), PRINT, or [INPUTERR](#), 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 *UniVerse NLS Guide*.

## BEGIN CASE statement

---

Use the BEGIN CASE statement to begin a set of CASE statements. For details, see the [CASE](#) statement.

# BEGIN TRANSACTION statement

---

## Syntax

```
BEGIN TRANSACTION [ISOLATION LEVEL level]  
[statements]
```

## Description

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

The ISOLATION LEVEL clause sets the transaction [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. <sup>1</sup>
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.

1. Lost updates are prevented if the ISOMODE configurable parameter is set to 1 or 2.

## 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

---

### Syntax

BITAND (*expression1*, *expression2*)

### Description

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.

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

### Syntax

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

### Description

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

*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

---

### Syntax

BITOR (*expression1*, *expression2*)

### Description

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.

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
```



### Syntax

`BITRESET (expression, bit#)`

### Description

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.

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

---

### Syntax

BITSET (*expression*, *bit#*)

### Description

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.

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
```

### Syntax

`BITTEST (expression, bit#)`

### Description

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.

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 run-time 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

---

### Syntax

BITXOR (*expression1*, *expression2*)

### Description

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.

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
```

## Syntax

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

## Description

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

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:

Counters and the BREAK Statement

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

## BREAK statement

---

### 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 ; *
```

### Syntax

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

### Description

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.

*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 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.

## BSCAN statement

---

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:

- 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.
- 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.
- 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 *UniVerse 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
```



## BSCAN statement

---

```
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

---

### Syntax

BYTE (*expression*)

### Description

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.

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. For more [information](#), see *UniVerse NLS Guide*.

### Example

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

- |           |  |
|-----------|--|
| 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). |

### Syntax

BYTELEN (*expression*)

### Description

In NLS mode, use the BYTELEN function to generate the number of bytes contained in the ASCII string value in *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. For more [information](#), see *UniVerse NLS Guide*.

## BYTETYPE function

---

### Syntax

BYTETYPE (*value*)

### Description

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

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

- |    |   |
|----|---|
| -1 | <i>value</i> 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 *UniVerse NLS Guide*.

### Syntax

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

### Description

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.

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 *UniVerse NLS Guide*.

# CALL statement

---

## Syntax

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

## Description

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

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 on page 3-12.

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 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 **DIM** 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.

### Examples

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

```
CALL SUB
```

## CALL statement

---

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 )
```



### Syntax

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

### Description

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.

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.

## CASE statement

---

### 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 statement

---

```
      CASE NAME[1,2]='BA'  
        PRINT NAME  
        GOSUB 30  
      END CASE  
      PRINT 'NO MATCH'  
      STOP
```

This is the program output:

```
NO MATCH
```

## CATS function

---

### Syntax

`CATS (array1, array2)`

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

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

### Description

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

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
```

### Syntax

`CHAIN command`

### Description

Use the CHAIN statement to terminate execution of a 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 run-time 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](#) or [PERFORM](#) statements in that CHAIN does not return control to the calling program. If a program chains to a proc, any nested calling procs are removed.

### 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

---

## Syntax

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

## Description

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.

*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 = "AAABBBCCDDDBBB"
PRINT CHANGE (A, "BBB", "ZZZ")
PRINT CHANGE (A, " ", "ZZZ")
PRINT CHANGE (A, "BBB", " ")
```

This is the program output:

```
AAAZZCCDDDDZZZ
AAABBBCCDDDBBB
AAACCCDDDD
```

### Syntax

CHAR (*expression*)

### Description

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 *UniVerse NLS Guide*.

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

### 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

---

## Syntax

CHARS (*dynamic.array*)

CALL –CHARS (*return.array*, *dynamic.array*)

CALL !CHARS (*return.array*, *dynamic.array*)

## Description

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 *UniVerse NLS Guide*.

## 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

---

### Syntax

CHECKSUM (*string*)

### Description

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

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

### 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

---

## Syntax

CLEAR [COMMON]

## Description

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.

## 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
```

### Syntax

CLEARDATA

### Description

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.

### Example

The following program is invoked from a paragraph. A list of filenames 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

---

## Syntax

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

## Description

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.

*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 run-time 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](#) (or [TRANSACTION START](#)) statement and the [COMMIT](#) (or [TRANSACTION START](#)) 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 (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.

## CLEARFILE statement

---

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

---

### Syntax

CLEARPROMPTS

CALL !CLEAR.PROMPTS

### Description

Use the CLEARPROMPTS statement to clear the value of the in-line 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 in-line prompts.

For information about in-line prompts, see the [ILPROMPT](#) function.

## CLEARSELECT statement

---

### Syntax

```
CLEARSELECT [ALL | list.number]
```

### Description

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.

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 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
```

## CLEARSELECT statement

---

```
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
```



### Syntax

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

### Description

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

*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.

## CLOSE statement

---

### 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
```

### Syntax

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

### Description

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

*file.variable* specifies a file previously opened with an [OPENSEQ](#) statement. If the file is neither accessible nor open, the program terminates with a run-time 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.

## CLOSESEQ statement

---

### 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
```

### Syntax

COL1 ( )

### Description

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 (see the [FIELD](#) function). 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.

### 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 ( )
```

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 ( )
```

## COL2 function

---

### Syntax

COL2 ( )

### Description

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 (see the [FIELD](#) function). 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.

### 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( )
```

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( )
```

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( )
```

## COL2 function

---

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()
```

# COMMIT statement

---

## Syntax

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

## Description

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.

A transaction includes all statements between a [BEGIN TRANSACTION](#) statement and the COMMIT 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](#), [WRITE](#), [WRITEV](#), or [MATWRITE](#) statement that is 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.



## COMMIT statement

---

If no transaction is active, the COMMIT statement generates a run-time 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](#) statement but on completion of the COMMIT statement.

# COMMON statement

---

## Syntax

```
COM[MON] [/name/] variable [,variable ...]
```

## Description

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.

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.

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.

### Syntax

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

### Description

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

*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:

- 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 *UniVerse 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')
```

## COMPARE function

---

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
```

## CONTINUE statement

---

The CONTINUE statement is a loop-controlling statement. For syntax details, see the [FOR](#) statement and the [LOOP](#) statement.

# CONVERT function

---

## Syntax

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

## Description

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.

*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
```

### Syntax

CONVERT *expression1* TO *expression2* IN *variable*

### Description

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.

*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

---

## Syntax

`COS (expression)`

## Description

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.

## Example

```
PRINT "COS(45) = " : COS(45)
END
```

This is the program output:

```
COS(45) = 0.7071
```



### Syntax

`COSH (expression)`

### Description

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.

### Example

```
PRINT "COSH( 2 ) = " : COSH( 2 )
```

This is the program output:

```
COSH( 2 ) = 3.7622
```

# COUNT function

---

## Syntax

COUNT (*string*, *substring*)

## Description

Use the COUNT function to return the number of times a substring is repeated in a string value.

*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#'
```

## COUNT function

---

```
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

---

## Syntax

COUNTS (*dynamic.array*, *substring*)

CALL –COUNTS (*return.array*, *dynamic.array*, *substring*)

CALL !COUNTS (*return.array*, *dynamic.array*, *substring*)

## Description

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*.

*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 run-time 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

---

## Syntax

CREATE *file.variable* { THEN *statements* [ ELSE *statements* ] | ELSE *statements* }

## Description

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 pathname 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 run-time 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](#), [READSEQ](#), or [READBLK](#). You need not use the CREATE statement if the first file operation is a [WRITESEQ](#), 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.

## 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
```

### Syntax

CRT [*print.list*]

### Description

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 [PRINT](#) statements.

*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.

If NLS is enabled, the CRT statement uses the terminal map in order to print. For more information about [maps and devices](#), see *UniVerse 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

---

## Syntax

DATA *expression* [ ,*expression* ...]

## Description

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.

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 run-time 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
```



## DATA statement

---

```
*  
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

---

## Syntax

DATE ( )

## Description

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.

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:

### Internal Formats for Dates

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
```

### Syntax

DCOUNT (*string*, *delimiter*)

### Description

Use the DCOUNT function to return the number of delimited fields in a data string.

*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 [COUNT](#) 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
```

## DEBUG statement

---

### Syntax

DEBUG

### Description

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:

**RAID Commands**

Command	Action
<i>line</i>	Displays the specified line of the source code.
<i>/[string]</i>	Searches the source code for <i>string</i> .
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 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.

### RAID Commands (Continued)

Command	Action
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.
<i>variable/</i>	Print the value of <i>variable</i> .
<i>variable!string</i>	Change the value of <i>variable</i> to <i>string</i> .

## DEFFUN statement

---

### Syntax

```
DEFFUN function [ ( [MAT] argument [ , [MAT] argument ... ] ) ]  
[CALLING call.ID]
```

### Description

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.

*function* is the name of the user-written function.

*arguments* 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). If the RETURN (*value*) statement specifies no value, an empty string is returned. The extra argument is reported by the [MAP](#) and [MAKE.MAP.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

---

### Syntax

DEL *dynamic.array* < *field#* [ ,*value#* [ ,*subvalue#*] ] >

### Description

Use the DEL statement to delete a field, value, or subvalue from a dynamic array. The DEL statement works similarly to the [DELETE](#) function.

*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**.



## DEL statement

---

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>
```

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

---

## Syntax

DELETE (*dynamic.array*, *field#*[ ,*value#*[ ,*subvalue#*] ] )

## Description

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.

*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**.

## DELETE function

---

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)
```

The next example deletes the first subvalue in field 4 and sets the value of Q to FLD1FVAL1VSUBV1SSUBV2FFSUBV4:

```
Q=DELETE (R,4,1,1)
```

The next example deletes the second value in field 2 and sets the value of Q to FLD1FVAL1FFSUBV3SSUBV4:

```
Q=DELETE (R,2,2)
```

The next example deletes field 3 entirely and sets the value of Q to FLD1FVAL1VSUBV1SSUBV2FSUBV3SSUBV4:

```
Q=DELETE (R,3,0,0)
```

The next example deletes the second subvalue in field 4 and sets the value of Q to FLD1FVAL1VSUBV1SSUBV2FFSUBV3:

```
Q=DELETE (R,4,1,2)
```

## DELETE statements

---

### 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 ]
```

### Description

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 (see the [OPEN](#) statement).

*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 run-time 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.

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).

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 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 is the error number.

### The LOCKED Clause

The LOCKED clause is optional, but recommended.

## DELETE statements

---

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 DELETED statement can be released explicitly with a [RELEASE](#) statement or implicitly with a [WRITE](#), [WRITEV](#), [MATWRITE](#), or DELETE statement. 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 STATEMENT:"
EXECUTE "LIST MLIST"
```

This is the program output:

```
FILE BEFORE DELETE STATEMENT:

3 records listed.

FILE AFTER DELETE STATEMENT:

2 records listed.
```

## DELETE statements

---

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 DELETEDU statement.

```
OPEN "", "SUBSIDIARIES" TO SUBS
READU REC FROM SUBS, 'MADRID'
      ELSE STOP 'Sorry, cannot open Subsidiaries file'
DELETEDU SUBS, "MADRID"
```

## DELETEDLIST statement

---

### Syntax

DELETEDLIST *listname*

### Description

Use the DELETEDLIST statement to delete a select list saved in the [&SAVEDLISTS& file](#).

*listname* can evaluate to the form:

*record.ID*

or:

*record.ID account.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 DELETEDLIST statement fails and the program terminates with a run-time error message.



## DELETEU statement

---

Use the DELETEU statement to maintain an update record lock while performing the [DELETE](#) statement.

## DIMENSION statement

---

### Syntax

DIM[ENSION] *matrix* (*rows*, *columns*) [ , *matrix* (*rows*, *columns*) ...]

DIM[ENSION] *vector* (*subscript*) [ , *vector* (*subscript*) ...]

### Description

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.

*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](#), [MATPARSE](#), [MATREAD](#), [MATREADU](#), 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.

## DIMENSION statement

---

- 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).

### 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
```

## DIMENSION statement

---

```
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
```

### Syntax

DISPLAY [*print.list*]

### Description

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** statements.

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.

### 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

---

### Syntax

DIV (*dividend*, *divisor*)

### Description

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.

### Example

```
X=100;  Y=25
Z = DIV (X,Y)
PRINT Z
```

This is the program output:

4

### Syntax

`DIVS (array1, array2)`

`CALL -DIVS (return.array, array1, array2)`

`CALL !DIVS (return.array, array1, array2)`

### Description

Use the DIVS function to create a dynamic array containing the result of the element-by-element division of two dynamic arrays.

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

---

## Syntax

DOWNCASE (*expression*)

## Description

Use the DOWNCASE function to change all uppercase letters in *expression* to lowercase. If *expression* evaluates to the null value, null is returned.

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 *UniVerse 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:
```



### Syntax

DQUOTE (*expression*)

### Description

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).

### Example

```
PRINT DQUOTE(12 + 5) : " IS THE ANSWER."  
END
```

This is the program output:

```
"17" IS THE ANSWER.
```

## DTX function

---

### Syntax

DTX (*number* [ ,*size* ] )

### Description

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 run-time error message.

### Example

```
X = 25
Y = DTX (X)
PRINT Y
Y = DTX (X,4)
PRINT Y
END
```

This is the program output:

```
19
0019
```

### Syntax

EBCDIC (*expression*)

### Description

Use the EBCDIC function to convert each character of *expression* from its ASCII representation value to its EBCDIC representation value. The EBCDIC and [ASCII](#) functions 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 run-time error message.

### 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

---

### Syntax

ECHO {ON | OFF | *expression*}

### Description

Use the ECHO statement to control the display of input characters on the screen.

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
```

### Syntax

END

### Description

Use the END statement to terminate a BASIC program or a section of an [IF](#), [READ](#), or [OPEN](#) statement.

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

---

## Syntax

```
ENTER subroutine  
variable = 'subroutine'  
ENTER @variable
```

## Description

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.

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). 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 run-time error message.

## Example

The following program transfers control to the cataloged program PROGRAM2:

```
ENTER PROGRAM2
```



### Syntax

EOF(ARG.)

### Description

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\(ARG.\)](#) and [SEEK\(ARG.\)](#) statements affect the value of the EOF(ARG.) function.

## EQS function

---

### Syntax

EQS (*array1*, *array2*)

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

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

### Description

Use the EQS 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 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
```

### Syntax

EQU[ATE] *symbol* TO *expression* [ ,*symbol* TO *expression* ... ]

EQU[ATE] *symbol* LIT[ERALLY] *string* [ ,*symbol* LIT *string* ... ]

### Description

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 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 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.

### Examples

In the following example, A is made equivalent to the string JANE:

```
JANE="HI"  
EQUATE A TO "JANE"
```

## EQUATE statement

---

Next, B is made equivalent to the variable 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
```

### Syntax

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

### Description

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

*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 = "AAABBBCCCDDBBB"
PRINT EREPLACE (A, "BBB", "ZZZ")
PRINT EREPLACE (A, "", "ZZZ")
PRINT EREPLACE (A, "BBB", "")
```

This is the program output:

```
AAAZZCCCDZZZ
ZZZAAABBBCCCDDBBB
AAACCCDD
```

# ERRMSG statement

---

## Syntax

ERRMSG *message.ID* [ ,*message.ID* ...]

## Description

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

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:

**ERRMSG File Codes**

Code	Action
A[( <i>n</i> )]	Display next argument left-justified; <i>n</i> specifies field length.
D	Display system date.
E [ <i>string</i> ]	Display record ID of message in brackets; <i>string</i> displayed after ID.
H [ <i>string</i> ]	Display <i>string</i> .
L [( <i>n</i> )]	Output a newline; <i>n</i> specifies number of newlines.
R [( <i>n</i> )]	Display next argument right-justified; <i>n</i> specifies field length.
S [( <i>n</i> )]	Output <i>n</i> blank spaces from beginning of line.
T	Display system time.

## Example

```
>ED ERRMSG 1
7 lines long.
----: P
0001: HBEGINNING OF ERROR MESSAGE
```

## ERRMSG statement

---

```
0002: L
0003: HFILE NAMED "
0004: A

0005: H" NOT FOUND.
0006: L
0007: H END OF MESSAGE
Bottom at line 7
----: Q
OPEN '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

---

## Syntax

EXCHANGE (*string*, *xx*, *yy*)

## Description

Use the EXCHANGE function to replace one character with another or to delete all occurrences of the specified character.

*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 *UniVerse NLS Guide*.



## EXCHANGE function

---

### 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

---

## 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 ]
```

## Description

Use the EXECUTE statement to execute UniVerse commands from within the BASIC program and then return execution to the statement following the EXECUTE statement.

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 **in-line 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.

## EXECUTE statement

---

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 **SOPTIONS** statement to cause EXECUTE to behave like PERFORM in other flavors.

## EXECUTE statement

---

### \$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.

### Syntax

EXIT

### Description

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.

### 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

---

### Syntax

EXP (*expression*)

### Description

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.

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* =  $2.71828^{**}(\textit{expression})$

### Example

```
X=5  
PRINT EXP ( X-1 )
```

This is the program output:

```
54.5982
```

### Syntax

EXTRACT (*dynamic.array*, *field#* [, *value#* [, *subvalue#*] ] )

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

### Description

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.

*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).

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 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.

## EXTRACT function

---

**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
```



### Syntax

FADD (*number1*, *number2*)

CALL !FADD (*return.array*, *number1*, *number2*)

### Description

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.

### Example

```
PRINT FADD(.234,.567)
```

This is the program output:

```
0.801
```

## FDIV function

---

### Syntax

FDIV (*number1*, *number2*)

CALL !FDIV (*return.array*, *number1*, *number2*)

### Description

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 run-time 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.

### Example

```
PRINT FDIV(.234,.567)
```

This is the program output:

```
0.4127
```

### Syntax

FFIX (*number*)

### Description

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.

## FFLT function

---

### Syntax

FFLT (*number*)

### Description

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

FIELD (*string*, *delimiter*, *occurrence* [ ,*num.substr*] )

### Description

Use the FIELD function to return one or more substrings located between specified delimiters in *string*.

*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
```

## FIELD function

---

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
```

### 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)`

### Description

Use the FIELDS function to return a dynamic array of substrings located between specified delimiters in each element of *dynamic.array*.

*delimiter* evaluates to any character, including 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 run-time 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

---

## Syntax

FIELDSTORE (*string*, *delimiter*, *start*, *n*, *new.string*)

## Description

Use the FIELDSTORE function to modify character strings by inserting, deleting, or replacing fields separated by specified delimiters.

*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 run-time 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
*
```



## FIELDSTORE function

---

```
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

---

## Syntax

FILEINFO (*file.variable*, *key*)

## Description

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 filename, and its VOC name. The information returned depends on the file type and the value of the 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 table [“Keys and Values Supplied to the FILEINFO Function.”](#)

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:

**Keys and Values Supplied to the FILEINFO Function**

Symbolic Name	Value	Description
FINFO\$IS.FILEVAR	0	1 if <i>file.variable</i> is a valid file variable; 0 otherwise.
FINFO\$VOCNAME	1	VOC name of the file.
FINFO\$PATHNAME	2	Pathname of the file.

## FILEINFO function

**Keys and Values Supplied to the FILEINFO Function (Continued)**

Symbolic Name	Value	Description
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.
FINFO\$LARGERRECORDSIZE	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.

## FILEINFO function

---

### Keys and Values Supplied to the FILEINFO Function (Continued)

Symbolic Name	Value	Description
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 (e.g., 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 <i>uvconfig</i> file, the returned string is the map name followed by the name of the configurable parameter in parentheses.

### 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*:

#### FILEINFO Values Returned by File Type

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 pathname	Pathname of file	Dynamic array of complete pathnames in VOC record order (pathname as used in VOC for unavailable files)	File's pathname
3	3	4	7	5

## FILEINFO function

**FILEINFO Values Returned by File Type (Continued)**

Key	Dynamic	Directory	Distributed	Sequential
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
10	Split load value	Empty string	Dynamic array of the split load value of each part file <sup>1</sup>	Empty string
11	Current load value	Empty string	Dynamic array of the current load value of each part file <sup>1</sup>	Empty string
12	Local file: empty string Remote file: node name	Empty string	Dynamic array of values where <i>value</i> 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

## FILEINFO function

---

**FILEINFO Values Returned by File Type (Continued)**

Key	Dynamic	Directory	Distributed	Sequential
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

1. 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.

**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 *UniVerse 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)
```

## FILEINFO function

---

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

---

## Syntax

```
FILELOCK [ file.variable ] [ , lock.type ]  
        [ ON ERROR statements ] [ LOCKED statements ]
```

## Description

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.

*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 error message. If *file.variable* evaluates to the null value, the FILELOCK statement fails and the program terminates with a run-time 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.



# FILELOCK statement

---

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](#), [RELEASE](#), or [STOP](#) statement.

## FILELOCK 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
```

### Syntax

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

### Description

Use the FILEUNLOCK statement to release a file lock set by the [FILELOCK](#) statement.

*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). 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.

## FILEUNLOCK statement

---

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
```

### Syntax

```
FIND element IN dynamic.array [ ,occurrence] SETTING fmc [ ,vmc [ ,smc] ]  
      { THEN statements [ ELSE statements] | ELSE statements }
```

### Description

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.

*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 run-time 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

---

### Syntax

```
FINDSTR substring IN dynamic.array [ ,occurrence]  
      SETTING fmc [ ,vmc [ ,smc ] ]  
      { THEN statements [ ELSE statements ] | ELSE statements }
```

### Description

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.

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 run-time 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
```

### Syntax

`FIX (number [ ,precision [ ,mode ] ] )`

### Description

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.

*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

---

## Syntax

FLUSH *file.variable* { THEN *statements* [ ELSE *statements* ] | ELSE *statements* }

## Description

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.

*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](#) and [WRITESEQ](#) statements 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
```



### Syntax

FMT (*expression*, *format*)

*expression format*

### Description

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.

*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:

- 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.

## FMT function

---

Exponential notation:

Q	Right justification – Break on field length.
QR	Right justification – Break on field length.
QL	Left justification

*edit* can be any of the following:

$n[m]$	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).
$nEm$	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:

# <i>n</i>	Data is displayed in a field of <i>n</i> 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.
% <i>n</i>	Data is displayed in a field of <i>n</i> zeros.
* <i>n</i>	Data is displayed in a field of <i>n</i> 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.

## FMT function

---

If *string* is larger than width *n*, a text mark (CHAR(251)) is inserted every *nth* character and each field is padded with the fill character to *width*.

The STATUS function reflects the result of *edit* as follows:

- 0 The edit code is successful.
- 1 The string expression is invalid.
- 2 The edit code is invalid.

See the [STATUS](#) function for more information.

### REALITY Flavor

In REALITY flavor accounts, you can use conversion codes in format expressions.

### Examples

Format Expression	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

## FMT function

---

Format Expression	Formatted Value
PRINT 233779 "R24"	233779.00
PRINT 233779 "R26"	2337.79
PRINT 2337.79 "R"	2337.79
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

---

## Syntax

FMTDP (*expression*, *format* [, *mapname*])

## Description

In NLS mode, use the FMTDP function to format data for output in display positions rather than character lengths.

*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.

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 *UniVerse NLS Guide*.

### Syntax

FMTS (*dynamic.array*, *format*)

CALL –FMTS (*return.array*, *dynamic.array*, *format*)

CALL !FMTS (*return.array*, *dynamic.array*, *format*)

### Description

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.

*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.

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 run-time error message.

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

# FMTSDP function

---

## Syntax

FMTSDP (*dynamic.array*, *format* [, *mapname*])

## Description

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.

*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.

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](#).

For more information about [display length](#), see *UniVerse NLS Guide*.



### Syntax

FMUL (*number1*, *number2*)

CALL !FMUL (*return.array*, *number1*, *number2*)

### Description

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.

### Example

```
PRINT FMUL(.234,.567)
```

This is the program output:

```
0.1327
```

# FOLD function

---

## Syntax

FOLD (*string*, *length*)

CALL !FOLD (*subdivided.string*, *string*, *length*)

## Description

Use the FOLD function to divide a string into a number of substrings separated by field marks.

*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 run-time 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
```

### Syntax

FOLDDP (*string*, *length* [, *mapname*])

### Description

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.

*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 [FOLD](#). For more information about [display length](#), see *UniVerse NLS Guide*.

## FOOTING statement

---

### Syntax

FOOTING [ON *print.channel*] *footing*

### Description

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). Logical print channel -1 prints the data on the screen, regardless of whether a PRINTER ON statement has been executed.

*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:

C[n]	Prints footing line centered in a field of <i>n</i> blanks. If <i>n</i> is not specified, centers the line on the page.
D	Prints current date formatted as <i>dd mmm yyyy</i> .
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 <i>n</i> blanks.
S	Left-justified, inserted page number.
T	Prints current time and date formatted as <i>dd mmm yyyy hh:mm:ss</i> . Time is in 12-hour format with "am" or "pm" appended.
\	Prints current time and date formatted as <i>dd mmm yyyy hh:mm:ss</i> . 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[ <i>n</i> ]	Prints current page number right-justified in a field of <i>n</i> blanks. The default value for <i>n</i> is 4.
^	Prints current page number right-justified in a field of <i>n</i> blanks. The default value for <i>n</i> 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 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](#) 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.

# FOOTING statement

---

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"	Hellothere
" 'G'Hello'G'there'G' "	Hellothere

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 *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.

## FOOTING statement

---

**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](#), FOOTING, and [\\$PAGE](#) statements 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

---

## Syntax

```
FOR variable = start TO end [STEP increment]  
    [loop.statements]  
    [CONTINUE | EXIT]  
[ { WHILE | UNTIL } expression ]  
    [loop.statements]  
    [CONTINUE | EXIT]  
NEXT [variable]
```

## Description

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.

*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 run-time 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.

# FOR statement

---

## 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
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

## FOR statement

---

Source Code	Program Output
<pre>PRINT 'LOOP 1 : ' SUM=0 FOR A=1 TO 10 UNTIL SUM&gt;20     SUM=SUM+A*A     PRINT "SUM= ",SUM NEXT</pre>	<pre>LOOP 1 : SUM=      1 SUM=      5 SUM=     14 SUM=     30</pre>
<pre>PRINT 'LOOP 2 : ' * Y=15 Z=0 FOR X=1 TO 20 WHILE Z&lt;Y     Z=Z+X     PRINT "Z= ",Z NEXT X</pre>	<pre>LOOP 2 : Z=        1 Z=        3 Z=        6 Z=       10 Z=       15</pre>

## FORMLIST statement

---

### Syntax

FORMLIST [*variable*] [TO *list.number*] [ON ERROR *statements*]

### Description

The FORMLIST statement is the same as the [SELECT](#) statement.

### Syntax

FSUB (*number1*, *number2*)

CALL !FSUB (*result*, *number1*, *number2*)

### Description

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.

### Example

```
PRINT FSUB(.234,.567)
```

This is the program output:

```
-0.333
```

# FUNCTION statement

---

## Syntax

FUNCTION [*name*] [ ( [MAT] *variable* [ , [MAT] *variable* ... ] ) ]

## Description

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.

*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
```

## GES function

---

### Syntax

GES (*array1*, *array2*)

CALL –GES (*return.array*, *array1*, *array2*)

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

### Description

Use the GES function to test if elements of one dynamic array are greater than or equal to corresponding elements of another dynamic array.

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*.



### 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 ]
```

### Description

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](#) 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.

**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.

## 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:

**GET Statements Terminating Conditions**

Condition	Description
Requested read length has been satisfied	The read is fully satisfied. <i>read.var</i> contains the characters read, and <i>last.char.read</i> 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 <i>read.var</i> . The end-of-packet character is stored in <i>last.char.read</i> . 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.

### GET Statements Terminating Conditions (Continued)

Condition	Description
Timeout limit has expired	The read could not be satisfied within the specified time limitation. If no characters have been read, <i>read.var</i> and <i>last.char.read</i> are set to the empty string, and <i>read.count</i> is set to 0. The system status code is set to 0 and may be checked with the <a href="#">STATUS</a> 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 <i>read.var</i> , and the empty string is stored in <i>last.char.read</i> . If no characters have been read, <i>read.var</i> and <i>last.char.read</i> are set to the empty string, and <i>read.count</i> is set to 0. The system status code is set to a nonzero value and may be 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.

## GET statements

---

- 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()
```

## GET statements

---

```
        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
```

## 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.

### Syntax

GET(ARG. [ ,*arg#*] ) *variable* [THEN *statements*] [ELSE *statements*]

### Description

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.

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.) statement

---

```
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.



### Syntax

```
GETLIST listname [TO list.number] [SETTING variable]  
      { THEN statements [ELSE statements] | ELSE statements }
```

### Description

Use the GETLIST statement to activate a saved select list so that a [READNEXT](#) statement can use it.

*listname* is an expression that evaluates to the form:

*record.ID*

or:

*record.ID account.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 [SOPTIONS](#) statement, the syntax of the GETLIST statement is:

```
GETLIST listname [TO list.variable] [SETTING variable]  
      { THEN statements [ELSE statements] | ELSE statements }
```

# GETLOCALE function

---

## Syntax

GETLOCALE (*category*)

## Description

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.

*category* is one of the following tokens that are defined in the UniVerse include file UVNLSLOC.H:

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:

LCE\$NO.LOCALES	UniVerse locales are not enabled.
LCE\$BAD.CATEGORY	Category is invalid.

For more information about [locales](#), see *UniVerse NLS Guide*.

### Syntax

GETREM (*dynamic.array*)

### Description

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*.

*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

# GOSUB statement

---

## Syntax

GOSUB *statement.label* [ : ]

GO SUB *statement.label* [ : ]

## Description

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.

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](#), [END](#), 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
```

## GOSUB statement

---

```
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

---

## Syntax

GO[TO] *statement.label* [ : ]

GO TO *statement.label* [ : ]

## Description

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.

## 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
```

### Syntax

GROUP (*string*, *delimiter*, *occurrence* [ ,*num.substr* ] )

### Description

Use the GROUP function to return one or more substrings located between specified delimiters in *string*.

*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
```

## GROUP function

---

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

---

### Syntax

GROUPSTORE *new.string* IN *string* USING *start*, *n* [ ,*delimiter*]

### Description

Use the GROUPSTORE statement to modify character strings by inserting, deleting, or replacing fields separated by specified delimiters.

*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,"#"
```

## GROUPSTORE statement

---

```
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
```

### Syntax

GTS (*array1*, *array2*)

CALL –GTS (*return.array*, *array1*, *array2*)

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

### Description

Use the GTS function to test if elements of one dynamic array are greater than elements of another dynamic array.

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

---

### Syntax

HEADING [ON *print.channel*] *heading*

HEADINGS [ON *print.channel*] *heading*

HEADINGN [ON *print.channel*] *heading*

### Description

Use the HEADING statement to specify the text and format of the heading to print at the top 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). 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:

C[ <i>n</i> ]	Prints heading line centered in a field of <i>n</i> blanks. If <i>n</i> is not specified, centers the line on the page.
D	Prints current date formatted as <i>dd mmm yyyy</i> .
T	Prints current time and date formatted as <i>dd mmm yyyy hh:mm:ss</i> . Time is in 12-hour format with "am" or "pm" appended.
\	Prints current time and date formatted as <i>dd mmm yyyy hh:mm:ss</i> . 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[ <i>n</i> ]	Inserts the record ID left-justified in a field of <i>n</i> blanks.
L	Starts a new line.

## HEADING statement

---

]	Starts a new line. Do not put the right bracket inside single quotation marks.
N	Suppresses automatic paging.
P[ <i>n</i> ]	Prints current page number right-justified in a field of <i>n</i> blanks. The default value for <i>n</i> is 4.
S	Left-justified, inserted page number.
^	Prints current page number right-justified in a field of <i>n</i> blanks. The default value for <i>n</i> 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 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](#) 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

# HEADING statement

---

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 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, HEADING calculates gaps using varying display positions rather than character lengths. For more information about [display length](#), see *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 [SOPTIONS](#) statement to cause HEADING, [FOOTING](#), and [PAGE](#) statements 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.

## HEADING statement

---

### 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
```



### Syntax

HUSH { ON | OFF | *expression* } [ SETTING *status* ]

### Description

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.

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.

## HUSH statement

---

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
```

### Syntax

ICHECK (*dynamic.array* [, *file.variable*] , *key* [, *column#* ])

### Description

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.

*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).

*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 function

---

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#column#constraint*

<i>error.code</i>	A code that indicates the type of failure. Error codes can be any of the following: <ul style="list-style-type: none"><li>0 No failure</li><li>1 SINGLEVALUED failure</li><li>2 NOT NULL failure</li><li>3 NOT EMPTY failure</li><li>4 ROWUNIQUE failure (including single-column association KEY)</li><li>5 UNIQUE (column constraint) failure</li><li>6 UNIQUE (table constraint) failure</li><li>7 Association KEY ROWUNIQUE failure when association has multiple KEY fields.</li><li>8 CHECK constraint failure</li><li>9 Primary key has too many parts</li><li>10 Referential constraint failure</li><li>11 Referential constraint failure that occurs when a numeric column references a nonnumeric column in the referenced table.</li></ul>
<i>column#</i>	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.
<i>constraint</i>	This element is returned only when <i>error.code</i> 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

## ICHECK function

---

first program writes its ICHECKed data, the first program's write fails even though the ICHECK did not fail.

# ICONV function

---

## Syntax

ICONV (*string*, *conversion*)

## Description

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.

*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:

- 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.

## 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

## ICONV function

Source Line	Converted Value
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 Lines	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

---

## Syntax

ICONVS (*dynamic.array, conversion*)

CALL -ICONVS (*return.array, dynamic.array, conversion*)

CALL !ICONVS (*return.array, dynamic.array, conversion*)

## Description

Use the ICONVS function to convert each element of *dynamic.array* to a specified internal storage format.

*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:

- 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.



### 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 }

### Description

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.

## IF statement

---

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

---

### 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*)

### Description

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.

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*.

### Syntax

ILPROMPT (*in.line.prompt*)

### Description

Use the ILPROMPT function to evaluate a string containing UniVerse in-line prompts.

*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:

- 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*).
- In 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.

## ILPROMPT function

---

*Sn* 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](#) conversion or matching pattern (see the [MATCH](#) operator). 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

---

## Syntax

INCLUDE [*filename*] *program*

INCLUDE *program* FROM *filename*

## Description

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.

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 filename 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 page). 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
```



### Syntax

INDEX (*string*, *substring*, *occurrence*)

### Description

Use the INDEX function to return the starting character position for the specified occurrence of *substring* in *string*.

*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='P1234XX001299XX00P'
TEST=INDEX(Q,X,Y)
PRINT "TEST= ",TEST
*
Q=INDEX("1234",'A',1)
PRINT "Q= ",Q
* The substring cannot be found.
```

## INDEX function

---

```
*  
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
```

### Syntax

INDEXS (*dynamic.array*, *substring*, *occurrence*)

CALL -INDEXS (*return.array*, *dynamic.array*, *substring*, *occurrence*)

CALL !INDEXS (*return.array*, *dynamic.array*, *substring*, *occurrence*)

### Description

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*.

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

---

## Syntax

INDICES (*file.variable* [ ,*indexname*])

## Description

Use the INDICES function to return information about the secondary key indexes in a file.

*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:

### Field 1 of Dynamic Arrays

Value	Value can be...	Description
Value 1	D	Data descriptor index.
	I	I-descriptor index.
	A	A-descriptor index.
	S	S-descriptor index.
	C	A- or S-descriptor index with correlative in field 8.
	SQL	SQL index.
Value 2	1	Index needs rebuilding.
	empty	Index does not need rebuilding.
Value 3	1	Empty strings are not indexed.
	empty	Empty strings are indexed.
Value 4	1	Automatic updating enabled.
	empty	Automatic updating disabled.
Value 5	pathname	Full pathname of the index file.
	empty	File is a distributed file.
Value 6	1	Updates are pending.
	empty	No updates pending.

### Field 1 of Dynamic Arrays (Continued)

Value	Value can be...		Description
Value 7	L	Left-justified.	Left-justified. Right-justified.
	R	Right-justified.	
Value 8	N	Nonunique.	Nonunique. Unique.
	U	Unique.	
Value 9	part numbers		Subvalued list of distributed file part numbers.
Value 10	1	Index needs building	Subvalued list corresponding to subvalues in Value 9.
	empty	No build needed	
Value 11	1	Empty strings not indexed	Subvalued list corresponding to subvalues in Value 9.
	empty	Empty strings indexed	
Value 12	1	Updating enabled	Subvalued list corresponding to subvalues in Value 9.
	empty	Updating disabled	
Value 13	index pathnames		Subvalued list of pathnames for indexes on distributed file part files, corresponding to subvalues in Value 9.
Value 14	1	Updates pending	Subvalued list corresponding to subvalues in Value 9.
	empty	No updates pending	
Value 15	L	Left-justified	Subvalued list corresponding to subvalues in Value 9.
	R	Right-justified	
Value 16	N	Nonunique	Subvalued list corresponding to subvalues in Value 9.
	U	Unique	
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).

## INDICES function

---

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.

## INDICES function

---

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 *UniVerse NLS Guide*.

# INMAT function

---

## Syntax

INMAT ( [array] )

## Description

Use the INMAT function to return the number of array elements that have been loaded after the execution of a [MATREAD](#), MATREADL, MATREADU, 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 [DIM](#) 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.

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
```



## Syntax

```
INPUT variable [ ,length] [ : ] [ _ ]
```

```
INPUT @ (col, row) [ , | : ] variable [ ,length] [ : ] [ format] [ _ ]
```

```
INPUTIF @ (col, row) [ , | : ] variable [ ,length] [ : ] [ format] [ _ ]  
      [ THEN statements] [ ELSE statements]
```

## Description

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*.

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.

## INPUT statement

---

*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.

## INPUT statement

---

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 *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.

### 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	? <b>Dave</b> Dave

## INPUT statement

---

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

## INPUT statement

---

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? <b>Dave</b> 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	? <b>NEW</b> _____ NEW

## INPUTCLEAR statement

---

### Syntax

INPUTCLEAR

### Description

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.

### Example

```
PRINT "DO YOU WANT TO CONTINUE (Y/N)?"  
INPUTCLEAR  
INPUT ANSWER, 1
```

### Syntax

`INPUTDISP [ @(col, row) [ , | : ] ] variable [format]`

### Description

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.

*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

---

## Syntax

INPUTDP *variable* [ , *length* ] [ : ] [ \_ ] [ THEN *statements* ] [ ELSE *statements* ]

## Description

In NLS mode, use the INPUTDP statement to let the user enter data. The INPUTDP statement is similar to the [INPUT](#), INPUTIF, and [INPUTDISP](#) statements, but it calculates display positions rather than character lengths.

*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 *UniVerse NLS Guide*.



### Syntax

INPUTERR [*error.message*]

### Description

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 or **PRINTERR** statement. INPUTERR clears the type-ahead buffer.

*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. 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 [INPUT](#) statement.

### Syntax

INPUTNULL *character*

### Description

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.

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

---

## Syntax

```
INPUTTRAP [trap.chars] { GOTO | GOSUB } label [ ,label ... ]
```

## Description

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.

*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.

### Syntax

INS *expression* BEFORE *dynamic.array* < *field#* [ , *value#* [ , *subvalue#* ] ] >

### Description

Use the INS statement to insert a new field, value, or subvalue into the specified *dynamic.array*.

*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 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.

## INS statement

---

- 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 [SOPTIONS](#) statements 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 [SOPTIONS](#) 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 **#FF1V2V3S6F9F5F7V3**:

```
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 "V#" 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 "S#" BEFORE Q<3,2,2>
```

# INSERT function

---

## Syntax

INSERT (*dynamic.array*, *field#*, *value#*, *subvalue#*, *expression*)

INSERT (*dynamic.array*, *field#* [ ,*value#* [ ,*subvalue#*] ] ; *expression*)

## Description

Use the INSERT function to return a dynamic array that has a new field, value, or subvalue inserted into the specified dynamic array.

*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 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 [SOPTIONS](#) 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.

## INSERT function

---

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;"#")
```

### Syntax

INT (*expression*)

### Description

Use the INT function to return the integer portion of an 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

---

### Syntax

ISNULL (*variable*)

### Description

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.

### Example

```
X = @NULL
Y = @NULL.STR
PRINT ISNULL(X), ISNULL(Y)
```

This is the program output:

```
1  0
```

### Syntax

ISNULLS (*dynamic.array*)

CALL -ISNULLS (*return.array*, *dynamic.array*)

### Description

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*.

### 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

---

## Syntax

ITYPE (*i.type*)

## Description

Use the ITYPE function to return the value resulting from the evaluation of an I-type expression in a UniVerse file dictionary.

*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:

```
DEPARTMENT
F1: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

---

## Syntax

KEYEDIT (*function*, *key*) [, (*function*, *key*) ] ...

## Description

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:

- 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	<b>Return</b> key
4	Back space
5	<b>Esc</b> key
6	Right arrow (—>)
7	Insert character
8	Delete character
9	Insert mode ON



## KEYEDIT statement

---

Code	Function
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.
Escape	32 through 159	Consists of the characters defined by the <b>Esc</b> key followed by the ASCII value 0 through 127 (see <a href="#">“Defining Escape Keys”</a> ).
Function	160 through 2,139,062,303	Consists of the characters defined by the <b>FUNCTION</b> key followed by the ASCII value 0 through 127. You can specify up to four ASCII values for complex keys (see <a href="#">“Defining Function Keys”</a> ).

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.

## KEYEDIT statement

---

### 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).

## KEYEDIT statement

Once you define the function key, the following formula is applied to the remaining characters in the sequence:

$$\text{ASCII value} * (2^{(8 * (\text{character position} - 1))})$$

Using the example above:

Key	ASCII	Formula	Intermediate Result	Final Result
J	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>
```

## KEYEDIT statement

---

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\(1050\)](#) function returns a dynamic array of defined KEYEDIT, [KEYEXIT](#) and [KEYTRAP](#) keys. Field marks (ASCII 254) delimit the elements of the dynamic array. Each field in the dynamic array has the following structure:

*key.typevfunction.parametervkey.parameter*

## KEYEDIT statement

---

*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

---

### Syntax

KEYEXIT (*value*, *key*) [ , (*value*, *key*) ] ...

### Description

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.

*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 <b>Esc</b> 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

---

### Syntax

KEYIN ( )

### Description

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

KEYTRAP (*value*, *key*) [ , (*value*, *key*) ] ...

### Description

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.

*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 <b>Esc</b> 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 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.

## KEYTRAP statement

---

### 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
```

### Syntax

LEFT (*string*, *n*)

### Description

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.

### Example

```
PRINT LEFT( "ABCDEFGH" , 3 )
```

This is the program output:

```
ABC
```

# LEN function

---

## Syntax

LEN (*string*)

## Description

Use the LEN function to return the number of characters in *string*. Calculations are based on character length rather than display length.

*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 *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
```

### Syntax

LENDP (*string* [ ,*mapname* ] )

### Description

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.

*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 to return the length of a string in character rather than display positions.

For more information about [display length](#), see *UniVerse NLS Guide*.

# LENS function

---

## Syntax

LENS (*dynamic.array*)

CALL -LENS (*return.array*, *dynamic.array*)

CALL !LENS (*return.array*, *dynamic.array*)

## Description

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.

Each element of *dynamic.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 *UniVerse NLS Guide*.

### Syntax

LENSDP (*dynamic.array* [, *mapname* ])

CALL –LENSDP (*return.array*, *dynamic.array* [, *mapname* ])

CALL !LENSDP (*return.array*, *dynamic.array* [, *mapname* ])

### Description

In NLS mode, use the LENS DP 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.

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 LENS DP 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 LENS DP 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 *UniVerse NLS Guide*.

## LES function

---

### Syntax

LES (*array1*, *array2*)

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

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

### Description

Use the LES function to test if elements of one dynamic array are less than or equal to the elements of another dynamic array.

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*.



### Syntax

[LET] *variable* = *expression*

### Description

Use the LET statement to assign the value of *expression* to *variable*. See “[assignment statements](#)” for more information about assigning values to variables.

### 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

---

### Syntax

LN (*expression*)

### Description

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.

### Example

```
PRINT LN(6)
```

This is the program output:

```
1.7918
```

### Syntax

LOCALEINFO (*category*)

### Description

In NLS mode, use the LOCALEINFO function to retrieve the settings of the current locale.

*category* is one of the following tokens that are defined in the UniVerse include file UVNLSLOC.H:

UVLC\$TIME	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.
UVLC\$NUMERIC	
UVLC\$MONETARY	
UVLC\$CTYPE	
UVLC\$COLLATE	
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:

LCES\$NO.LOCALES	NLS locales are not in force.
LCES\$BAD.CATEGORY	Category is invalid.

For more information about [locales](#), see *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)

---

### Syntax

```
LOCATE expression IN dynamic.array [ < field# [ , value# ] > ] [ , start ] [ BY seq ]  
    SETTING variable  
    { THEN statements [ ELSE statements ] | ELSE statements }
```

### Description

Use the LOCATE statement to search *dynamic.array* for a field, value, or subvalue. LOCATE returns a value indicating one of the following:

- 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 [SOPTIONS](#). 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 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*.

## LOCATE statement (IDEAL and REALITY syntax)

---

- 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.

## LOCATE statement (IDEAL and REALITY syntax)

---

Use the following values for *seq* to describe the ASCII sequence being searched:

AL or A	Ascending, left-justified (standard alphanumeric sort)
AR	Ascending, right-justified
DL or D	Descending, left-justified (standard alphanumeric sort)
DR	Descending, right-justified

*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 *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
```

## LOCATE statement (IDEAL and REALITY syntax)

---

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)

---

### Syntax

```
LOCATE expression IN dynamic.array <field# [, value# [, subvalue# ] ]> [ BY seq ]  
      SETTING variable  
      { THEN statements [ ELSE statements ] | ELSE statements }
```

### Description

Use the LOCATE statement to search *dynamic.array* for a field, value, or subvalue. LOCATE returns a value indicating one of the following:

- 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 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#*.



## LOCATE statement (INFORMATION syntax)

---

- 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.

## LOCATE statement (INFORMATION syntax)

---

Use the following values for *seq* to describe the ASCII sequence being searched:

AL or A	Ascending, left-justified (standard alphanumeric sort)
AR	Ascending, right-justified
DL or D	Descending, left-justified (standard alphanumeric sort)
DR	Descending, right-justified

*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 *UniVerse NLS Guide*.

### Examples

The examples show the INFORMATION flavor of the LOCATE statement. A field mark is shown by **␣**, a value mark is shown by **␣**, and a subvalue mark is shown by **␣**.

```
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
```

## LOCATE statement (INFORMATION syntax)

---

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)

---

### Syntax

```
LOCATE (expression, dynamic.array[, field# [, value#]]); variable[;seq])  
    { THEN statements [ ELSE statements ] | ELSE statements }
```

### Description

Use the LOCATE statement to search *dynamic.array* for a field, value, or subvalue. LOCATE returns a value indicating one of the following:

- 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 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.

## LOCATE statement (PICK syntax)

---

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:

AL or A	Ascending, left-justified (standard alphanumeric sort)
AR	Ascending, right-justified
DL or D	Descending, left-justified (standard alphanumeric sort)
DR	Descending, right-justified

*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.

## LOCATE statement (PICK syntax)

---

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 *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
```

### Syntax

`LOCK expression [THEN statements] [ELSE statements]`

### Description

Use the LOCK statement to protect specified user-defined resources or events against unauthorized use or simultaneous data file access by different users.

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.

## LOCK statement

---

**Note:** The LOCK statement protects user-defined resources only. The [READL](#), [READU](#), [READVL](#), [READVU](#), [MATREADL](#), and [MATREADU](#) statements 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:--
```



### Syntax

```
LOOP
    [loop.statements]
    [CONTINUE | EXIT]
    [{ WHILE | UNTIL } expression [DO] ]
    [loop.statements]
    [CONTINUE | EXIT]
REPEAT
```

### Description

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.

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.

## LOOP statement

---

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 Lines	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

## LOOP statement

---

Source Lines	Program Output
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

---

## Syntax

LOWER (*expression*)

## Description

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.

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  $\mathfrak{I}$ , a field mark is shown by  $\mathfrak{F}$ , a value mark is shown by  $\mathfrak{V}$ , a subvalue mark is shown by  $\mathfrak{S}$ , and a text mark is shown by  $\mathfrak{T}$ . CHAR(250) is shown as  $\mathfrak{Z}$ .

The following example sets A to DD $\mathfrak{F}$ EE $\mathfrak{V}$ 123 $\mathfrak{V}$ 77:

```
A= LOWER( 'DD' : IM'EE' : FM:123 : FM:777 )
```

## LOWER function

---

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

---

### Syntax

LTS (*array1*, *array2*)

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

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

### Description

Use the LTS function to test if elements of one dynamic array are less than elements of another dynamic array.

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*.

## Syntax

**MAT** *array* = *expression*

**MAT** *array1* = **MAT** *array2*

## Description

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.

## Examples

Source Lines	Program Output
<pre> DIM ARRAY(5) QTY=10 MAT ARRAY=QTY FOR X=1 TO 5     PRINT "ARRAY( ":X: " )=" ,ARRAY(X) NEXT X </pre>	<pre> ARRAY( 1 )=      10 ARRAY( 2 )=      10 ARRAY( 3 )=      10 ARRAY( 4 )=      10 ARRAY( 5 )=      10 </pre>
<pre> 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 </pre>	<pre> ONE( 1,1)=        1 ONE( 2,1)=        1 ONE( 3,1)=        1 ONE( 4,1)=        1 </pre>

## MAT statement

---

Source Lines	Program Output
<pre>DIM ONE(4,1) MAT ONE=1 DIM TWO(2,2) MAT TWO = MAT ONE FOR X=1 TO 2   FOR Y=1 TO 2     PRINT "TWO( ":X: ", ":Y: ")=", TWO(X,Y)   NEXT Y NEXT X</pre>	<pre>TWO(1,1)=      1 TWO(1,2)=      1 TWO(2,1)=      1 TWO(2,2)=      1</pre>

The following example sets all elements in ARRAY to the empty string:

```
MAT ARRAY= ' '
```



### Syntax

MATBUILD *dynamic.array* FROM *array* [ ,*start* [ ,*end* ] ] [ USING *delimiter* ]

### Description

Use the MATBUILD statement to build a dynamic array from a dimensioned array.

*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 [DIMENSION](#) 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

---

## Syntax

*string* MATCH[ES] *pattern*

## Description

Use the MATCH operator or its synonym MATCHES to compare a string expression with a 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, *nN* is the pattern for strings of *n* numeric characters.

The following table lists the *pattern* codes and their definitions:

**Pattern Matching Codes**

Pattern	Definition
...	Any number of any characters (including none).
0X	Any number of any characters (including none).
<i>nX</i>	<i>n</i> number of any characters.
0A	Any number of alphabetic characters (including none).
<i>nA</i>	<i>n</i> number of alphabetic characters.
0N	Any number of numeric characters (including none).
<i>nN</i>	<i>n</i> number of numeric characters.
' <i>text</i> '	Exact text; any literal string (quotation marks required).
" <i>text</i> "	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 *UniVerse NLS Guide*.

# MATCHFIELD function

---

## Syntax

MATCHFIELD (*string*, *pattern*, *field*)

## Description

Use the MATCHFIELD function to check a string against a match pattern (see the [MATCH](#) operator 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".

## MATCHFIELD function

### Examples

Source Lines	Program Output
<pre>Q=MATCHFIELD("AA123BBB9","2A0N3A0N",3) PRINT "Q= ",Q  ADDR='20 GREEN ST. NATICK, MA.,01234' ZIP=MATCHFIELD(ADDR,"0N0X5N",3) PRINT "ZIP= ",ZIP  INV='PART12345 BLUE AU' COL=MATCHFIELD(INV,"10X4A3X",2) PRINT "COL= ",COL</pre>	<pre>Q=      BBB  ZIP=    01234  COL=    BLUE</pre>

In the following example the string does not match the pattern:

Source Lines	Program Output
<pre>XYZ=MATCHFIELD('ABCDE1234',"2N3A4N",1) PRINT "XYZ= ",XYZ</pre>	<pre>XYZ=</pre>

In the following example the entire string does not match the pattern:

Source Lines	Program Output
<pre>ABC=MATCHFIELD('1234AB',"4N1A",2) PRINT "ABC= ",ABC</pre>	<pre>ABC=</pre>

# MATPARSE statement

---

## Syntax

MATPARSE *array* FROM *dynamic.array* [ ,*delimiter* ]

MATPARSE *array* [ ,*start* [ ,*end* ] ] FROM *dynamic.array* [ USING *delimiter* ]  
[ SETTING *elements* ]

## Description

Use the MATPARSE statement to separate the fields of *dynamic.array* into consecutive elements of *array*.

*array* must be named and dimensioned in a [DIMENSION](#) 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.

## MATPARSE statement

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)	X(0) 5#66#7
Y='1#22#3#44#5#66#7'	X(1) 1
MATPARSE X FROM Y, '#'	X(2) 22
FOR Z=0 TO 4	X(3) 3
PRINT "X( ":Z:" )",X(Z)	X(4) 44
NEXT Z	
PRINT	

## MATPARSE statement

---

Source Lines	Program Output
<pre>DIM Q(6) MATPARSE Q FROM 'ABCDEF', '' FOR P=1 TO 6 PRINT "Q(":P:")",Q(P) NEXT P PRINT</pre>	<pre>Q(1)  A Q(2)  B Q(3)  C Q(4)  D Q(5)  E Q(6)  F</pre>
<pre>DIM A(8,2) MATPARSE A FROM 'ABCDEFGDDHJCK', 'CD' FOR I = 1 TO 8 FOR J = 1 TO 2 PRINT "A(":I:",":J:")=",A(I,J)," ": NEXT J PRINT NEXT I END</pre>	<pre>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)=</pre>



### 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 }
```

### Description

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](#) or [COMMON](#) statement before it is used in this statement.

*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.

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

## MATREAD statements

---

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](#) statement.

### 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.

## MATREAD statements

---

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 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](#), [RELEASE](#), or [STOP](#) statement. An update record lock can be released with a [CLOSE](#), [DELETE](#), [MATWRITE](#), [RELEASE](#), [STOP](#), [WRITE](#), or [WRITEV](#) statement.

Locks acquired or promoted within a transaction are not released when the previous statements are processed.

## MATREAD statements

---

### 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
```

## MATREAD statements

---

```
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](#) statement.

## MATREADU statement

---

Use the MATREADU statement to set an update record lock and perform the MATREAD statement. For details, see the [MATREAD](#) statement.

# MATWRITE statements

---

## Syntax

```
MATWRITE[U] array ON | TO [file.variable,] record.ID  
    [ON ERROR statements] [LOCKED statements]  
    [THEN statements] [ELSE statements]
```

## Description

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.

*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](#) statement. For more information about [maps](#), see *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.



## MATWRITE statements

---

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. 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.

## MATWRITE statements

---

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 [MATREAD](#) statement). 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 for more information):

- 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
```

## MATWRITE statements

---

```
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](#) statement.

### Syntax

MAXIMUM (*dynamic.array*)

CALL !MAXIMUM (*result*, *dynamic.array*)

### Description

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.

### 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

---

### Syntax

MINIMUM (*dynamic.array*)

CALL !MINIMUM (*result*, *dynamic.array*)

### Description

Use the MINIMUM function to return the element with the lowest 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 smallest element found in *dynamic.array*.

*dynamic.array* is the array to be tested.

### 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
```

### Syntax

`MOD (dividend, divisor)`

### Description

Use the MOD function to calculate the value of the remainder after integer division is performed on the dividend expression by the divisor expression.

The MOD function calculates the remainder using the following formula:

$$\text{MOD}(X, Y) = X - (\text{INT}(X / Y) * Y)$$

*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.

### Example

```
X=85; Y=3  
PRINT 'MOD (X,Y)= ',MOD (X,Y)
```

This is the program output:

```
MOD (X,Y)=      1
```

# MODS function

---

## Syntax

MODS (*array1*, *array2*)

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

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

## Description

Use the MODS function to create a dynamic array of the remainder after the integer division of corresponding elements of two dynamic arrays.

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
```



### Syntax

`MULS (array1, array2)`

`CALL -MULS (return.array, array1, array2)`

`CALL !MULS (return.array, array1, array2)`

### Description

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*.

### 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

---

### Syntax

NAP [*milliseconds*]

### Description

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

NEG (*number*)

### Description

Use the NEG function to return the arithmetic inverse of the value of the argument.

*number* is an expression evaluating to a 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

---

### Syntax

NEGS (*dynamic.array*)

CALL -NEGS (*return.array*, *dynamic.array*)

### Description

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

NES (*array1*, *array2*)

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

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

### Description

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*.

## NEXT statement

---

### Syntax

NEXT [*variable*]

### Description

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.

### 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
```

### Syntax

NOBUF *file.variable* { THEN *statements* [ ELSE *statements* ] | ELSE *statements* }

### Description

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](#) 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.

### 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

---

## Syntax

NOT (*expression*)

## Description

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.

## Example

```
X=5; Y=5  
PRINT NOT(X-Y)  
PRINT NOT(X+Y)
```

This is the program output:

```
1  
0
```



### Syntax

NOTS (*dynamic.array*)

CALL -NOTS (*return.array*, *dynamic.array*)

CALL !NOTS (*return.array*, *dynamic.array*)

### Description

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*.

### Example

```
X=5; Y=5  
PRINT NOTS (X-Y:@VM:X+Y)
```

This is the program output:

```
1V0
```

## NULL statement

---

### Syntax

NULL

### Description

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.

### 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
```

### Syntax

NUM (*expression*)

### Description

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 *UniVerse NLS Guide*.

### Example

```
X=NUM( 2400 )
Y=NUM( "Section 4" )
PRINT "X= ",X,"Y= ",Y
```

This is the program output:

```
X=      1      Y=      0
```

## NUMS function

---

### Syntax

NUMS (*dynamic.array*)

CALL -NUMS (*return.array*, *dynamic.array*)

CALL !NUMS (*return.array*, *dynamic.array*)

### Description

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 *UniVerse NLS Guide*.

### Syntax

OCONV (*string*, *conversion*)

### Description

Use the OCONV function to convert *string* to a specified format for external output. The result is always a string expression.

*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 left-most 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:

- 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.

## OCONV function

---

### 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' ) <sup>1</sup>	3/2/1993
DATE=OCONV( '9166' , "D2-" )	2-3-93
DATE=OCONV( 0 , 'D' )	31 Dec 1967

1. For IN2, PICK, and REALITY flavor accounts.

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

## OCNV function

Source Line	Converted Value
BIN=OCNV( 1024 , "MB" )	10000000000
BIN=OCNV( ' CDE' , "MB0C" )	010000110100010001000101

The following examples show masked decimal conversions:

Source Line	Converted Value
X=OCNV( 987654 , "MD2" )	9876.54
X=OCNV( 987654 , "MD0" )	987654
X=OCNV( 987654 , "MD2 , \$" )	\$9,876.54
X=OCNV( 987654 , "MD24\$" )	\$98.77
X=OCNV( 987654 , "MD2-Z" )	9876.54
X=OCNV( 987654 , "MD2 , D" )	9,876.54
X=OCNV( 987654 , "MD3 , \$CPZ" )	\$987.654
X=OCNV( 987654 , "MD2 , ZP12#" )	####9,876.54

## OCONVS function

---

### Syntax

OCONVS (*dynamic.array*, *conversion*)

CALL –OCONVS (*return.array*, *dynamic.array*, *conversion*)

CALL !OCONVS (*return.array*, *dynamic.array*, *conversion*)

### Description

Use the OCONVS function to convert the elements of *dynamic.array* to a specified format for external output.

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 left-most 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:

- 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.



### Syntax

ON *expression* GOSUB *statement.label* [ : ] [ ,*statement.label* [ : ] ... ]

ON *expression* GO[TO] *statement.label* [ : ] [ ,*statement.label* [ : ] ... ]

### Description

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.

### 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 subrou-

## ON statement

---

time. 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

---

## Syntax

```
OPEN [dict,] filename [TO file.variable] [ON ERROR statements]  
    { THEN statements [ELSE statements] | ELSE statements }
```

## Description

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.

*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.

## OPEN statement

---

### 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	Filename not found in the VOC file.
-2 <sup>1</sup>	Null filename or file. This error may also occur when you cannot open a file across UV/Net.
-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 <sup>1</sup>	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 <sup>1</sup>	Invalid part file information.
-9 <sup>1</sup>	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 <sup>1</sup>	Index problem.
-14	Cannot open the NFS file.

1. A generic error that can occur for various reasons.

### 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
```

## OPEN statement

---

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

---

## Syntax

```
OPENCHECK [dict,] filename [TO file.variable]  
    { THEN statements [ELSE statements] | ELSE statements }
```

## Description

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.

## 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	Filename not found in the VOC file.
-2 <sup>1</sup>	Null filename or file. This error may also occur when you cannot open a file across UV/Net.
-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 <sup>1</sup>	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 <sup>1</sup>	Invalid part file information.
-9 <sup>1</sup>	Invalid type 30 file information in a distributed file.



## OPENCHECK statement

---

Value	Description
-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 <sup>1</sup>	Index problem.
-14	Cannot open the NFS file.

1. A generic error that can occur for various reasons.

# OPENDEV statement

---

## Syntax

```
OPENDEV device TO file.variable [LOCKED statements]  
      { THEN statements [ELSE statements] | ELSE statements }
```

## Description

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](#) and [WRITESEQ](#) statements for more details on sequential processing.

*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 “[Devices on Windows NT](#).”

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 *UniVerse NLS Guide*.

## Devices on Windows NT

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

## OPENDEV statement

---

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.
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.

## OPENDEV statement

---

### 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
```

### Syntax

```
OPENPATH pathname [TO file.variable] [ON ERROR statements]  
      { THEN statements [ELSE statements] | ELSE statements }
```

### Description

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).

*pathname* specifies the relative or absolute pathname 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.

## OPENPATH statement

---

- 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	Filename not found in the VOC file.
-2 <sup>1</sup>	Null filename or file. This error may also occur when you cannot open a file across UV/Net.
-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 <sup>1</sup>	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 <sup>1</sup>	Invalid part file information.
-9 <sup>1</sup>	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.

## OPENPATH statement

---

Value	Description
-13 <sup>1</sup>	Index problem.
-14	Cannot open the NFS file.

1. A generic error that can occur for various reasons.

### Example

The following example opens the file SUN.MEMBER. The pathname specifies the file.

```
OPENPATH '/user/members/SUN.MEMBER' ELSE ABORT
```

## OPENSEQ statement

---

### 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}
```

### Description

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](#) and [WRITESEQ](#) statements for more details on sequential processing.

**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](#) 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 pathname 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 pathname 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 pathname 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 *.uvnlsmmap* file in the directory containing the type 1 or type 19 file. If there is no *.uvnlsmmap* 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 *.uvnlsmmap* file in the directory holding *pathname*. If there is no *.uvnlsmmap* 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 *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.

### 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

## OPENSEQ statement

---

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

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	Filename not found in the VOC file.
-2 <sup>1</sup>	Null filename or file. This error may also occur when you cannot open a file across UV/Net.
-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 <sup>1</sup>	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 <sup>1</sup>	Invalid part file information.
-9 <sup>1</sup>	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 <sup>1</sup>	Index problem.
-14	Cannot open the NFS file.

<sup>1</sup> A generic error that can occur for various reasons.

## OPENSEQ statement

---

### 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
```

### Syntax

ORS (*array1*, *array2*)

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

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

### Description

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*.

### 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

---

## Syntax

PAGE [ON *print.channel*] [*page#*]

## Description

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](#) and [FOOTING](#) statements). If there is no heading or footing, PAGE clears the screen.

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). 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
```

## Syntax

PERFORM *command*

## Description

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.

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	INSERT	SEARCH	SSELECT
COUNT	LIST	SELECT (Retrieve)	STAT
DELETE (SQL)	LIST.ITEM	SELECT (SQL)	SUM
DISPLAY	LIST.LABEL	SORT	UPDATE
ESEARCH	RUN	SORT.ITEM	
GET.LIST	SAVE.LIST	SORT.LABEL	

## 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 [SOPTIONS](#) statement.

## PERFORM 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
```



### Syntax

PRECISION *expression*

### Description

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.

*expression* specifies a number from 0 through 9. 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
```

# PRINT statement

---

## Syntax

PRINT [ON *print.channel*] [*print.list*]

## Description

Use the PRINT statement to send data to the screen, a line printer, or another print file.

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](#), [FOOTING](#), [PAGE](#), 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. 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 *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

---

## Syntax

PRINTER { ON | OFF | RESET }

PRINTER CLOSE [ON *print.channel*]

## Description

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.

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](#) to set a delay before printing. See also the [TPARM](#) 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.

## PRINTER statement

---

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

---

## Syntax

PRINTERR [*error.message*]

## Description

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.

*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. 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.

## 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.

## PRINTERR statement

---

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:

**ERRMSG File Codes**

Code	Action
A[(n)]	Display next argument left-justified; <i>n</i> specifies field length.
D	Display system date.
E [string]	Display record ID of message in brackets; <i>string</i> displayed after ID.
H [string]	Display <i>string</i> .
L [(n)]	Output newline; <i>n</i> specifies number of newlines.
R [(n)]	Display next argument right-justified; <i>n</i> specifies field length.
S [(n)]	Output <i>n</i> blank spaces from beginning of line.
T	Display system time.

## PROCREAD statement

---

### Syntax

PROCREAD *variable*

{ THEN *statements* [ ELSE *statements* ] | ELSE *statements* }

### Description

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.



## PROCWRITE statement

---

### Syntax

PROCWRITE *string*

### Description

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.

# PROGRAM statement

---

## Syntax

PROG[RAM] [*name*]

## Description

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.

## Example

```
PROGRAM BYSTATE
```

Syntax

PROMPT *character*

Description

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 ( ? ).

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
PROMPT "HELLO" PRINT "ENTER ANSWER " : INPUT ANS PROMPT "- " PRINT "ENTER ANSWER " : INPUT ANS PROMPT " " PRINT "ENTER ANSWER " : INPUT ANS END	ENTER ANSWER <b>HANSWER</b> ENTER ANSWER <b>-YES</b> ENTER ANSWER <b>NO</b>

## PWR function

---

### Syntax

`PWR (expression, power)`

### Description

Use the PWR function to return the value of *expression* raised to the power specified by *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
```

### Syntax

QUOTE (*expression*)

### Description

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).

### Example

```
PRINT QUOTE(12 + 5) : " IS THE ANSWER."  
END
```

This is the program output:

```
"17" IS THE ANSWER.
```

# RAISE function

---

## Syntax

RAISE (*expression*)

## Description

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.

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)	to		CHAR(251)
	CHAR(249)	to		CHAR(250)
	CHAR(248)	to		CHAR(249)

## PIOPEN Flavor

In PIOPEN flavor, the delimiters that can be raised are CHAR(255) through CHAR(252). 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 [SOPTIONS](#) INFO.MARKS statement

## Examples

In the following examples an item mark is shown by **r**, 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 DD**r**EE**r**123**r**777:

```
A= RAISE('DD':FM'EE':FM:123:FM:777)
```

The next example sets **B** to 1r2F3r4V5:

```
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

---

## Syntax

RANDOMIZE (*expression*)

## Description

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. 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.

## 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 N
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
```



### 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 }
```

### Description

Use READ statements to assign the contents of a record from a UniVerse file to *dynamic.array*.

#### Uses for READ 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). If the file is neither accessible nor open, the program terminates with a run-time error message.

## READ statements

---

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.

**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:

- 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:

- 3 The unmappable character is in the record ID.
- 4 The unmappable character is in the record's data.<sup>1</sup>

For more information about [maps](#), see *UniVerse NLS Guide*.

---

1. 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.

### 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

## READ statements

---

In this statement...	This requested lock...	Conflicts with...
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 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](#), [RELEASE](#), or [STOP](#) statement. An update record lock can be released with a [CLOSE](#), [DELETE](#), [MATWRITE](#), [RELEASE](#), [STOP](#), [WRITE](#), 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*.

## READ statements

---

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
```

## READ statements

---

```
        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
```

### Syntax

```
READBLK variable FROM file.variable, blocksize  
      { THEN statements [ ELSE statements ] | ELSE statements }
```

### Description

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.

*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, 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](#), [READSEQ](#), and [WRITESEQ](#) statements.

## READBLK statement

---

If NLS is enabled and *file.variable* has a map associated with it, the data is mapped accordingly. For more information about [maps](#), see *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](#) statement.

# READLIST statement

---

## Syntax

```
READLIST dynamic.array [FROM list.number]  
      { THEN statements [ ELSE statements ] | ELSE statements }
```

## Description

Use the READLIST statement to read the remainder of an active select list into a dynamic array.

*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 a [READNEXT](#) statement is 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 [SOPTIONS](#) 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.

## READLIST statement

---

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.ID account.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 [SOPTIONS](#) statement to get READLIST to behave as it does in IDEAL flavor accounts.

# READNEXT statement

---

## Syntax

```
READNEXT dynamic.array [ ,value [ ,subvalue ] ] [FROM list]  
      { THEN statements [ELSE statements] | ELSE statements }
```

## Description

Use the READNEXT statement to assign the next record ID from an active select list to *dynamic.array*.

*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](#) statement 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 [SOPTIONS](#) 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:
```

## READNEXT statement

---

```
*  
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

---

## Syntax

```
READSEQ variable FROM file.variable [ON ERROR statements]  
      { THEN statements [ELSE statements] | ELSE statements }
```

## Description

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 [OPENSEQ](#) statement sets this pointer to the first byte of the file, and it is advanced by READSEQ, [READBLK](#), [WRITESEQ](#), and [WRITEBLK](#) 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:

- 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 *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
```

## READT statement

---

### Syntax

```
READT [UNIT (mtu)] variable  
      { THEN statements [ELSE statements] | ELSE statements }
```

### Description

Use the READT statement to read the next tape record from a magnetic tape unit and assign its contents to a variable.

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:

***mtu* Codes**

Code	Available Options
<i>m</i> (mode)	0 = No conversion 1 = EBCDIC conversion 2 = Invert high bit 3 = Invert high bit and EBCDIC conversion
<i>t</i> (tracks)	0 = 9 tracks. Only 9-track tapes are supported.
<i>u</i> (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 *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](#) statement.

## 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](#) statement.

## READVL statement

---

Use the READVL statement to set a shared record lock and perform the READV statement. For details, see the [READ](#) statement.

## 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](#) statement.

## REAL function

---

### Syntax

REAL (*number*)

### Description

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.

## RECORDLOCK statements

---

### Syntax

RECORDLOCKL *file.variable*, *record.ID* [ ON ERROR *statements* ]  
[ LOCKED *statements* ]

RECORDLOCKU *file.variable*, *record.ID* [ ON ERROR *statements* ]  
[ LOCKED *statements* ]

### Description

Use RECORDLOCK statements to acquire a record lock on a record without reading the record.

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](#), 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.

## RECORDLOCK statements

---

### 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



## RECORDLOCK statements

---

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**, **RELEASE**, or **STOP** statement. An update record lock can be released with a **CLOSE**, **DELETE**, **MATWRITE**, **RELEASE**, **STOP**, **WRITE**, 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

---

## Syntax

RECORDLOCKED (*file.variable*, *record.ID*)

## Description

Use the RECORDLOCKED function to return the status of a 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 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.

**RECORDLOCKED.INS.IBAS File Equate Names**

Equate Name	Value	Meaning
LOCK\$MY.FILELOCK	3	This user has a FILELOCK.
LOCK\$MY.READL	2	This user has a shared record lock.
LOCK\$MY.READU	1	This user has an update 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.

## RECORDLOCKED function

---

### Value Returned by the STATUS Function

Possible values returned by the STATUS function, and their meanings, are as follows:

- > 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

---

## Syntax

```
RELEASE [ file.variable [ ,record.ID ] ] [ ON ERROR statements ]
```

## Description

Use the RELEASE statement to unlock, or release, locks set by a [FILELOCK](#), [MATREADL](#), [MATREADU](#), [READL](#), [READU](#), [READVL](#), [READVU](#), 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.

*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). 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.

## RELEASE statement

---

- 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"
```

## REM function

---

### Syntax

REM (*dividend*, *divisor*)

### Description

Use the REM function to calculate the remainder after integer division is performed on the dividend expression by the divisor expression.

The REM function calculates the remainder using the following formula:

$$\text{REM}(X, Y) = X - (\text{INT}(X / Y) * Y)$$

*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.

### Example

```
X=85; Y=3
PRINT 'REM (X,Y)= ',REM (X,Y)
```

This is the program output:

```
REM (X,Y)=      1
```

### Syntax

REM [*comment.text*]

### Description

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.

### 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

---

## Syntax

REMOVE (*dynamic.array*, *variable*)

## Description

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 multi-value list processing.

*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:

- 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) (*Not available in the PIOPEN flavor*)
- 7 ASCII CHAR(249) (*Not available in the PIOPEN flavor*)
- 8 ASCII CHAR(248) (*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.



# REMOVE function

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 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  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	FIRST = 2      X = MIKE  Y = 3    A = 12 Y = 3    A = 4 Y = 3    A = 5 Y = 3    A = 7654 Y = 0    A = 0

# REMOVE statement

---

## Syntax

REMOVE *element* FROM *dynamic.array* SETTING *variable*

## Description

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.

*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:

- 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) – Not supported in the PIOPEN flavor
- 7 ASCII CHAR(249) – Not supported in the PIOPEN flavor
- 8 ASCII CHAR(248) – 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).

# REMOVE statement

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 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.

### Syntax

REPLACE (*expression*, *field#*, *value#*, *subvalue#* { , | ; } *replacement*)

REPLACE (*expression* [ ,*field#* [ ,*value#*] ] ; *replacement*)

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

### Description

Use the REPLACE function to return a copy of a dynamic array with the specified field, value, or subvalue replaced with new data.

*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).

*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 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.

## REPLACE function

---

- 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 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

## REPLACE function

---

**SOPTIONS** 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 **SOPTIONS** 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
```

## REPLACE function

---

```
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;"#")
```



### Syntax

RETURN [TO *statement.label*]

### Description

Use the RETURN statement to terminate a subroutine and return control to the calling program or statement.

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
```

## RETURN statement

---

```
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

---

### Syntax

RETURN (*expression*)

### Description

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.

# REUSE function

---

## Syntax

REUSE (*expression*)

## Description

Use the REUSE function to specify that the value of the last field, value, or subvalue be reused in a dynamic array operation.

*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 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

---

### Syntax

REVREMOVE *element* FROM *dynamic.array* SETTING *variable*

### Description

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*.

*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:

- 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)
- 7 ASCII CHAR(249)
- 8 ASCII CHAR(248)

The REVREMOVE statement extracts one element each time it is executed, beginning with the last element in *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.

## REVREMOVE statement

---

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

---

### Syntax

```
REWIND [UNIT (mtu)]  
      { THEN statements [ELSE statements] | ELSE statements }
```

### Description

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.

### 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"
```



### Syntax

`RIGHT (string, n)`

### Description

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.

### Example

```
PRINT RIGHT( "ABCDEFGH" , 3 )
```

This is the program output:

```
FGH
```

# RND function

---

## Syntax

RND (*expression*)

## Description

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 for details on generating repeatable sequences of random numbers.

## Example

```
A=20
PRINT RND(A)
PRINT RND(A)
PRINT RND(A)
PRINT RND(A)
```

This is the program output:

```
10
3
6
10
```

### Syntax

```
ROLLBACK [WORK] [ THEN statements ] [ ELSE statements ]
```

### Description

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.

### Example

This example begins a transaction that applies locks to rec1 and rec2. If errors occur (such as a failed **READU** or a failed **WRITE**), 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
```

## **ROLLBACK statement**

---

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

### Syntax

RPC.CALL (*connection.ID*, *procedure*, *#args*, MAT *arg.list*, *#values*,  
MAT *return.list*)

### Description

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.

*connection.ID* is the handle of the open server connection on which to issue the RPC request. The [RPC.CONNECT](#) function 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:

- |       |  |
|-------|--|
| 81001 | Connection closed, reason unspecified.                                       |
| 81002 | <i>connection.ID</i> 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 <i>arg.list</i> .               |

## RPC.CALL function

---

- 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
```

### Syntax

RPC.CONNECT (*host*, *server*)

### Description

Use the RPC.CONNECT function to establish a connection to a server process. Once the host and server are identified, the local UV/Net 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](#) and [RPC.DISCONNECT](#).

*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:

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 <i>/etc/hosts</i> file.
81012	Remote <i>unirpcd</i> cannot start <i>service</i> because it could not fork the process.
81013	Cannot open the remote <i>unirpcservices</i> file.
81014	Service not found in the remote <i>unirpcservices</i> file.
81015	Connection attempt timed out.

## RPC.CONNECT function

---

### 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

---

### Syntax

RPC.DISCONNECT (*connection.ID*)

### Description

Use the RPC.DISCONNECT function to end an RPC session.

*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:

- 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
```

# SADD function

---

## Syntax

SADD (*string.number.1*, *string.number.2*)

## Description

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.

## Example

```
A = 888888888888888888
B = 777777777777777777
X = "888888888888888888"
Y = "777777777777777777"
PRINT A + B
PRINT SADD(X,Y)
```

This is the program output:

```
1666666666666667000
16666666666666665
```

### Syntax

SCMP (*string.number.1*, *string.number.2*)

### Description

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.

### 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

---

### Syntax

SDIV (*string.number.1*, *string.number.2* [, *precision*])

### Description

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.

### 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
0.3333333333333333
```

### Syntax

```
SEEK file.variable [ , offset [ , relto ] ]  
      { THEN statements [ ELSE statements ] | ELSE statements }
```

### Description

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:

- 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

## SEEK statement

---

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](#), [READSEQ](#), and [WRITESEQ](#) 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
```

### Syntax

SEEK(ARG. [ ,arg# ] ) [ THEN *statements* ] [ ELSE *statements* ]

### Description

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.

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\(ARG.\)](#) 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.) statement

---

SEEK ( ARG . , A-B )

SEEK ( ARG . , 1 )

the system pointer moves as follows:

ARG2

ARG2

ARG3

ARG3

ARG1



### 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*]

### Description

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.

*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 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.

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](#) state-

## SELECT statements

---

ment. 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 [SOPTIONS](#) 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](#) function 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
```

## SELECT statements

---

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

---

### Syntax

SELECTE TO *list.variable*

### Description

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 [READ-NEXT](#) statement.

## SELECTINDEX statement

---

### Syntax

```
SELECTINDEX index [, alt.key] FROM file.variable [TO list.number]
```

### Description

Use the SELECTINDEX statement to create select lists from secondary indexes.

*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:** 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.

## SELECTINDEX 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
```

### Syntax

SELECTINFO (*list*, *key*)

### Description

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:

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 SYSCOM 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

---

## Syntax

```
SEND output [:] TO device  
    { THEN statements [ ELSE statements ] | ELSE statements }
```

## Description

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](#) or [OPENSEQ](#) statement.

*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
```



### Syntax

SENTENCE ( )

### Description

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.

### Example

```
PRINT SENTENCE ( )
```

This is the program output:

```
RUN BP TESTPROGRAM
```

# SEQ function

---

## Syntax

SEQ (*expression*)

## Description

Use the SEQ function to convert an ASCII character to its numeric string equivalent.

*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.

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 *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
```

### Syntax

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

### Description

Use the SEQs function to convert a dynamic array of ASCII characters to their numeric string equivalents.

*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 *UniVerse NLS Guide*.

### Example

```
G="T":@VM:"G"  
A=SEQS(G)  
PRINT A  
PRINT SEQs("G")
```

This is the program output:

```
84V71  
71
```

# SET TRANSACTION ISOLATION LEVEL statement

---

## Syntax

SET TRANSACTION ISOLATION LEVEL *level*

## Description

Use the SET TRANSACTION ISOLATION LEVEL statement to set the default transaction isolation level you need for your program.

**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:

$\{ n \mid \text{keyword} \mid \text{expression} \}$

*level* is an expression that evaluates to 0 through 4, or one of the following keywords:

### Effects of ISOLATION LEVELs on Transactions

Integer	Keyword	Effect on This Transaction
0	NO.ISOLATION	Prevents lost updates. <sup>1</sup>
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.

1. Lost updates are prevented if the ISOMODE configurable parameter is set to 1 or 2.

## SET TRANSACTION ISOLATION LEVEL statement

---

### 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
```

# SETLOCALE function

---

## Syntax

SETLOCALE (*category*, *value*)

## Description

In NLS mode, use the SETLOCALE function to enable or disable a locale for a specified category or change its setting.

*category* is one of the following tokens that are defined in the UVNLSLOC.H file:

UVLC\$ALL	Sets or disables all categories as specified in <i>value</i> . <i>value</i> is the name of a locale, OFF, or DEFAULT. <i>value</i> can also be a dynamic array whose elements correspond to the categories.
UVLC\$TIME	Sets or disables the Time category. <i>value</i> is the name of a locale, OFF, or DEFAULT.
UVLC\$NUMERIC	Sets or disables the Numeric category. <i>value</i> is the name of a locale, OFF, or DEFAULT.
UVLC\$MONETARY	Sets or disables the Monetary category. <i>value</i> is the name of a locale, OFF, or DEFAULT.
UVLC\$CTYPE	Sets or disables the Ctype category. <i>value</i> is the name of a locale, OFF, or DEFAULT.
UVLC\$COLLATE	Sets or disables the Collate category. <i>value</i> is the name of a locale, OFF, or DEFAULT.
UVLC\$SAVE	Saves the current locale state, overwriting any previous saved locale. <i>value</i> is ignored.
UVLC\$RESTORE	Restores the saved locale state. <i>value</i> 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.

## SETLOCALE function

---

The STATUS function returns 0 if SETLOCALE is successful, or one of the following error tokens if it fails:

LCE\$NO.LOCALES	UniVerse locales are disabled.
LCE\$BAD.LOCALE	<i>value</i> 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	<i>value</i> 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 *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

---

## Syntax

SETREM *position* ON *dynamic.array*

## Description

Use the SETREM statement to set the remove pointer in *dynamic.array* to the position specified by *position*.

*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](#) and [REVREMOVE](#) statements 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](#) to ensure that *position* is at the start of a character. For more information about [locales](#), see *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
```



### Syntax

`SIN (expression)`

### Description

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.

### Example

```
PRINT SIN( 45 )
```

This is the program output:

```
0.7071
```

## SINH function

---

### Syntax

`SINH (expression)`

### Description

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.

### Example

```
PRINT "SINH(2) = ":SINH(2)
```

This is the program output:

```
SINH(2) = 3.6269
```

### Syntax

SLEEP [*seconds*]

### Description

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.

### 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

---

## Syntax

SMUL (*string.number.1*, *string.number.2*)

## Description

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.

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
```

### Syntax

SOUNDEX (*expression*)

### Description

The SOUNDEX function evaluates *expression* and returns the most significant letter in the input string followed by a phonetic code. Nonalphabetic 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.

### Example

Source Lines	Program Output
DATA "MCDONALD" , "MACDONALD" , "MACDOUGALL" FOR I=1 TO 3 INPUT CUSTOMER PHONETIC.CODE=SOUNDEX(CUSTOMER) PRINT PHONETIC.CODE NEXT	?MCDONALD M235 ?MACDONALD M235 ?MACDOUGALL M232

# SPACE function

---

## Syntax

SPACE (*expression*)

## Description

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.

## 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
```

### Syntax

SPACES (*dynamic.array*)

CALL –SPACES (*return.array*, *dynamic.array*)

CALL !SPACES (*return.array*, *dynamic.array*)

### Description

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*.

## SPLICE function

---

### Syntax

SPLICE (*array1*, *expression*, *array2*)

CALL –SPLICE (*return.array*, *array1*, *expression*, *array2*)

CALL !SPLICE (*return.array*, *array1*, *expression*, *array2*)

### Description

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*.

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-
```



### Syntax

`SQRT (expression)`

### Description

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.

### 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

---

### Syntax

SQUOTE (*expression*)

CALL !SQUOTE (*quoted.expression*, *expression*)

### Description

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.

*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.
```

### 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*]

### Description

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.

*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.

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.

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](#) or SSELECT commands in a BASIC [EXECUTE](#) state-

## SSELECT statements

---

ment. Processing the list by [READNEXT](#) 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 [SOPTIONS](#) 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 *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 [SOPTIONS](#) 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

---

### Syntax

SSUB (*string.number.1*, *string.number.2*)

### Description

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.

### Example

```
X = "123456"  
Y = "225"  
Z = SSUB (X,Y)  
PRINT Z
```

This is the program output:

```
123231
```

### Syntax

STATUS ( )

### Description

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.

The following sections describe STATUS function values.

#### After a **BSCAN** statement:

- 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.
- 10 An internal error was detected.

**After a DELETE statement:** After a **DELETE** statement with an ON ERROR clause, the value returned is the error number.

**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.

## STATUS function

---

### After an **FMT** function:

- 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:

- 0 The timeout limit expired.
- Any nonzero value A device input error occurred.

### After an **ICONV** or **OCONV** function:

- 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** statements).

### After a **MATWRITE**, **WRITE**, **WRITEU**, **WRITEV**, or **WRITEVU** statement:

- 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.



## STATUS function

---

**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	The filename was not found in the VOC file.
-2 <sup>1</sup>	The filename or file is null. This error may also occur when you cannot open a file across UV/Net.
-3	An operating system access error occurs when you do not have permission to access a UniVerse file in a directory. For example, this error may occur when trying to access a type 1 or type 30 file.
-4 <sup>1</sup>	An access error appears when you do not have operating system permissions or if DATA.30 is missing for a type 30 file.
-5	The operating system detected a read error.
-6	The lock file header cannot be unlocked.
-7	Invalid file revision or wrong byte-ordering exists for the platform.
-8 <sup>1</sup>	Invalid part file information exists.
-9 <sup>1</sup>	Invalid type 30 file information exists 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 exist to open the table.
-13 <sup>1</sup>	An index problem exists.
-14	The NFS file cannot be opened.

1. A generic error that can occur for various reasons.

**After a `READ` statement:** If the file is a distributed file, the STATUS function returns the following:

- 1 The partitioning algorithm does not evaluate to an integer.
- 2 The part number is invalid.

## STATUS function

---

### After a **READBLK** statement:

- 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.
- 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 unmapable character.
- 3 The unmapable character is in the record ID.
  - 4 The unmapable character is in the record's data.

### After a **READSEQ** statement:

- 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**, **RPC.CONNECT**, or **RPC.DISCONNECT** function:

- 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.

**After a SETLOCALE function:** The STATUS function returns 0 if SETLOCALE is successful, or one of the following error tokens if it fails:

## Example

## BASIC Statements and Functions

# STATUS statement

---

## Syntax

STATUS *dynamic.array* FROM *file.variable*  
{ THEN *statements* [ ELSE *statements* ] | ELSE *statements* }

## Description

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*.

The following table lists the values of the dynamic array returned by the STATUS statement:

STATUS Statement Values

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	Permissions (convert to octal). <b>Windows NT.</b> 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.
7	Number of hard links	0 if no links. <b>Windows NT.</b> The value is always 1 on non-NTFS partitions, > 0 on NTFS partitions.
8	User ID of owner	<b>UNIX.</b> The number assigned in <i>/etc/passwd</i> . <b>Windows NT.</b> It is a UniVerse pseudo user ID based on the user name and domain of the user.
9	Group ID of owner	<b>UNIX.</b> The number assigned in <i>/etc/passwd</i> . <b>Windows NT.</b> It is always 0.

## STATUS statement

---

**STATUS Statement Values (Continued)**

<b>Field</b>	<b>Stored Value</b>	<b>Description</b>
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 filename	The internal pathname 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.

## STATUS statement

---

STATUS Statement Values (Continued)

Field	Stored Value	Description
25	Pathnames of part files belonging to a distributed file	Multivalued list. If file is a part file, this field is empty.
26	Filenames of part files belonging to a distributed file	Multivalued list. If file is a part file, this field is empty.
27	Full pathname	The full pathname of the file. On Windows NT, the value begins with the UNC share name, if available; if not, the drive letter.
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.

*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.

### Example

```
OPENSEQ '/etc/passwd' TO test THEN PRINT "File Opened" ELSE  
ABORT
```

## STATUS statement

---

```
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

---

## Syntax

```
STOP [expression]  
STOPE [expression]  
STOPM [expression]
```

## Description

Use the STOP statement to terminate program execution and return system control to the calling environment, which can be a menu, a paragraph, another BASIC program, or the UniVerse command processor.

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.

## 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 [SOPTIONS](#) statement to get this behavior in IDEAL and INFORMATION flavor accounts.

## Example

```
PRINT "1+2=":1+2  
STOP "THIS IS THE END"
```



## STOP statement

---

This is the program output:

```
1+2=3
```

```
THIS IS THE END
```

## STORAGE statement

---

### Syntax

STORAGE *arg1 arg2 arg3*

### Description

The STORAGE statement performs no function. It is provided for compatibility with other Pick systems.

### Syntax

STR (*string*, *repeat*)

### Description

Use the STR function to produce a specified number of repetitions of a particular character string.

*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
HAHAHAHAHAHA
```

# STRS function

---

## Syntax

STRS (*dynamic.array*, *repeat*)

CALL –STRS (*return.array*, *dynamic.array*, *repeat*)

CALL !STRS (*return.array*, *dynamic.array*, *repeat*)

## Description

Use the STRS function to produce a dynamic array containing the specified number of repetitions of each element of *dynamic.array*.

*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
```

### Syntax

SUBR (*name*, [ *argument* [ , *argument* ... ] ] )

### Description

Use the SUBR function to return the value of an external subroutine. The SUBR function is commonly used in I-descriptors.

*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
```

## SUBR function

---

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

---

### Syntax

```
SUBROUTINE [name] [ ( ([MAT] variable [ , [MAT] variable ... ] ) ) ]
```

### Description

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 for more information.

### 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

---

### Syntax

SUBS (*array1*, *array2*)

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

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

### Description

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*.

### 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

---

### Syntax

SUBSTRINGS (*dynamic.array*, *start*, *length*)

CALL –SUBSTRINGS (*return.array*, *dynamic.array*, *start*, *length*)

CALL !SUBSTRINGS (*return.array*, *dynamic.array*, *start*, *length*)

### Description

Use the SUBSTRINGS function to create a dynamic array each of whose elements are substrings of the corresponding elements of *dynamic.array*.

*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

---

## Syntax

SUM (*dynamic.array*)

## Description

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.

## 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

# SUM function

Source Lines	Program Output
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

---

## Syntax

SUMMATION (*dynamic.array*)

CALL !SUMMATION (*result*, *dynamic.array*)

## Description

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.

*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
```

### Syntax

SYSTEM (*expression*)

### Description

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.

*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.

**SYSTEM Function Values**

Value	Action
1	Checks to see if the <a href="#">PRINTER ON</a> 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.
8, <i>n</i>	Checks whether the tape is attached. Returns the current block size if it is and -1 if it is not. <i>n</i> 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.

## SYSTEM function

---

### SYSTEM Function Values (Continued)

Value	Action
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 <a href="#">OPTIONS</a> ), 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 <b>Break</b> key is enabled. Returns 1 if the <b>Break</b> 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. <b>Windows NT.</b> This is the same value as 27.
29	Returns the group ID of the person using the routine. <b>Windows NT.</b> This value is 0.
30	Returns the effective group ID of the person using the routine. <b>Windows NT.</b> This value is 0.
31	Returns the UniVerse serial number.
32	Returns the location of the UV account directory.

## SYSTEM function

---

### SYSTEM Function Values (Continued)

Value	Action
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 pathname 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.
50	Returns the field number of the last <a href="#">READNEXT</a> statement when reading an exploded select list.
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.
91	Returns 0; on Windows NT, returns 1.
99	Returns the system time in the number of seconds since midnight Greenwich Mean Time (GMT), 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	<i>Reserved for future NLS extensions.</i>
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.

## SYSTEM function

---

### SYSTEM Function Values (Continued)

Value	Action
105	Returns a dynamic array, with field marks separating the elements, containing the current values of the <i>uvconfig</i> file parameters for NLS maps, otherwise returns 0. See the <a href="#">UVNLS.H</a> 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 <a href="#">SET.SEQ.MAP</a> command.
107	Returns the current map name for GCI string arguments unless overridden by a <a href="#">SET.GCI.MAP</a> command.
1001	Returns the UniVerse flavor: 1 for IDEAL, 2 for PICK, 4 for INFORMATION, 8 for REALITY, 16 for IN2, and 64 for PIOPEN.
1017	Returns the user's supplementary UNIX groups in a dynamic array.
1021	Returns the GCI error number.
1200, <i>hostname</i>	Returns the UV/Net link number associated with <i>hostname</i> . If there is an internal error adding <i>hostname</i> , 0 returns. <i>hostname</i> is an expression that contains the host name from a file opened through UV/Net. It refers to the host name portion of the file's pathname. For example, in the pathname ORION!/u1/filename, <i>hostname</i> is ORION.
1201, <i>hostname</i>	Returns the RPC connection number associated with <i>hostname</i> . The UV/Net REMOTE.B interface program uses this number. If there is an internal error adding <i>hostname</i> , or if RPC has not yet opened, 0 returns. If the RPC connection was opened but is now closed, -1 returns.
1202, <i>hostname</i>	Returns the timeout associated with <i>hostname</i> . If there is no timeout associated with <i>hostname</i> , 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.



## SYSTEM function

---

### 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 UV/Net connection to the system ORION:

```
TIMEOUT SYSTEM(1200, "ORION"), 30
```

## TABSTOP statement

---

### Syntax

TABSTOP *expression*

### Description

Use the TABSTOP statement to set the current tabstop width for **PRINT** statements. 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.

### Example

```
A="FIRST"  
B="LAST"  
PRINT A,B  
TABSTOP 15  
PRINT A,B
```

This is the program output:

```
FIRST  LAST  
FIRST      LAST
```

### Syntax

TAN (*expression*)

### Description

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.

### Example

```
PRINT TAN( 45 )
```

This is the program output:

```
1
```

# TANH function

---

## Syntax

TANH (*expression*)

## Description

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.

## Example

```
PRINT TANH ( 45 )
```

This is the program output:

```
1
```

### Syntax

TERMINFO (*argument*)

### Description

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](#).

*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 EQUATES

<hr/>		
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>
EQU LINE.FEED		TO terminfo\$<5>
EQU NEWLINE		TO terminfo\$<6>
<hr/>		

## TERMINFO function

---

### TERMINFO EQUATEs (Continued)

---

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>

---

## TERMINFO function

---

### TERMINFO EQUATEs (Continued)

---

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>
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	TO terminfo\$<56>
EQU SCROLL.DOWN.PARM	TO terminfo\$<57>
EQU CHANGE.SCROLL.REGION	TO terminfo\$<58>
EQU SCROLL.MODE.END	TO terminfo\$<59>
EQU SCROLL.MODE.BEGIN	TO terminfo\$<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>

---

## TERMINFO function

---

### TERMINFO EQUATEs (Continued)

---

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>
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>

---



## TERMINFO function

---

### TERMINFO EQUATEs (Continued)

---

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>
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>

---

## TERMINFO function

---

### TERMINFO EQUATEs (Continued)

---

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>
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>

---

## TERMINFO function

---

### TERMINFO EQUATEs (Continued)

---

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>

---

## TERMINFO function

---

### TERMINFO EQUATEs (Continued)

---

EQU KEYEDIT.ERASE.FIELD	TO terminfo\$<217>
EQU AT.NEGATIVE.1	TO terminfo\$<218>
EQU AT.NEGATIVE.2	TO terminfo\$<219>
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>

---

## TERMINFO function

---

### TERMINFO EQUATEs (Continued)

---

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>
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>
EQU AT.NEGATIVE.67	TO terminfo\$<284>
EQU AT.NEGATIVE.68	TO terminfo\$<285>
EQU AT.NEGATIVE.69	TO terminfo\$<286>

---

## TERMINFO function

---

### TERMINFO EQUATEs (Continued)

---

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>
EQU AT.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>
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>

---

## TERMINFO function

---

### TERMINFO EQUATEs (Continued)

---

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>
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>

---

## TERMINFO function

---

### TERMINFO EQUATEs (Continued)

---

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 function

---

### TERMINFO EQUATEs (Continued)

---

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>

---

## TERMINFO function

---

### TERMINFO EQUATEs (Continued)

---

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>
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>

---

## TERMINFO function

---

### TERMINFO EQUATEs (Continued)

---

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>
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>

---

## TERMINFO function

---

### TERMINFO EQUATEs (Continued)

---

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>
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>

---

## TERMINFO function

---

### TERMINFO EQUATEs (Continued)

---

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>
EQU LABEL.KEY.FUNCTION.25	TO terminfo\$<597>
EQU LABEL.KEY.FUNCTION.26	TO terminfo\$<598>
EQU LABEL.KEY.FUNCTION.27	TO terminfo\$<599>

---

## TERMINFO function

---

### TERMINFO EQUATEs (Continued)

---

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>

---

## TERMINFO function

---

### TERMINFO EQUATEs (Continued)

---

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

---

### Syntax

TIME ( )

### Description

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.

**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
```



### Syntax

TIMEDATE ( )

### Description

Use the TIMEDATE function to return the current system time and date in the following format:

*hh:mm:ss dd mmm yyyy*

<i>hh</i>	Hours (based on a 24-hour clock)
<i>mm</i>	Minutes
<i>ss</i>	Seconds
<i>dd</i>	Day
<i>mmm</i>	Month
<i>yyyy</i>	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 *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

---

## Syntax

TIMEOUT { *file.variable* | *link.number* }, *time*

## Description

Use the TIMEOUT statement to terminate a [READSEQ](#) 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 UV/Net 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.

*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 UV/Net connections.

*link.number* is the UV/Net link. It is a positive number from 1 through 255 (or the number set in the NET\_MAXCONNECT VALUE for UV/Net 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 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
```

## TIMEOUT statement

---

This is the program output:

```
TIMEOUT OCCURRED
```

The following example sets a 30-second timeout for the UV/Net connection to the system ORION:

```
TIMEOUT SYSTEM (1200, "ORION"), 30
OPEN "ORION!/ul/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

---

## Syntax

TPARM (*terminfo.string*, [*arg1*], [*arg2*], [*arg3*], [*arg4*], [*arg5*], [*arg6*], [*arg7*], [*arg8*])

## Description

Use the TPARM function to evaluate a parameterized terminfo string.

*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:

\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).
^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
<b>^A</b>	<b>^a</b>
<b>ASCII 1 (Ctrl-A)</b>	<b>ASCII 1 (Ctrl-A)</b>
<b>^@</b>	<b>ASCII 0</b>
<b>^[</b>	<b>ASCII 27 (Esc)</b>
<b>^\</b>	<b>ASCII 28</b>
<b>^]</b>	<b>ASCII 29</b>
<b>^^</b>	<b>ASCII 30</b>
<b>^_</b>	<b>ASCII 31</b>
<b>^?</b>	<b>ASCII 127 (Del)</b>

**\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:

**%pn** 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.

**%nd** Pop the top parameter off the stack, and output it as a decimal number in a field *n* characters wide.

**%0nd** Like **%nd**, 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.

## TPARM function

---

**%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.

**%= %< %>**    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.

**%Px**           Pop the stack, and put the result into variable *x*, where *x* is a lower-case letter (a – z).

**%gx**           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 %**

*Cn* are conditions, and *Bn* 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.

### Syntax

TPRINT [ON *print.channel*] [*print.list*]

### Description

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.

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](#), [FOOTING](#), [PAGE](#), 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 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 ( : ).

## TPRINT statement

---

If NLS is enabled, the TPRINT statement maps data in the same way as the [PRINT](#) statement. For more information about [maps](#), see *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
```



### Syntax

TRANS ( [ DICT ] *filename*, *record.ID*, *field#*, *control.code* )

### Description

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.

*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:

- 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.

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.

## TRANS 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
```

### 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
```

### Description

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 Chapter 4.

# TRANSACTION ABORT statement

---

## Syntax

TRANSACTION ABORT

## Description

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.

## 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
```

## TRANSACTION ABORT statement

---

```
' :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

---

## Syntax

TRANSACTION COMMIT

{ THEN *statements* [ ELSE *statements* ] | ELSE *statements* }

## Description

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.

## TRANSACTION START statement

---

### Syntax

TRANSACTION START

{ THEN *statements* [ ELSE *statements* ] | ELSE *statements* }

### Description

Use the TRANSACTION START statement to begin a new transaction.

### 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

---

## Syntax

TRIM (*expression* [ ,*character* [ ,*option* ] ] )

## Description

Use the TRIM function to remove unwanted characters in *expression*. 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:

- 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 *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

---

## Syntax

TRIMB (*expression*)

## Description

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 *UniVerse NLS Guide*.

## 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
```

### Syntax

TRIMBS (*dynamic.array*)

CALL –TRIMBS (*return.array*, *dynamic.array*)

### Description

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 *UniVerse NLS Guide*.

# TRIMF function

---

## Syntax

TRIMF (*expression*)

## Description

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 *UniVerse NLS Guide*.

## 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
```

### Syntax

TRIMFS (*dynamic.array*)

CALL –TRIMFS (*return.array*, *dynamic.array*)

### Description

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 *UniVerse NLS Guide*.

## TRIMS function

---

### Syntax

TRIMS (*dynamic.array*)

CALL –TRIMS (*return.array*, *dynamic.array*)

### Description

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 *UniVerse NLS Guide*.

### Syntax

```
TTYCTL file.variable, code#  
      { THEN statements [ ELSE statements ] | ELSE statements }
```

### Description

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.

The following table lists the available actions:

**TTYCTL Action Codes**

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.

## TTYCTL 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
```



### Syntax

```
TTYGET variable [FROM { file.variable | LPTR [n] | MTU [n] } ]  
                { THEN statements [ELSE statements] | ELSE statements }
```

### Description

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*.

*file.variable* is a terminal opened for sequential processing with the [OPENDEV](#) 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

## TTYGET statement

---

different operating systems: SV indicates System V, B indicates Berkeley UNIX, and NT indicates Windows NT.

### TTYGET Statement Values

Value	Name	Description	Avail- ability		
			S V	B	N T
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	✓ ✓ ✓ ✓	✓ ✓ ✓ ✓	 ✓ ✓ 
2	MODE.MIN	Minimum number of characters before input.	✓	✓	✓
3	MODE.TIME	Minimum time in milliseconds before input.	✓	✓	✓
Field 2					
1	CC.INTR	Interrupt character. -1 undefined.	✓	✓	✓
2	CC.QUIT	Quit character. -1 undefined.	✓	✓	
3	CC.SUSP	Suspend character. -1 undefined.	✓	✓	
4	CC.DSUSP	dsusp character. -1 undefined.		✓	
5	CC.SWITCH	Switch character. -1 undefined.	✓		
6	CC.ERASE	erase character. -1 undefined.	✓	✓	✓
7	CC.WERASE	werase character. -1 undefined.		✓	
8	CC.KILL	Kill character. -1 undefined.	✓	✓	✓
9	CC.LNEXT	lnext character. -1 undefined.		✓	
10	CC.RPRINT	rprint character. -1 undefined.		✓	✓
11	CC.EOF	eof character. -1 undefined.	✓	✓	
12	CC.EOL	eol character. -1 undefined.	✓	✓	
13	CC.EOL2	eol2 character. -1 undefined.	✓		
14	CC.FLUSH	Flush character. -1 undefined.		✓	

## TTYGET statement

### TTYGET Statement Values (Continued)

Value	Name	Description	Availability		
			S V	B	N T
15	CC.START	Start character. -1 undefined. On System V, ^Q only.	✓	✓	✓
16	CC.STOP	Stop character. -1 undefined. On System V, ^S only.	✓	✓	✓
17	CC.LCONT	lcont character. -1 undefined. Emulated only.	✓	✓	✓
18	CC.FMC	<i>fmc</i> character. -1 undefined. Emulated only.	✓	✓	✓
19	CC.VMC	<i>vmc</i> character. -1 undefined. Emulated only.	✓	✓	✓
20	CC.SMC	<i>smc</i> character. -1 undefined. Emulated only.	✓	✓	✓
21	CCDEL	Delete character.	✓	✓	
<b>Field 3</b>					
1	CARRIER.RECEIVE	Terminal can receive data.	✓	✓	✓
2	CARRIER.HANGUP	Hang up upon close of terminal.	✓	✓	
3	CARRIER.LOCAL	Terminal is a local line.	✓	✓	✓
<b>Field 4</b>					
1	CASE.UCIN	Convert lowercase to uppercase on input.	✓	✓	
2	CASE.UCOUT	Convert lowercase to uppercase on output.	✓	✓	
3	CASE.XCASE	Uppercase is preceded by a backslash ( \ ) to distinguish it from lowercase.	✓	✓	
4	CASE.INVERT	Invert case on input. Emulated only.	✓	✓	✓
<b>Field 5</b>					
1	CRMODE.INLCR	Convert LINEFEED to RETURN on input.	✓	✓	
2	CRMODE.IGNCR	Ignore RETURN on input.	✓	✓	
3	CRMODE.ICRNL	Convert RETURN to LINEFEED on input.	✓	✓	
4	CRMODE.ONLCR	Convert LINEFEED to LINEFEED, RETURN on output.	✓	✓	
5	CRMODE.OCRNL	Convert RETURN to LINEFEED on output.	✓	✓	

## TTYGET statement

### TTYGET Statement Values (Continued)

Value	Name	Description	Availability		
			S V	B	N T
6	CRMODE.ONOCR	Prohibit output of RETURN when cursor is in column 0.	✓	✓	
7	CRMODE.ONLRET	LINEFEED performs RETURN function.	✓	✓	
<b>Field 6</b>					
1	DELAY.BS	Set backspace delay.	✓	✓	
2	DELAY.CR	Set RETURN delay.	✓	✓	
3	DELAY.FF	Set formfeed delay.	✓	✓	
4	DELAY.LF	Set LINEFEED delay.	✓	✓	
5	DELAY.VT	Set vertical tab delay.	✓	✓	
6	DELAY.TAB	Set tab delay.	✓	✓	
7	DELAY.FILL	0 = time delay 1 = fill with empty strings 2 = fill with DELETES	✓	✓	
<b>Field 7</b>					
1	ECHO.ON	Set terminal echo on.	✓	✓	✓
2	ECHO.ERASE	ECHO\$ERASE or 0 = print echo character ECHO\$BS or 1 = echo as backspace ECHO\$BSB or 2 = echo as backspace, space, backspace ECHO\$PRINTER or 3 = echo as a printer	✓	✓	
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	✓	✓	
4	ECHO.CTRL	Set control to echo as ^ character	✓	✓	
5	ECHO.LF	When echo is off, echo RETURN as RETURN, LINEFEED	✓	✓	✓

## TTYGET statement

### TTYGET Statement Values (Continued)

Value	Name	Description	Avail-ability		
			S V	B	N T
Field 8					
1	HANDSHAKE.XON	1 = turns on X-ON/X-OFF protocol 0 = turns off X-ON/X-OFF protocol	✓	✓	✓
2	HANDSHAKE. STARTANY	1 = any characters acts as X-ON 0 = only X-ON character acts as X-ON	✓	✓	
3	HANDSHAKE. TANDEM	1 = when input buffer is nearly full, X-OFF is sent 0 = turns off automatic X-OFF, X-ON mode	✓	✓	✓
4	HANDSHAKE.DTR	1 = turns on DTR 0 = turns off DTR	✓	✓	
Field 9					
1	OUTPUT.POST	Output postprocessing occurs.	✓	✓	
2	OUTPUT.TILDE	Special output processing for tilde.	✓	✓	
3	OUTPUT.BG	Stop background processes at output.	✓	✓	
4	OUTPUT.CS	Output clearscreen before reports. Emulated only.	✓	✓	
5	OUTPUT.TAB	Set output tab expansion.	✓	✓	
Field 10					
1	PROTOCOL.LINE	Line protocol	✓	✓	
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	✓	✓	✓

## TTYGET statement

---

### TTYGET Statement Values (Continued)

Value	Name	Description	Avail- ability		
			S V	B	N T
3	PROTOCOL.DATA	Character size: 5 = 5 bits 7 = 7 bits 6 = 6 bits 8 = 8 bits	✓	✓	✓
4	PROTOCOL.STOP	2 = 2 stopbits 1 = 1 stopbit	✓	✓	✓
5	PROTOCOL.OUTPUT	Output parity: 0 = no parity 1 = even parity 2 = odd parity	✓	✓	✓
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	✓	✓	✓
7	PROTOCOL.STRIP	1 = strip to 7 bits 0 = 8 bits	✓	✓	
<b>Field 11</b>					
1	SIGNALS.ENABLE	Enable signal keys: Interrupt, Suspend, Quit.	✓	✓	
2	SIGNALS.FLUSH	Flush type-ahead buffer.	✓	✓	
3	SIGNALS.BRKKEY	0 = break ignored 1 = break as interrupt 2 = break as null	✓	✓	

### Syntax

```
TTYSET dynamic.array [ON { file.variable | LPTR [n] | MTU [n] } ]  
      { THEN statements [ELSE statements] | ELSE statements }
```

### Description

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.

*file.variable* is a terminal opened for sequential processing with the [OPENDEV](#) 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

## TTYSET statement

---

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](#)).

**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](#) statements 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.



### Syntax

UNASSIGNED (*variable*)

### Description

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.

### 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

---

### Syntax

UNICHAR (*unicode*)

### Description

Use the UNICHAR function to generate a single character from a Unicode value.

*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.

### Syntax

UNICHARS (*dynamic.array*)

### Description

Use the UNICHARS function to generate a dynamic array of characters from a dynamic array of Unicode values.

*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

---

### Syntax

UNISEQ (*expression*)

### Description

Use the UNISEQ function to generate a Unicode value from *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.

**CAUTION:** 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 *UniVerse NLS Guide*.

### Syntax

UNISEQS (*dynamic.array*)

### Description

Use the UNISEQS function to generate an array of Unicode values from a dynamic array of characters.

*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.

**CAUTION:** 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 *UniVerse NLS Guide*.

# UNLOCK statement

---

## Syntax

UNLOCK [*expression*]

## Description

Use the UNLOCK statement to release a process lock set by the LOCK statement.

*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
```

### Syntax

UPCASE (*expression*)

### Description

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 *UniVerse NLS Guide*.

### 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

---

## Syntax

UPRINT [ ON *print.channel* ] [ *print.list* ]

## Description

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.

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](#), [FOOTING](#), [PAGE](#), 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. 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 *UniVerse NLS Guide*.



### Syntax

```
WEOF [UNIT (mtu) ] { THEN statements [ ELSE statements ] | ELSE  
                        statements }
```

### Description

Use the WEOF statement to write an end-of-file (EOF) mark to tape.

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 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

---

## Syntax

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

## Description

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.

*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.

## WEOFSEQ statement

---

- 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](#), [READSEQ](#), and [WRITESEQ](#) statements 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

---

## 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]
```

## Description

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.

### Effects of WRITE 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. 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.

## WRITE statements

---

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](#) statement 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.

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.	ON ERROR	Record ID
4 The WRITE fails, no records written.		Data
Program terminates with a run-time error message.	No ON ERROR, and NLSWRITEELSE = 1	Record ID or data

## WRITE statements

---

STATUS Value and Results	ON ERROR and Parameter Setting	Unmappable Character Location
Program terminates with a run-time error message.	No ON ERROR, NLSWRITEELSE = 0	Record ID
Record is written with unknown characters; lost data.		Data

For more information about [unmappable characters](#), see *UniVerse NLS Guide*.

Use the [STATUS](#) function after a WRITE statement is executed, to determine the result of the operation, as follows:

- 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.

## WRITE statements

---

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](#) statement). To release the update record lock set by a READU statement and maintained by a WRITEU statement, you must use a [RELEASE](#), [WRITE](#), [MATWRITE](#), 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.

## WRITE 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.

### 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 UV/Net, 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
```



## WRITE statements

---

```
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
?ELLEN
ENTER YOUR LAST NAME
?KRANZER
ENTER YOUR ADDRESS (PLEASE WAIT FOR PROMPTS)
STREET ADDRESS
?3 AMES STREET
ENTER CITY
?CAMBRIDGE
ENTER STATE
?MA
ENTER ZIP CODE
?02139
ENTER YOUR INTEREST
?SAILING

SUN.MEMBER  LAST NAME.
1111 KRANZER

1 records listed.
```

# WRITEBLK statement

---

## Syntax

WRITEBLK *expression* ON *file.variable*  
    { THEN *statements* [ ELSE *statements* ] | ELSE *statements* }

## Description

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.

*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, 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 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 *UniVerse NLS Guide*.

## WRITEBLK statement

---

### 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

---

### Syntax

WRITELIST *dynamic.array* ON *listname*

### Description

Use the WRITELIST statement to save a list as a record in the [&SAVEDLISTS&](#) file.

*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.

### Syntax

WRITESEQ *expression* { ON | TO } *file.variable* [ON ERROR *statements*]  
          { THEN *statements* [ELSE *statements*] | ELSE *statements*}

### Description

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](#), [READBLK](#), WRITESEQ, and [WRITEBLK](#) statements.

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:

- 0   The record was locked before the WRITESEQ operation.
- 2   The record was unlocked before the WRITESEQ operation.

### 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.

## WRITESEQ statement

---

For more information about buffering, see the FLUSH and NOBUF 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 *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
```

## WRITESEQ statement

---

```
NEXT  
*  
  
CLOSESEQ FILE  
END
```

This is the program output:

```
?NEW ITEM 1  
NEW ITEM 1  
?NEW ITEM 2  
NEW ITEM 2
```

## WRITESEQF statement

---

### Syntax

WRITESEQF *expression* { ON | TO } *file.variable* [ ON ERROR *statements* ]  
          { THEN *statements* [ ELSE *statements* ] | ELSE *statements* }

### Description

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.

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 *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 *UniVerse NLS Guide*.



## WRITESEQF statement

---

### 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."
```

# WRITET statement

---

## Syntax

WRITET [UNIT (*mtu*)] *variable*  
    { THEN *statements* [ELSE *statements*] | ELSE *statements* }

## Description

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.

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:

***mtu* Codes**

Code	Available Options
<i>m</i> (mode)	0 = No conversion 1 = EBCDIC conversion 2 = Invert high bit 3 = Invert high bit and EBCDIC conversion
<i>t</i> (tracks)	0 = 9 tracks. Only 9-track tapes are supported.
<i>u</i> (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 right-most 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.

## WRITET statement

---

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 *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](#) statement.

## 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](#) statement.

## 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](#) statement.

### Syntax

XLATE ( [DICT] *filename*, *record.ID*, *field#*, *control.code*)

### Description

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.

*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:

- 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.

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.

## XLATE 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
```



### Syntax

`XTD (string)`

### Description

Use the XTD function to convert a string of hexadecimal characters to an integer. If *string* evaluates to the null value, null is returned.

### Example

```
Y = "0019"  
Z = XTD (Y)  
PRINT Z
```

This is the program output:

25



# 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

- |   |  |
|---|--|
| * | Identifies a line as a comment line. Same as the !, \$*, and REM statements. |
| ! | Identifies a line as a comment line. Same as the *, \$*, and REM statements. |

<b>#INCLUDE</b>	Inserts and compiles UniVerse BASIC source code from another program into the program being compiled. Same as the \$INCLUDE and INCLUDE statements.
<b>\$*</b>	Identifies a line as a comment line. Same as the *, !, and REM statements.
<b>\$CHAIN</b>	Inserts and compiles UniVerse BASIC source code from another program into the program being compiled.
<b>\$COPYRIGHT</b>	Inserts comments into the object code header. (UniVerse supports this statement for compatibility with existing software.)
<b>\$DEFINE</b>	Defines a compile time symbol.
<b>\$EJECT</b>	Begins a new page in the listing record. (UniVerse supports this statement for compatibility with existing software.) Same as the \$PAGE statement.
<b>\$IFDEF</b>	Tests for the definition of a compile time symbol.
<b>\$IFNDEF</b>	Tests for the definition of a compile time symbol.
<b>\$INCLUDE</b>	Inserts and compiles UniVerse BASIC source code from another program into the program being compiled. Same as the #INCLUDE and INCLUDE statements.
<b>\$INSERT</b>	Performs the same operation as \$INCLUDE; the only difference is in the syntax. (UniVerse supports this statement for compatibility with existing software.)
<b>\$MAP</b>	In NLS mode, specifies the map for the source code.
<b>\$OPTIONS</b>	Sets compile time emulation of UniVerse flavors.
<b>\$PAGE</b>	Begins a new page in the listing record. (UniVerse supports this statement for compatibility with existing software.) Same as the \$EJECT statement.
<b>EQUATE</b>	Assigns a symbol as the equivalent of a variable, function, number, character, or string.
<b>INCLUDE</b>	Inserts and includes the specified BASIC source code from another program into the program being compiled. Same as the #INCLUDE and \$INCLUDE statements.
<b>NULL</b>	Indicates that no operation is to be performed.

<b>REM</b>	Identifies a line as a comment line. Same as the *, !, and \$* statements.
<b>\$UNDEFINE</b>	Removes the definition for a compile time symbol.

## Declarations

<b>COMMON</b>	Defines a storage area in memory for variables commonly used by programs and external subroutines.
<b>DEFFUN</b>	Defines a user-written function.
<b>DIMENSION</b>	Declares the name, dimensionality, and size constraints of an array variable.
<b>FUNCTION</b>	Identifies a user-written function.
<b>PROGRAM</b>	Identifies a program.
<b>SUBROUTINE</b>	Identifies a series of statements as a subroutine.

## Assignments

<b>ASSIGNED()</b>	Determines if a variable is assigned a value.
<b>CLEAR</b>	Assigns a value of 0 to specified variables.
<b>LET</b>	Assigns a value to a variable.
<b>MAT</b>	Assigns a new value to every element of an array with one statement.
<b>UNASSIGNED()</b>	Determines if a variable is unassigned.

## Program Flow Control

<b>ABORT</b>	Terminates all programs and returns to the UniVerse command level.
<b>BEGIN CASE</b>	Indicates the beginning of a set of CASE statements.
<b>CALL</b>	Executes an external subroutine.
<b>CASE</b>	Alters program flow based on the results returned by expressions.

CHAIN	Terminates a BASIC program and executes a UniVerse command.
CONTINUE	Transfers control to the next logical iteration of a loop.
END	Indicates the end of a program or a block of statements.
END CASE	Indicates the end of a set of CASE statements.
ENTER	Executes an external subroutine.
EXECUTE	Executes UniVerse sentences and paragraphs from within the BASIC program.
EXIT	Quits execution of a LOOP...REPEAT loop and branches to the statement following the REPEAT statement.
FOR	Allows a series of instructions to be performed in a loop a given number of times.
GOSUB	Branches to and returns from an internal subroutine.
GOTO	Branches unconditionally to a specified statement within the program or subroutine.
IF	Determines program flow based on the evaluation of an expression.
LOOP	Repeatedly executes a sequence of statements under specified conditions.
NEXT	Defines the end of a FOR...NEXT loop.
ON	Transfers program control to a specified internal subroutine or to a specified statement, under specified conditions.
PERFORM	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	Repeatedly executes a sequence of statements under specified conditions.
RETURN	Transfers program control from an internal or external subroutine back to the calling program.
RETURN ( <i>value</i> )	Returns a value from a user-written function.
STOP	Terminates the current program.

<b>SUBR()</b>	Returns the value of an external subroutine.
<b>WHILE/UNTIL</b>	Provides conditions under which the LOOP...REPEAT statement or FOR...NEXT statement terminates.

## File I/O

### AUTHORIZATION

Specifies the effective run-time UID (user identification) number of the program.

### BEGIN TRANSACTION

Indicates the beginning of a set of statements that make up a single transaction.

### BSCAN

Scans the leaf-nodes of a B-tree file (type 25) or a secondary index.

### CLEARFILE

Erases all records from a file.

### CLOSE

Writes data written to the file physically on the disk and releases any file or update locks.

### COMMIT

Commits all changes made during a transaction, writing them to disk.

### DELETE

Deletes a record from a UniVerse file.

### DELETEU

Deletes a record from a previously opened file.

### END TRANSACTION

Indicates where execution should continue after a transaction terminates.

### FILELOCK

Sets a file update lock on an entire file to prevent other users from updating the file until this program releases it.

### FILEUNLOCK

Releases file locks set by the FILELOCK statement.

### INDICES()

Returns information about the secondary key indexes in a file.

### MATREAD

Assigns the data stored in successive fields of a record from a UniVerse file to the consecutive elements of an array.

### MATREADL

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.

<b>MATREADU</b>	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.
<b>MATWRITE</b>	Assigns the data stored in consecutive elements of an array to the successive fields of a record in a UniVerse file.
<b>MATWRITEU</b>	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.
<b>OPEN</b>	Opens a UniVerse file to be used in a BASIC program.
<b>OPENPATH</b>	Opens a file to be used in a BASIC program.
<b>PROCREAD</b>	Assigns the contents of the primary input buffer of the proc to a variable.
<b>PROCWRITE</b>	Writes the specified string to the primary input buffer of the proc that called your BASIC program.
<b>READ</b>	Assigns the contents of a record to a dynamic array variable.
<b>READL</b>	Sets a shared read lock on a record, then assigns the contents of the record to a dynamic array variable.
<b>READU</b>	Sets an exclusive update lock on a record, then assigns the contents of the record to a dynamic array variable.
<b>READV</b>	Assigns the contents of a field of a record to a dynamic array variable.
<b>READVL</b>	Sets a shared read lock on a record, then assigns the contents of a field of a record to a dynamic array variable.
<b>READVU</b>	Sets an exclusive update lock on a record, then assigns the contents of a field of the record to a dynamic array variable.
<b>RECORDLOCKED()</b>	Establishes whether or not a record is locked by a user.
<b>RECORDLOCKL</b>	Sets a shared read-only lock on a record in a file.
<b>RECORDLOCKU</b>	Locks the specified record to prevent other users from accessing it.
<b>RELEASE</b>	Unlocks records locked by READL, READU, READVL, READVU, MATREADL, MATREADU, MATWRITEV, WRITEV, or WRITEVU statements.



<b>ROLLBACK</b>	Rolls back all changes made during a transaction. No changes are written to disk.
<b>SET TRANSACTION ISOLATION LEVEL</b>	Sets the default transaction isolation level for your program.
<b>TRANS()</b>	Returns the contents of a field in a record of a UniVerse file.
<b>TRANSACTION ABORT</b>	Discards changes made during a transaction. No changes are written to disk.
<b>TRANSACTION COMMIT</b>	Commits all changes made during a transaction, writing them to disk.
<b>TRANSACTION START</b>	Indicates the beginning of a set of statements that make up a single transaction.
<b>WRITE</b>	Replaces the contents of a record in a UniVerse file.
<b>WRITEU</b>	Replaces the contents of the record in a UniVerse file without releasing the record lock.
<b>WRITEV</b>	Replaces the contents of a field of a record in a UniVerse file.
<b>WRITEVU</b>	Replaces the contents of a field in the record without releasing the record lock.
<b>XLATE()</b>	Returns the contents of a field in a record of a UniVerse file.

## Sequential File I/O

<b>CLOSESEQ</b>	Writes an end-of-file mark at the current location in the record and then makes the record available to other users.
<b>CREATE</b>	Creates a record in a UniVerse type 1 or type 19 file or establishes a path.
<b>FLUSH</b>	Immediately writes all buffers.
<b>GET</b>	Reads a block of data from an input stream associated with a device, such as a serial line or terminal.
<b>GETX</b>	Reads a block of data from an input stream associated with a device, and returns the characters in ASCII hexadecimal format.

<b>NOBUF</b>	Turns off buffering for a sequential file.
<b>OPENSEQ</b>	Prepares a UniVerse file for sequential use by the BASIC program.
<b>READBLK</b>	Reads a block of data from a UniVerse file open for sequential processing and assigns it to a variable.
<b>READSEQ</b>	Reads a line of data from a UniVerse file opened for sequential processing and assigns it to a variable.
<b>SEND</b>	Writes a block of data to a device that has been opened for I/O using OPENDEV or OPENSEQ.
<b>STATUS</b>	Determines the status of a UniVerse file open for sequential processing.
<b>TIMEOUT</b>	Terminates READSEQ or READBLK if no data is read in the specified time.
<b>TTYCTL</b>	Controls sequential file interaction with a terminal device.
<b>TTYGET</b>	Gets a dynamic array of the terminal characteristics of a terminal, line printer channel, or magnetic tape channel.
<b>TTYSET</b>	Sets the terminal characteristics of a terminal, line printer channel, or magnetic tape channel.
<b>WEOFSEQ</b>	Writes an end-of-file mark to a UniVerse file open for sequential processing at the current position.
<b>WRITEBLK</b>	Writes a block of data to a record in a sequential file.
<b>WRITESEQ</b>	Writes new values to the specified record of a UniVerse file sequentially.
<b>WRITESEQF</b>	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

<b>@()</b>	Returns an escape sequence used for terminal control.
<b>BREAK</b>	Enables or disables the <b>Break</b> key on the keyboard.
<b>CLEARDATA</b>	Clears all data previously stored by the DATA statement.
<b>CRT</b>	Outputs data to the screen.
<b>DATA</b>	Stores values to be used in subsequent requests for data input.

DISPLAY	Outputs data to the screen.
ECHO	Controls the display of input characters on the terminal screen.
FOOTING	Specifies text to be printed at the bottom of each page.
HEADING	Specifies text to be printed at the top of each page.
HUSH	Suppresses all text normally sent to a terminal during processing.
INPUT	Allows data input from the keyboard during program execution.
INPUT @	Positions the cursor at a specified location and defines the length of the input field.
INPUTCLEAR	Clears the type-ahead buffer.
INPUTDISP @	Positions the cursor at a specified location and defines a format for the variable to print.
INPUTERR	Prints a formatted error message from the ERRMSG file on the bottom line of the terminal.
INPUTNULL	Defines a single character to be recognized as the empty string in an INPUT @ statement.
INPUTTRAP	Branches to a program label or subroutine on a TRAP key.
KEYEDIT	Assigns specific editing functions to the keys on the keyboard to be used with the INPUT statement.
KEYEXIT	Specifies exit traps for the keys assigned editing functions by the KEYEDIT statement.
KEYIN()	Reads a single character from the input buffer and returns it.
KEYTRAP	Specifies traps for the keys assigned specific functions by the KEYEDIT statement.
OPENDEV	Opens a device for input or output.
PAGE	Prints a footing at the bottom of the page, advances to the next page, and prints a heading at the top.
PRINT	Outputs data to the terminal screen or to a printer.

<b>PRINTER CLOSE</b>	Indicates the completion of a print file and readiness for the data stored in the system buffer to be printed on the line printer.
<b>PRINTER ON   OFF</b>	Indicates whether print file 0 is to output to the terminal screen or to the line printer.
<b>PRINTER RESET</b>	Resets the printing options.
<b>PRINTERR</b>	Prints a formatted error message from the ERRMSG file on the bottom line of the terminal.
<b>PROMPT</b>	Defines the prompt character for user input.
<b>TABSTOP</b>	Sets the current tabstop width for PRINT statements.
<b>TERMINFO()</b>	Accesses the information contained in the <i>terminfo</i> files.
<b>TPARM()</b>	Evaluates a parameterized <i>terminfo</i> string.
<b>TPRINT</b>	Sends data with delays to the screen, a line printer, or another specified print file (that is, a logical printer).

## Tape I/O

<b>READT</b>	Assigns the contents of the next record from a magnetic tape unit to the named variable.
<b>REWIND</b>	Rewinds the magnetic tape to the beginning of the tape.
<b>WEOF</b>	Writes an end-of-file mark to a magnetic tape.
<b>WRITET</b>	Writes the contents of a record onto magnetic tape.

## Select Lists

<b>CLEARSELECT</b>	Sets a select list to empty.
<b>DELETELIST</b>	Deletes a select list saved in the &SAVEDLISTS& file.
<b>GETLIST</b>	Activates a saved select list so it can be used by a READNEXT statement.
<b>READLIST</b>	Assigns an active select list to a variable.
<b>READNEXT</b>	Assigns the next record ID from an active select list to a variable.

<b>SELECT</b>	Creates a list of all record IDs in a UniVerse file for use by a subsequent READNEXT statement.
<b>SELECTE</b>	Assigns the contents of select list 0 to a variable.
<b>SELECTINDEX</b>	Creates select lists from secondary key indexes.
<b>SELECTINFO()</b>	Returns the activity status of a select list.
<b>SSELECT</b>	Creates a sorted list of all record IDs from a UniVerse file.
<b>WRITELIST</b>	Saves a list as a record in the &SAVEDLISTS& file.

## String Handling

<b>ALPHA()</b>	Determines whether the expression is an alphabetic or nonalphabetic string.
<b>CATS()</b>	Concatenates elements of two dynamic arrays.
<b>CHANGE()</b>	Substitutes an element of a string with a replacement element.
<b>CHECKSUM()</b>	Returns a cyclical redundancy code (a checksum value).
<b>COL1()</b>	Returns the column position immediately preceding the selected substring after a BASIC FIELD function is executed.
<b>COL2()</b>	Returns the column position immediately following the selected substring after a BASIC FIELD function is executed.
<b>COMPARE()</b>	Compares two strings for sorting.
<b>CONVERT</b>	Converts specified characters in a string to designated replacement characters.
<b>CONVERT()</b>	Replaces every occurrence of specified characters in a variable with other specified characters.
<b>COUNT()</b>	Evaluates the number of times a substring is repeated in a string.
<b>COUNTS()</b>	Evaluates the number of times a substring is repeated in each element of a dynamic array.
<b>DCOUNT()</b>	Evaluates the number of delimited fields contained in a string.
<b>DEL</b>	Deletes the specified field, value, or subvalue from a dynamic array.
<b>DELETE()</b>	Deletes a field, value, or subvalue from a dynamic array.

<b>DOWNCASE()</b>	Converts all uppercase letters in an expression to lowercase.
<b>DQUOTE()</b>	Encloses an expression in double quotation marks.
<b>EREPLACE()</b>	Substitutes an element of a string with a replacement element.
<b>EXCHANGE()</b>	Replaces one character with another or deletes all occurrences of a specific character.
<b>EXTRACT()</b>	Extracts the contents of a specified field, value, or subvalue from a dynamic array.
<b>FIELD()</b>	Examines a string expression for any occurrence of a specified delimiter and returns a substring that is marked by that delimiter.
<b>FIELDS()</b>	Examines each element of a dynamic array for any occurrence of a specified delimiter and returns substrings that are marked by that delimiter.
<b>FIELDSTORE()</b>	Replaces, deletes, or inserts substrings in a specified character string.
<b>FIND</b>	Locates a given occurrence of an element within a dynamic array.
<b>FINDSTR</b>	Locates a given occurrence of a substring.
<b>FOLD()</b>	Divides a string into a number of shorter sections.
<b>GETREM()</b>	Returns the numeric value for the position of the REMOVE pointer associated with a dynamic array.
<b>GROUP()</b>	Returns a substring that is located between the stated number of occurrences of a delimiter.
<b>GROUPSTORE</b>	Modifies existing character strings by inserting, deleting, or replacing substrings that are separated by a delimiter character.
<b>INDEX()</b>	Returns the starting column position of a specified occurrence of a particular substring within a string expression.
<b>INDEXS()</b>	Returns the starting column position of a specified occurrence of a particular substring within each element of a dynamic array.
<b>INS</b>	Inserts a specified field, value, or subvalue into a dynamic array.

<b>INSERT()</b>	Inserts a field, value, or subvalue into a dynamic array.
<b>LEFT()</b>	Specifies a substring consisting of the first <i>n</i> characters of a string.
<b>LEN()</b>	Calculates the length of a string.
<b>LENS()</b>	Calculates the length of each element of a dynamic array.
<b>LOCATE</b>	Searches a dynamic array for a particular value or string, and returns the index of its position.
<b>LOWER()</b>	Converts system delimiters that appear in expressions to the next lower-level delimiter.
<b>MATBUILD</b>	Builds a string by concatenating the elements of an array.
<b>MATCHFIELD()</b>	Returns the contents of a substring that matches a specified pattern or part of a pattern.
<b>MATPARSE</b>	Assigns the elements of an array from the elements of a dynamic array.
<b>QUOTE()</b>	Encloses an expression in double quotation marks.
<b>RAISE()</b>	Converts system delimiters that appear in expressions to the next higher-level delimiter.
<b>REMOVE</b>	Removes substrings from a dynamic array.
<b>REMOVE()</b>	Successively removes elements from a dynamic array. Extracts successive fields, values, etc., for dynamic array processing.
<b>REVREMOVE</b>	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.
<b>REPLACE()</b>	Replaces all or part of the contents of a dynamic array.
<b>REUSE()</b>	Reuses the last value in the shorter of two multivalued lists in a dynamic array operation.
<b>RIGHT()</b>	Specifies a substring consisting of the last <i>n</i> characters of a string.
<b>SETREM</b>	Sets the position of the REMOVE pointer associated with a dynamic array.
<b>SOUNDEX()</b>	Returns the soundex code for a string.

<b>SPACE()</b>	Generates a string consisting of a specified number of blank spaces.
<b>SPACES()</b>	Generates a dynamic array consisting of a specified number of blank spaces for each element.
<b>SPLICE()</b>	Inserts a string between the concatenated values of corresponding elements of two dynamic arrays.
<b>SQUOTE()</b>	Encloses an expression in single quotation marks.
<b>STR()</b>	Generates a particular character string a specified number of times.
<b>STRS()</b>	Generates a dynamic array whose elements consist of a character string repeated a specified number of times.
<b>SUBSTRINGS()</b>	Creates a dynamic array consisting of substrings of the elements of another dynamic array.
<b>TRIM()</b>	Deletes extra blank spaces and tabs from a character string.
<b>TRIMB()</b>	Deletes all blank spaces and tabs after the last nonblank character in an expression.
<b>TRIMBS()</b>	Deletes all trailing blank spaces and tabs from each element of a dynamic array.
<b>TRIMF()</b>	Deletes all blank spaces and tabs up to the first nonblank character in an expression.
<b>TRIMFS()</b>	Deletes all leading blank spaces and tabs from each element of a dynamic array.
<b>TRIMS()</b>	Deletes extra blank spaces and tabs from the elements of a dynamic array.
<b>UPCASE()</b>	Converts all lowercase letters in an expression to uppercase.

## Data Conversion and Formatting

<b>ASCII()</b>	Converts EBCDIC representation of character string data to the equivalent ASCII character code values.
<b>CHAR()</b>	Converts a numeric value to its ASCII character string equivalent.
<b>CHARS()</b>	Converts numeric elements of a dynamic array to their ASCII character string equivalents.



<b>DTX()</b>	Converts a decimal integer into its hexadecimal equivalent.
<b>EBCDIC()</b>	Converts data from its ASCII representation to the equivalent code value in EBCDIC.
<b>FIX()</b>	Rounds an expression to a decimal number having the accuracy specified by the PRECISION statement.
<b>FMT()</b>	Converts data from its internal representation to a specified format for output.
<b>FMTS()</b>	Converts elements of a dynamic array from their internal representation to a specified format for output.
<b>ICONV()</b>	Converts data to internal storage format.
<b>ICONVS()</b>	Converts elements of a dynamic array to internal storage format.
<b>OCONV()</b>	Converts data from its internal representation to an external output format.
<b>OCONVS()</b>	Converts elements of a dynamic array from their internal representation to an external output format.
<b>PRECISION</b>	Sets the maximum number of decimal places allowed in the conversion from the internal binary format of a numeric value to the string representation.
<b>SEQ()</b>	Converts an ASCII character code value to its corresponding numeric value.
<b>SEQS()</b>	Converts each element of a dynamic array from an ASCII character code to a corresponding numeric value.
<b>XTD()</b>	Converts a hexadecimal string into its decimal equivalent.

## **NLS**

<b>\$MAP</b>	Directs the compiler to specify the map for the source code.
<b>AUXMAP</b>	Assigns the map for the auxiliary printer to print unit 0 (i.e., the terminal).
<b>BYTE()</b>	Generates a string made up of a single byte.
<b>BYTELEN()</b>	Generates the number of bytes contained in the string value in an expression.

<b>BYTETYPE()</b>	Determines the function of a byte in a character.
<b>BYTEVAL()</b>	Retrieves the value of a byte in a string value in an expression.
<b>FMTDP()</b>	Formats data for output in display positions rather than character lengths.
<b>FMTSDP()</b>	Formats elements of a dynamic array for output in display positions rather than character lengths.
<b>FOLDDP()</b>	Divides a string into a number of substrings separated by field marks, in display positions rather than character lengths.
<b>GETLOCALE()</b>	Retrieves the names of specified categories of the current locale.
<b>INPUTDP</b>	Lets the user enter data in display positions rather than character lengths.
<b>LENDP()</b>	Returns the number of display positions in a string.
<b>LENSDP()</b>	Returns a dynamic array of the number of display positions in each element of a dynamic array.
<b>LOCALEINFO()</b>	Retrieves the settings of the current locale.
<b>SETLOCALE()</b>	Changes the setting of one or all categories for the current locale.
<b>UNICHAR()</b>	Generates a character from a Unicode integer value.
<b>UNICHARS()</b>	Generates a dynamic array from an array of Unicode values.
<b>UNISEQ()</b>	Generates a Unicode integer value from a character.
<b>UNISEQS()</b>	Generates an array of Unicode values from a dynamic array.
<b>UPRINT</b>	Prints data without performing any mapping. Typically used with data that has already been mapped using OCONV ( <i>mapname</i> ).

## Mathematical Functions

<b>ABS()</b>	Returns the absolute (positive) numeric value of an expression.
<b>ABSS()</b>	Creates a dynamic array containing the absolute values of a dynamic array.
<b>ACOS()</b>	Calculates the trigonometric arc-cosine of an expression.

<code>ADDS()</code>	Adds elements of two dynamic arrays.
<code>ASIN()</code>	Calculates the trigonometric arc-sine of an expression.
<code>ATAN()</code>	Calculates the trigonometric arctangent of an expression.
<code>BITAND()</code>	Performs a bitwise AND of two integers.
<code>BITNOT()</code>	Performs a bitwise NOT of two integers.
<code>BITOR()</code>	Performs a bitwise OR of two integers.
<code>BITRESET()</code>	Resets one bit of an integer.
<code>BITSET()</code>	Sets one bit of an integer.
<code>BITTEST()</code>	Tests one bit of an integer.
<code>BITXOR()</code>	Performs a bitwise XOR of two integers.
<code>COS()</code>	Calculates the trigonometric cosine of an angle.
<code>COSH()</code>	Calculates the hyperbolic cosine of an expression.
<code>DIV()</code>	Outputs the whole part of the real division of two real numbers.
<code>DIVS()</code>	Divides elements of two dynamic arrays.
<code>EXP()</code>	Calculates the result of base "e" raised to the power designated by the value of the expression.
<code>INT()</code>	Calculates the integer numeric value of an expression.
<code>FADD()</code>	Performs floating-point addition on two numeric values. This function is provided for compatibility with existing software.
<code>FDIV()</code>	Performs floating-point division on two numeric values.
<code>FFIX()</code>	Converts a floating-point number to a string with a fixed precision. FFIX is provided for compatibility with existing software.
<code>FFLT()</code>	Rounds a number to a string with a precision of 14.
<code>FMUL()</code>	Performs floating-point multiplication on two numeric values. This function is provided for compatibility with existing software.
<code>FSUB()</code>	Performs floating-point subtraction on two numeric values.
<code>LN()</code>	Calculates the natural logarithm of an expression in base "e".

<b>MAXIMUM()</b>	Returns the element with the highest numeric value in a dynamic array.
<b>MINIMUM()</b>	Returns the element with the lowest numeric value in a dynamic array.
<b>MOD()</b>	Calculates the modulo (the remainder) of two expressions.
<b>MODS()</b>	Calculates the modulo (the remainder) of elements of two dynamic arrays.
<b>MULS()</b>	Multiplies elements of two dynamic arrays.
<b>NEG()</b>	Returns the arithmetic additive inverse of the value of the argument.
<b>NEGS()</b>	Returns the negative numeric values of elements in a dynamic array. If the value of an element is negative, the returned value is positive.
<b>NUM()</b>	Returns true (1) if the argument is a numeric data type; otherwise, returns false (0).
<b>NUMS()</b>	Returns true (1) for each element of a dynamic array that is a numeric data type; otherwise, returns false (0).
<b>PWR()</b>	Calculates the value of an expression when raised to a specified power.
<b>RANDOMIZE</b>	Initializes the RND function to ensure that the same sequence of random numbers is generated after initialization.
<b>REAL()</b>	Converts a numeric expression into a real number without loss of accuracy.
<b>REM()</b>	Calculates the value of the remainder after integer division is performed.
<b>RND()</b>	Generates a random number between zero and a specified number minus one.
<b>SADD()</b>	Adds two string numbers and returns the result as a string number.
<b>SCMP()</b>	Compares two string numbers.
<b>SDIV()</b>	Outputs the quotient of the whole division of two integers.
<b>SIN()</b>	Calculates the trigonometric sine of an angle.
<b>SINH()</b>	Calculates the hyperbolic sine of an expression.

<b>SMUL()</b>	Multiplies two string numbers.
<b>SQRT()</b>	Calculates the square root of a number.
<b>SSUB()</b>	Subtracts one string number from another and returns the result as a string number.
<b>SUBS()</b>	Subtracts elements of two dynamic arrays.
<b>SUM()</b>	Calculates the sum of numeric data within a dynamic array.
<b>SUMMATION()</b>	Adds the elements of a dynamic array.
<b>TAN()</b>	Calculates the trigonometric tangent of an angle.
<b>TANH()</b>	Calculates the hyperbolic tangent of an expression.

## Relational Functions

<b>ANDS()</b>	Performs a logical AND on elements of two dynamic arrays.
<b>EQS()</b>	Compares the equality of corresponding elements of two dynamic arrays.
<b>GES()</b>	Indicates when elements of one dynamic array are greater than or equal to corresponding elements of another dynamic array.
<b>GTS()</b>	Indicates when elements of one dynamic array are greater than corresponding elements of another dynamic array.
<b>IFS()</b>	Evaluates a dynamic array and creates another dynamic array on the basis of the truth or falsity of its elements.
<b>ISNULL()</b>	Indicates when a variable is the null value.
<b>ISNULLS()</b>	Indicates when an element of a dynamic array is the null value.
<b>LES()</b>	Indicates when elements of one dynamic array are less than or equal to corresponding elements of another dynamic array.
<b>LTS()</b>	Indicates when elements of one dynamic array are less than corresponding elements of another dynamic array.
<b>NES()</b>	Indicates when elements of one dynamic array are not equal to corresponding elements of another dynamic array.
<b>NOT()</b>	Returns the complement of the logical value of an expression.

<b>NOTS()</b>	Returns the complement of the logical value of each element of a dynamic array.
<b>ORS()</b>	Performs a logical OR on elements of two dynamic arrays.

## System

<b>DATE()</b>	Returns the internal system date.
<b>DEBUG</b>	Invokes RAID, the interactive UniVerse BASIC debugger.
<b>ERRMSG</b>	Prints a formatted error message from the ERRMSG file.
<b>INMAT()</b>	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.
<b>ITYPE()</b>	Returns the value resulting from the evaluation of an I-descriptor.
<b>LOCK</b>	Sets an execution lock to protect user-defined resources or events from being used by more than one concurrently running program.
<b>NAP</b>	Suspends execution of a BASIC program, pausing for a specified number of milliseconds.
<b>SENTENCE()</b>	Returns the stored sentence that invoked the current process.
<b>SLEEP</b>	Suspends execution of a BASIC program, pausing for a specified number of seconds.
<b>STATUS()</b>	Reports the results of a function or statement previously executed.
<b>SYSTEM()</b>	Checks the status of a system function.
<b>TIME()</b>	Returns the time in internal format.
<b>TIMEDATE()</b>	Returns the time and date.
<b>UNLOCK</b>	Releases an execution lock that was set with the LOCK statement.

## Remote Procedure Calls

- RPC.CALL()** Sends requests to a remote server.
- RPC.CONNECT()** Establishes a connection with a remote server process.
- RPC.DISCONNECT()** Disconnects from a remote server process.

## Miscellaneous

- CLEARPROMPTS** Clears the value of the in-line prompt.
- EOF(ARG.)** Checks whether the command line argument pointer is past the last command line argument.
- FILEINFO()** Returns information about the specified file's configuration.
- ILPROMPT()** Evaluates strings containing in-line prompts.
- GET(ARG.)** Retrieves a command line argument.
- SEEK(ARG.)** Moves the command line argument pointer.





# B

## ASCII and Hex Equivalents

Table B-1 lists binary, octal, hexadecimal, and ASCII equivalents of decimal numbers.

**Table B-1. ASCII 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

**Table B-1. ASCII Equivalents (Continued)**

<b>Decimal</b>	<b>Binary</b>	<b>Octal</b>	<b>Hexadecimal</b>	<b>ASCII</b>
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	'
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

**Table B-1. ASCII Equivalents (Continued)**

<b>Decimal</b>	<b>Binary</b>	<b>Octal</b>	<b>Hexadecimal</b>	<b>ASCII</b>
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
083	01010011	123	53	S
084	01010100	124	54	T
085	01010101	125	55	U
086	01010110	126	56	V

**Table B-1. ASCII Equivalents (Continued)**

<b>Decimal</b>	<b>Binary</b>	<b>Octal</b>	<b>Hexadecimal</b>	<b>ASCII</b>
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

**Table B-1. ASCII Equivalents (Continued)**

Decimal	Binary	Octal	Hexadecimal	ASCII
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	}
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

Table B-2 provides additional hexadecimal and decimal equivalents.

**Table B-2. 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



# 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 and both the [ICONV](#) function and [OCONV](#) function sections 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.

Table C-1 lists correlative and conversion codes.

**Table C-1. Correlative and Conversion Codes**

Code	Description
<a href="#">A</a>	Algebraic functions
<a href="#">BB</a>	Bit conversion (binary)
<a href="#">BX</a>	Bit conversion (hexadecimal)
<a href="#">C</a>	Concatenation
<a href="#">D</a>	Date conversion
<a href="#">DI</a>	International date conversion
<a href="#">ECS</a>	Extended character set conversion

**Table C-1. Correlative and Conversion Codes (Continued)**

<b>Code</b>	<b>Description</b>
<b>F</b>	Mathematical functions
<b>G</b>	Group extraction
<b>L</b>	Length function
<b>MB</b>	Binary conversion
<b>MCA</b>	Masked alphabetic conversion
<b>MC/A</b>	Masked nonalphabetic conversion
<b>MCD</b>	Decimal to hexadecimal conversion
<b>MCDX</b>	Decimal to hexadecimal conversion
<b>MCL</b>	Masked lowercase conversion
<b>MCM</b>	Masked multibyte conversion
<b>MC/M</b>	Masked single-byte conversion
<b>MCN</b>	Masked numeric conversion
<b>MC/N</b>	Masked nonnumeric conversion
<b>MCP</b>	Masked unprintable character conversion
<b>MCT</b>	Masked initial capitals conversion
<b>MCU</b>	Masked uppercase conversion
<b>MCW</b>	Masked wide-character conversion
<b>MCX</b>	Hexadecimal to decimal conversion
<b>MCXD</b>	Hexadecimal to decimal conversion
<b>MD</b>	Masked decimal conversion
<b>ML</b>	Masked left conversion
<b>MM</b>	NLS monetary conversion
<b>MO</b>	Octal conversion
<b>MP</b>	Packed decimal conversion
<b>MR</b>	Masked right conversion
<b>MT</b>	Time conversion
<b>MU0C</b>	Hexadecimal Unicode character conversion
<b>MX</b>	Hexadecimal conversion
<b>MY</b>	ASCII conversion
<b>NL</b>	NLS Arabic numeral conversion
<b>NLSmapname</b>	Conversion using NLS map name



**Table C-1. Correlative and Conversion Codes (Continued)**

<b>Code</b>	<b>Description</b>
NR	Roman numeral conversion
P	Pattern matching
Q	Exponential conversion
R	Range function
S	Soundex
S	Substitution
T	Text extraction
T <i>filename</i>	File translation
TI	International time conversion

# A code: Algebraic Functions

---

## Format

A [ ; ] *expression*

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.

*expression* can be one or more of the following:

### A data location or string

<i>loc</i> [R]	Field number specifying a data value, followed by an optional R (repeat code).
N( <i>name</i> )	<i>name</i> 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 <i>name</i> are applied to the field defined by <i>name</i> , and the converted value is processed by the A code.
<i>string</i>	Literal string enclosed in pairs of double quotation marks ( " ), single quotation marks ( ' ), or backslashes ( \ ).
<i>number</i>	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

@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	<b>Load Previous Value:</b> load the result of the last correlative or conversion onto the stack.

## A code: Algebraic Functions

---

### A function

$R(exp)$	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 <i>exp</i> . 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.
$IF(expression) \mid THEN(expression) \mid ELSE(expression)$	A conditional expression.
$(conv)$	Conversion expression in parentheses (except A and F conversions).

### An arithmetic operator

$*$	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

$=$	Equal to
$<$	Less than
$>$	Greater than
$\#$ or $<>$	Not equal to
$<=$	Less than or equal to
$>=$	Greater than or equal to

### A conditional operator

AND	Logical AND
OR	Logical OR

## A code: Algebraic Functions

---

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.

## A code: Algebraic Functions

---

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

---

### Formats

BB	Binary conversion (base 2)
BX	Hexadecimal conversion (base 16)

The BB and BX codes convert data from external binary and hexadecimal format to internal bit string format and vice versa.

Characters outside of the range for each of the bases produce conversion errors. The ranges are as follows:

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.

### Format

`C [ ; ] expression1 c expression2 [ c expression3 ] ...`

The C code chains together field values or quoted strings, or both.

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 = "oneftwothreevfour":

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
------------------------------	--------

## C code: Concatenation

---

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.



### Format

`D [n] [*m] [s] [fmt [f1, f2, f3, f4, f5]] [E] [L]`

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.

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 [ICONV](#) or [OCONV](#) functions 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.

*n* Single digit (normally 1 through 4) that specifies the number of digits of the year to output. The default is 4.

*\** 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.

## D code: Date Conversion

---

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.

*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.

## D code: Date Conversion

---

- 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.

[ *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.

## D code: Date Conversion

---

*'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]

## D code: Date Conversion

---

Each format modifier must correspond to a format option. The following table shows which modifiers can modify which options:

Format Modifier	Format Option				
	D	M	Y	J	W
A	no	yes	yes	no	yes
<i>n</i>	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 **ICONV** is the same as for **OCONV**, except that:

- 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()** 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.

## D code: Date Conversion

---

### 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

---

### Format

DI

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.

## ECS code: Extended Character Set Conversion

---

### Format

ECS

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](#) or [OCONV](#) function.



## F code: Mathematical Functions

---

### Format

F [ ; ] *element* [ ; *element* ... ]

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.

The semicolon ( ; ) is the element separator.

*element* can be one or more of the items from the following categories:

#### A data location or string

<i>loc</i> [R]	Numeric location specifying a data value to be pushed onto the stack, optionally followed by an R (repeat code).
<i>Cn</i>	<i>n</i> is a constant to be pushed onto the stack.
<i>string</i>	Literal string enclosed in pairs of double quotation marks ( " ), single quotation marks ( ' ), or backslashes ( \ ).
<i>number</i>	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

@NI	Current item 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	<b>Load Previous Value:</b> load the result of the last correlative code onto the stack.

## F code: Mathematical Functions

---

### 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.

*[ <i>n</i> ]	Multiply <i>stack1</i> by <i>stack2</i> . The optional <i>n</i> is the descaling factor (that is, the result is divided by 10 raised to the <i>n</i> th power).
/	Divide <i>stack1</i> into <i>stack2</i> , result to <i>stack1</i> .
R	Same as /, but instead of the quotient, the remainder is returned to the top of the stack.
+	Add <i>stack1</i> to <i>stack2</i> .
-	Subtract <i>stack1</i> from <i>stack2</i> , result to <i>stack1</i> (except for REALITY flavor, which subtracts <i>stack2</i> from <i>stack1</i> ).
:	Concatenate <i>stack1</i> string onto the end of <i>stack2</i> string.
[ ]	Extract substring. <i>stack3</i> string is extracted, starting at the character specified by <i>stack2</i> and continuing for the number of characters specified in <i>stack1</i> . This is equivalent to the BASIC [ <i>m</i> , <i>n</i> ] operator, where <i>m</i> is in <i>stack2</i> and <i>n</i> is in <i>stack1</i> .
S	Sum of multivalues in <i>stack1</i> is placed at the top of the stack.
_	Exchange <i>stack1</i> and <i>stack2</i> values.
P or \	Push <i>stack1</i> back onto the stack (that is, duplicate <i>stack1</i> ).
^	Pop the <i>stack1</i> value off the stack.
(conv)	Standard conversion operator converts data in <i>stack1</i> , putting the result into <i>stack1</i> .

### A logical operator

Logical operators compare *stack1* to *stack2*. Each returns 1 for true and 0 for false:

=	Equal to.
<	Less than.
>	Greater than.
# or <>	Not equal to.
[	Less than or equal to.
]	Greater than or equal to.

## F code: Mathematical Functions

---

<b>&amp;</b>	Logical AND.
<b>!</b>	Logical OR.
<b>\n\</b>	Defines a label by a positive integer enclosed by two backslashes (\).
<b>#n</b>	Connection to label <i>n</i> if <i>stack1</i> differs from <i>stack2</i> .
<b>&gt;n</b>	Connection to label <i>n</i> if <i>stack1</i> is greater than <i>stack2</i> .
<b>&lt;n</b>	Connection to label <i>n</i> if <i>stack1</i> is less than <i>stack2</i> .
<b>=n</b>	Connection to label <i>n</i> if <i>stack1</i> equals <i>stack2</i> .
<b>}n</b>	Connection to label <i>n</i> if <i>stack1</i> is greater than or equal to <i>stack2</i> .
<b>{n</b>	Connection to label <i>n</i> if <i>stack1</i> is less than or equal to <i>stack2</i> .
<b>IN</b>	Tests <i>stack1</i> to see if it is the null value.
<b>Fnnnn</b>	If <i>stack1</i> evaluates to false, branch forward <i>nnnn</i> characters in the F code, and continue processing.
<b>Bnnnn</b>	Branch forward unconditionally <i>nnnn</i> characters in the F code, and continue processing.
<b>Gnnnn</b>	Go to label <i>nnnn</i> . The label must be a string delimited by backslashes (\).
<b>G*</b>	Go to the label defined in <i>stack1</i> . The label must be a string delimited by backslashes (\).

**Note:** The F code performs only integer arithmetic.

## G code: Group Extraction

---

### Format

G [ *skip* ] *delim* *#fields*

The G code extracts one or more values, separated by the specified delimiter, from a field.

*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

---

### Format

$L [ n [ ,m ] ]$

The L code places length constraints on the data to be returned.

If  $Ln$  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  $Ln,m$  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

---

### Formats

MCA  
MC/A  
MCD[X]  
MCL  
MCM  
MC/M  
MCN  
MC/N  
MCP  
MCT  
MCU  
MCW  
MCX[D]

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.

MCA	Extracts all alphabetic characters in the field, both upper- and lowercase. Nonalphabetic characters are not printed. In NLS mode, uses the ALPHABETICS field in the NLS.LC.CTYPE file.
MC/A	Extracts all nonalphabetic 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 nonalphabetic characters. In NLS mode, uses the UPPER-CASE 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 <a href="#">STATUS</a> of 2, that is, an invalid conversion code.

## MC codes: Masked Character Conversion

---

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 LOWER-CASE and UPCASED fields of the NLS.LC.CTYPE file. <sup>1</sup>
MCU	Converts all lowercase letters to uppercase. Does not affect uppercase letters or nonalphabetic characters. In NLS mode, uses the LOWER-CASE 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.

1. 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

---

### Format

MD [*n* [*m*]] [,] [\$] [F] [I] [Y] [*intl*] [- | < | C | D] [P] [Z] [T] [*fx*]

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** or an **OCNV** 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.

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.

- , 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.)



## MD code: Masked Decimal Conversion

---

- 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.
- 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 bold 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.

## MD code: Masked Decimal Conversion

---

- 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.

## ML and MR codes: Formatting Numbers

---

### Format

ML  $[n [m]] [Z] [,] [C | D | M | E | N] [\$] [F] [intl] [(fx)]$

MR  $[n [m]] [Z] [,] [C | D | M | E | N] [\$] [F] [intl] [(fx)]$

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.

**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.

- 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.

## ML and MR codes: Formatting Numbers

---

They are unaffected by the Numeric or Monetary categories. If no options are set, the value is returned unchanged.

- 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.
- 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 bold 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.

## ML and MR codes: Formatting Numbers

---

<i>decimal</i>	Character string to use as a decimal delimiter. If <i>decimal</i> contains spaces, commas, or right square brackets, enclose it in quotation marks.
<i>suffix</i>	Character string to append to the number. If <i>suffix</i> contains spaces, commas, or right square brackets, enclose it in quotation marks.
<i>f</i>	One of three format codes: <ul style="list-style-type: none"><li># Data justifies in a field of x blanks.</li><li>* Data justifies in a field of x asterisks ( * ).</li><li>% Data justifies in a field of x zeros.</li></ul>

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.

## MM code: Monetary Conversion

---

### Format

MM [*n*] [I [L]]

The MM code provides for local conventions for monetary formatting.

**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:

- 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.

## MM code: Monetary Conversion

---

The STATUS values are as follows:

- 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.

## MP code: Packed Decimal Conversion

---

### Format

#### MP

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.

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.



### Format

MT [H] [P] [Z] [S] [c] [[f1, f2, f3]]

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.

MT is required when you specify time in either the [ICONV](#) 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.

- |   |  |
|---|--|
| 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.   |

## MT code: Time Conversion

---

- 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

---

### Formats

MX[0C]	Hexadecimal conversion (base 16)
MO[0C]	Octal conversion (base 8)
MB[0C]	Binary conversion (base 2)
MU0C	Hexadecimal Unicode character conversion

The MX, MO, and MB codes convert data from hexadecimal, octal, and binary format to decimal (base 10) format and vice versa.

**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:

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.

## **MX, MO, MB, and MU0C codes: Radix Conversion**

---

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

---

### Format

MY

The MY code specifies conversion from hexadecimal to ASCII on output, and ASCII to hexadecimal on input. 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

---

### Format

NL

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.

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:

- 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

---

### Format

*NLSmapname*

The *NLSmapname* 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.

The STATUS function returns one of the following:

- 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

---

### Format

NR

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 **CONV** function, the NR code converts Arabic numerals into Roman numerals.

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



### Format

`P(pattern) [ { ; | / } (pattern) ] ...`

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.

*pattern* can contain one or more of the following codes:

<i>nN</i>	An integer followed by the letter N, which tests for <i>n</i> numeric characters.
<i>nA</i>	An integer followed by the letter A, which tests for <i>n</i> alphabetic characters.
<i>nX</i>	An integer followed by the letter X, which tests for <i>n</i> alphanumeric characters.
<i>nnnn</i>	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

---

### Format

QR [ *n* { E | . } *m* ] [ *edit* ] [ *mask* ]

QL [ *n* { E | . } *m* ] [ *edit* ] [ *mask* ]

QX

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.

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:

- \$ 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.

## Q code: Exponential Notation

---

*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:

- #*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 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 the [ML and MR codes](#) for more information.

See the [FMT](#) function for more information about formatting numbers.

## R code: Range Function

---

### Format

$R_{n,m} [ \{ ; | / \} n, m ] \dots$

The R code limits returned data to that which falls within specified ranges.  $n$  is the lower bound,  $m$  is the upper bound.

Separate multiple ranges by a semicolon ( ; ) or a slash ( / ).

If range specifications are not met, an empty string is returned.

### Format

S

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.

## S (substitution) code

---

### Format

*S ; nonzero.substitute ; zero.substitute ; null.substitute*

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.

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

### Format

T [ *start*, ] *length*

The T code extracts a contiguous string of characters from a field.

*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:

$$\text{string.length} - \text{substring.length} + 1$$

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

---

### Format

T[**DICT**] *filename* ; *c* [*vloc*] ; [*iloc*] ; [*oloc*] [ ;*bloc*]

T[**DICT**] *filename* ; *c* ; [*iloc*] ; [*oloc*] [ ;*bloc*] [ ,*vloc* | [*vloc*] ]

The *Tfile* 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.

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.

<b>DICT</b>	Specifies the lookup file's dictionary. (In REALITY flavor accounts, you can use an asterisk ( * ) to specify the dictionary: for instance, T* <i>filename</i> ... )
<i>filename</i>	Name of the lookup file.
<i>c</i>	Translation subcode, which must be one of the following: <ul style="list-style-type: none"><li><b>V</b> Conversion item must exist on file, and the specified field must have a value, otherwise an error message is returned.</li><li><b>C</b> If conversion is impossible, return the original value-to-be-translated.</li><li><b>I</b> Input verify only. Functions like V for input and like C for output.</li><li><b>N</b> Returns the original value-to-be-translated if the null value is found.</li><li><b>O</b> Output verify only. Functions like C for input and like V for output.</li><li><b>X</b> If conversion is impossible, return an empty string.</li></ul>
<i>vloc</i>	Number of the value to be returned from a multivalued field. If you do not specify <i>vloc</i> and the field is multivalued, the whole field is returned with all system delimiters turned into blanks. If the <i>vloc</i> specification follows the <i>oloc</i> or <i>bloc</i> specification, enclose <i>vloc</i> in square brackets or separate <i>vloc</i> from <i>oloc</i> or <i>bloc</i> with a comma.



## Tfile code: File Translation

---

<i>iloc</i>	Field number (decimal) for <i>input</i> 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 <i>iloc</i> . If the <i>iloc</i> is omitted, no input translation takes place.
<i>oloc</i>	Field number (decimal) for <i>output</i> translation. When Retrieve creates a listing, data from the field specified by <i>oloc</i> in the lookup file are listed instead of the original value.
<i>bloc</i>	Field number (decimal) which is used instead of <i>oloc</i> during the listing of BREAK.ON and TOTAL lines.

## TI code: International Time Conversion

---

### Format

TI

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](#).

# D

## BASIC Reserved Words

This appendix lists reserved words in the BASIC language. We recommend that you not use them as variable names in your programs.

ABORT	CASE	DEBUG	EXP
ABORTE	CAT	DECLARE	EXTRACT
ABORTM	CATS	DEFFUN	FADD
ABS	CHAIN	DEL	FDIV
ABSS	CHANGE	DELETE	FFIX
ACOS	CHAR	DELETELIST	FFLT
ADDS	CHARS	DELETEU	FIELD
ALL	CHECKSUM	DIAGNOSTICS	FIELDS
ALPHA	CLEAR	DIM	FIELDSTORE
AND	CLEARCOMMON	DIMENSION	FILEINFO
ANDS	CLEARDATA	DISPLAY	FILELOCK
ARG.	CLEARFILE	DIV	FILEUNLOCK
ASCII	CLEARINPUT	DIVS	FIND
ASIN	CLEARPROMPTS	DO	FINDSTR
ASSIGN	CLEARSELECT	DOWNCASE	FIX
ASSIGNED	CLOSE	DQUOTE	FLUSH
ATAN	CLOSESEQ	DTX	FMT
AUTHORIZATION	COL1	EBCDIC	FMTS
BCONVERT	COL2	ECHO	FMUL
BEFORE	COM	ELSE	FOLD
BEGIN	COMMIT	END	FOOTING
BITAND	COMMON	ENTER	FOR
BITNOT	COMPARE	EOF	FORMLIST
BITOR	CONTINUE	EQ	FROM
BITRESET	CONVERT	EQS	FSUB
BITSET	COS	EQU	FUNCTION
BITTEST	COSH	EQUATE	GARBAGECOLLECT
BITXOR	COUNT	EREPLACE	GCI
BREAK	COUNTS	ERRMSG	GE
BSCAN	CREATE	ERROR	GES
BY	CRT	EXCHANGE	GET
CALL	DATA	EXEC	GETLIST
CALLING	DATE	EXECUTE	GETREM
CAPTURING	DCOUNT	EXIT	

GETX	LIT	OPENSEQ	REUSE
GO	LITERALLY	OR	REVREMOVE
GOSUB	LN	ORS	REWIND
GOTO	LOCATE	OUT	RIGHT
GROUP	LOCK	PAGE	RND
GROUPSTORE	LOCKED	PASSLIST	ROLLBACK
GT	LOOP	PCDRIVER	RPC.CALL
GTS	LOWER	PERFORM	RPC.CONNECT
HEADING	LPTR	PRECISION	RPC.DISCONNECT
HEADINGE	LT	PRINT	RQM
HEADINGN	LTS	PRINTER	RTNLIST
HUSH	MAT	PRINTERIO	SADD
ICHECK	MATBUILD	PRINTERR	SCMP
ICONV	MATCH	PROCREAD	SDIV
ICONVS	MATCHES	PROCWRITE	SEEK
IF	MATCHFIELD	PROG	SELECT
IFS	MATPARSE	PROGRAM	SELECTE
ILPROMPT	MATREAD	PROMPT	SELECTINDEX
IN	MATREADL	PWR	SELECTN
INCLUDE	MATREADU	QUOTE	SELECTV
INDEX	MATWRITE	RAISE	SEND
INDEXS	MATWRITEU	RANDOMIZE	SENTENCE
INDICES	MAXIMUM	READ	SEQ
INMAT	MESSAGE	READ.COMMITTED	SEQS
INPUT	MINIMUM	READ.UNCOM-	SEQSUM
INPUTCLEAR	MOD	MITTED	SERIALIZABLE
INPUTDISP	MODS	READBLK	SET
INPUTERR	MTU	READL	SETREM
INPUTIF	MULS	READLIST	SETTING
INPUTNULL	NAP	READNEXT	SIN
INPUTTRAP	NE	READSEQ	SINH
INS	NEG	READT	SLEEP
INSERT	NEGS	READU	SMUL
INT	NES	READV	SOUNDEX
ISNULL	NEXT	READVL	SPACE
ISNULLS	NOBUF	READVU	SPACES
ISOLATION	NO.ISOLATION	REAL	SPLICE
ITYPE	NOT	RECIO	SQLALLOCONNECT
KEY	NOTS	RECORDLOCKED	SQLALLOCENV
KEYEDIT	NULL	RECORDLOCKL	SQLALLOCSTMT
KEYEXIT	NUM	RECORDLOCKU	SQLBINDCOL
KEYIN	NUMS	RELEASE	SQLCANCEL
KEYTRAP	OCONV	REM	SQLCOLATTRI-
LE	OCONVS	REMOVE	BUTES
LEFT	OFF	REPEAT	SQLCONNECT
LEN	ON	REPEATABLE.READ	SQLDESCRIBECOL
LENS	OPEN	REPLACE	SQLDISCONNECT
LES	OPENCHECK	RESET	SQLERROR
LET	OPENDEV	RETURN	SQLEXECDIRECT
LEVEL	OPENPATH	RETURNING	SQLEXECUTE

SQLFETCH	TRANSACTION
SQLFREECONNECT	TRIM
SQLFREEENV	TRIMB
SQLFREESTMT	TRIMBS
SQLGETCURSOR-	TRIMF
NAME	TRIMFS
SQLNUMRESULT-	TRIMS
COLS	TTYCTL
SQLPREPARE	TTYGET
SQLROWCOUNT	TTYSET
SQLSETCONNECT-	UNASSIGNED
OPTION	UNIT
SQLSETCURSOR-	UNLOCK
NAME	UNTIL
SQLSETPARAM	UPCASE
SQRT	USING
SQUOTE	WEOF
SSELECT	WEOFSEQ
SSELECTN	WEOFSEQF
SSELECTV	WHILE
SSUB	WORDSIZE
START	WORKWRITE
STATUS	WRITEBLK
STEP	WRITELIST
STOP	WRITESEQ
STOPE	WRITESEQF
STOPM	WRITET
STORAGE	WRITEU
STR	WRITEV
STRS	WRITEVU
SUB	XLATE
SUBR	XTD
SUBROUTINE	
SUBS	
SUBSTRINGS	
SUM	
SUMMATION	
SYSTEM	
TABSTOP	
TAN	
TANH	
TERMINFO	
THEN	
TIME	
TIMEDATE	
TIMEOUT	
TO	
TPARM	
TPRINT	
TRANS	



# E

## @Variables

Table E-1 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.

**Table E-1. BASIC @Variables**

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 <b>ABORT</b> statement was executed. 2 – An abort was requested after pressing the <b>Break</b> key followed by option A. 3 – An internal or fatal error 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.

**Table E-1. BASIC @Variables (Continued)**

<b>Variable</b>	<b>Read-Only</b>	<b>Value</b>
@CRTHIGH	*	Number of lines on the terminal.
@CRTWIDE	*	Number of columns on the terminal.
@DATA.PENDING	*	Dynamic array containing input generated by the <a href="#">DATA</a> 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 filename. Same as @FILENAME.
@FILENAME		Current filename. Same as @FILE.NAME.
@FM	*	Field mark: CHAR(254). Same as @AM.
@FORMAT		For future use.
@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.
@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).



**Table E-1. BASIC @Variables (Continued)**

<b>Variable</b>	<b>Read-Only</b>	<b>Value</b>
@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.
@NEW	*	New contents of the current record. Use in trigger programs. Nonstacked.
@NI		Current item counter (the number of items listed or selected). Used only in I-descriptors. Same as @RECCOUNT.
@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.
@OLD	*	Original contents of the current record. Use in trigger programs. Nonstacked.
@OPTION		Value of field 5 in the VOC for the calling verb.
@PARASENTENCE	*	Last sentence or paragraph that invoked the current process.
@PATH	*	Pathname of the current account.
@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.

**Table E-1. BASIC @Variables (Continued)**

Variable	Read-Only	Value
@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 <b>EXECUTE</b> 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.
@SYSTEM.SET		Status codes returned by system processes. Same as @SYSTEM.RETURN.CODE.
@TERM.TYPE	*	Terminal type. Nonstacked.

**Table E-1. BASIC @Variables (Continued)**

<b>Variable</b>	<b>Read-Only</b>	<b>Value</b>
@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'.
@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).



# F

## BASIC Subroutines

This appendix describes the following subroutines you can call from a UniVerse BASIC program:

!ASYNC (!AMLC)  
!EDIT.INPUT  
!ERRNO  
!FCMP  
!GET.KEY  
!GET.PARTNUM  
!GET.PATHNAME  
!GET.USER.COUNTS  
!GETPU  
!INLINE.PROMPTS  
!INTS  
!MAKE.PATHNAME  
!MATCHES  
!MESSAGE  
!PACK.FNKEYS  
!REPORT.ERROR  
!SET.PTR  
!SETPU  
!TIMDAT  
!USER.TYPE  
!VOC.PATHNAME

In addition, the subroutines listed in Table F-1 have been added to existing functions for PI/open compatibility.

**Table F-1. PI/open Subroutines**

<b>Subroutine</b>	<b>Associated Function</b>
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
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

**Table F-1. PI/open Subroutines (Continued)**

<b>Subroutine</b>	<b>Associated Function</b>
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

---

## Syntax

CALL !ASync (*key*, *line*, *data*, *count*, *carrier*)

## Description

Use the !ASync subroutine (or its synonym !AMLC) to send data to, and receive data from an asynchronous device.

*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. A 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:

<i>key</i>	Action
1	Inputs the number of characters indicated by the value of <i>count</i> .
2	Inputs the number of characters indicated by the value of <i>count</i> or until a linefeed character is encountered.
3	Outputs the number of characters indicated by the value of <i>count</i> .
4	Returns the number of characters in the input buffer to <i>count</i> . On operating systems where the FIONREAD key is not supported, 0 is returned in <i>count</i> . When the value of <i>key</i> is 4, 1 is always returned to <i>carrier</i> .
5	Returns 0 in <i>count</i> if there is insufficient space in the output buffer. On operating systems where the TIOCOUTQ key is not supported, 0 is returned in <i>count</i> . When the value of <i>key</i> is 5, 1 is always returned to <i>carrier</i> .



### 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

---

### Syntax

CALL !EDIT.INPUT (*keys*, *wcol*, *wrow*, *wwidth*, *buffer*, *startpos*, *bwidth*, *ftable*,  
*code*)

### Qualifiers

*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 <i>bwidth</i> 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.

## !EDIT.INPUT subroutine

---

<i>bwidth</i>	The maximum number of positions allowed in <i>buffer</i> . <i>bwidth</i> can be more than <i>wwidth</i> , in which case the contents of <i>buffer</i> scroll horizontally as required.
<i>ftable</i>	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.
<i>code</i>	The reply code: = 0            User pressed <b>Return</b> or entered <i>bwidth</i> characters and IK\$ATM was set. > 0            The function key number that terminated !EDIT.INPUT.

### Description

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](#), [KEYTRAP](#) and [KEYEXIT](#) statements. 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.

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*.

### !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

## !EDIT.INPUT subroutine

---

Value	Token	Description
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 immediately 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, i.e., 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.

## **!EDIT.INPUT subroutine**

---

The complete listing of the fields defined within the GTI.FNKEYS.IH include file is shown below:

<b>Key Name</b>	<b>Field</b>	<b>Description</b>
FK\$FIN	1	Finish
FK\$HELP	2	Help
FK\$BSP	3	Backspace <sup>1</sup>
FK\$LEFT	4	Left arrow <sup>1</sup>
FK\$RIGHT	5	Right arrow <sup>1</sup>
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) <sup>1</sup>
FK\$INSLIN	20	Insert line
FK\$INSTXT	21	Insert text, Toggle insert/overlay mode <sup>1</sup>
FK\$INSDOC	22	Insert document
FK\$DELCH	23	Delete character <sup>1</sup>
FK\$DELLIN	24	Delete line <sup>1</sup>
FK\$DELTXT	25	Delete text
FK\$SRCHNX	26	Search next

## **!EDIT.INPUT subroutine**

---

<b>Key Name</b>	<b>Field</b>	<b>Description</b>
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 line <sup>1</sup>
FK\$SCRWID	52	Toggle between 80 and 132 mode
FK\$PERF	53	Invoke DSS PERFORM emulator
FK\$INCLUDE	54	DSS Include scratchpad data

## !EDIT.INPUT subroutine

---

Key Name	Field	Description
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)
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

1. Indicates supported functionality.

## !EDIT.INPUT subroutine

---

### 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
```



### Syntax

CALL !ERRNO (*variable*)

### Description

Use the !ERRNO subroutine to return the current value of the operating system *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

---

### Syntax

CALL !FCMP (*result*, *number1*, *number2*)

### Description

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 !GET.KEY (*string*, *code*)

### Qualifiers

*string* Returns the character sequence of the next key pressed at the keyboard.

*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, <i>code</i> = 0 and <i>string</i> = 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 labelled --> (right cursor move) returns <i>code</i> = 5 and <i>string</i> = 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 <i>terminfo</i> entry or the CUSTOM.GCI.DEFS file.

### Description

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.

### 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
```

## !GET.KEY subroutine

---

```
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 "  (Unrecognised 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

---

### Syntax

CALL !GET.PARTNUM (*file*, *record.ID*, *partnum*, *status*)

### Description

Use the !GET.PARTNUM subroutine with distributed files to determine the number of the part file to which a given record ID belongs.

*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](#) SYSCOM INFO\_ERRORS.INS.IBAS when you compile the program.

Equate Name	Description
IESNOT.DISTFILE	The file specified by the file variable is not a distributed file.
IESDIST.DICT.OPEN.FAIL	The program failed to open the file dictionary for the distributed file.
IESDIST.ALG.READ.FAIL	The program failed to read the partitioning algorithm from the distributed file dictionary.
IESNO.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.

## !GET.PARTNUM subroutine

---

### 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

---

### Syntax

CALL !GET.PATHNAME (*pathname*, *directoryname*, *filename*, *status*)

### Description

Use the !GET.PATHNAME subroutine to return the directory name and filename parts of a pathname.

*pathname* (input) is the pathname from which the details are required.

*directoryname* (output) is the directory name portion of the pathname, that is, the pathname with the last entry name stripped off.

*filename* (output) is the filename portion of the pathname.

*status* (output) is the returned status of the operation. A 0 indicates success, another number is an error code indicating that the supplied pathname 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

---

### Syntax

CALL !GETPU (*key*, *print.channel*, *set.value*, *return.code*)

### Description

Use the !GETPU subroutine to read individual parameters of any logical print channel.

*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 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.



## !GETPU subroutine

---

Mnemonic	Key	Parameter
PUSDEFERTIME	8	Spool defer time. This cannot be 0.
PUSFORM	9	Spool form (string).
PUSBANNER	10	Spool banner or hold filename (string).
PUSLOCATION	11	Spool location (string).
PUSCOPIES	12	Spool copies. A single copy can be returned as 1 or 0.
PUSPAGING	14	Terminal paging (nonzero is on). This only works when PUSMODE is set to 1.
PUSPAGENUMBER	15	Returns the current page number.
PUSDISABLE	16	0 is returned if <i>print.channel</i> is enabled; and a 1 is returned if <i>print.channel</i> is disabled.
PUSCONNECT	17	Returns the number of a connected print channel or an empty string if no print channels are connected.
PUSNLSMAP	22	If NLS is enabled, returns the NLS map name associated with the specified print channel.
PUSLINESLEFT	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.
PUSHEADERLINES	1003	Lines used by current header.
PUSFOOTERLINES	1004	Lines used by current footer.
PUSDATA LINES	1005	Lines between current header and footer.
PUSDATA COLUMNS	1006	Columns between left margin and device width.

### The PUS\$POOLFLAGS Key

The PUS\$POOLFLAGS 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

## !GETPU subroutine

---

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: <table><tr><th>Character</th><th>Meaning</th></tr><tr><td>0</td><td>Advances two lines.</td></tr><tr><td>1</td><td>Ejects to the top of the next page.</td></tr><tr><td>+</td><td>Overprints the last line.</td></tr><tr><td>Space</td><td>Advances one line.</td></tr><tr><td>-</td><td>Advances three lines (skip two lines). Any other character is interpreted as advance one line.</td></tr></table>	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.
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 <a href="#">!SETPU</a> 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
ESBKEY	Bad key ( <i>key</i> is out of range)

Code	Meaning
ESBPAR	Bad parameter (value of <i>new.value</i> is out of range)
ESBUNT	Bad unit number (value of <i>print.channel</i> is out of range)
ESNRIT	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 ON** statement 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**

---

### **Syntax**

CALL !GET.USER.COUNTS (*uv.users*, *max.uv.users*, *os.users*)

### **Description**

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.

*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.

## !INLINE.PROMPTS subroutine

---

### Syntax

CALL !INLINE.PROMPTS (*result*, *string*)

### Description

Use the !INLINE.PROMPTS subroutine to evaluate a string that contains in-line prompts. 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:

@(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.
@( <i>col</i> , <i>row</i> )	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 <i>n</i>	Specifies that the <i>n</i> th word on the command line is used as the input value. (Word 1 is the verb in the sentence.)
F( <i>filename</i> , <i>record.id</i> [ , <i>fm</i> [ , <i>vm</i> [ , <i>sm</i> ] ] ])	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.
In	Takes the <i>n</i> th word from the command line, but prompts if the word is not entered.
R( <i>string</i> )	Repeats the prompt until an empty string is entered. If <i>string</i> is specified, each response to the prompt is appended by <i>string</i> between each entry. If <i>string</i> is not specified, a space is used to separate the responses.

## !INLINE.PROMPTS subroutine

---

<b>P</b>	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.
<b>Sn</b>	Takes the <i>n</i> th word from the command (as in the <i>In</i> 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.
<b>text</b>	The prompt to be displayed.
<b>option</b>	A valid conversion code or pattern match. A valid conversion code is one that can be used with the <b>ICONV</b> function. Conversion codes must be enclosed in parentheses. A valid pattern match is one that can be used with the <b>MATCHING</b> 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
```

### Syntax

CALL !INTS (*result*, *dynamic.array*)

### Description

Use the !INTS subroutine to retrieve the integer portion of elements in a 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

---

## Syntax

CALL !MAKE.PATHNAME (*path1*, *path2*, *result*, *status*)

## Description

Use the !MAKE.PATHNAME subroutine to construct the full pathname of a file. The !MAKE.PATHNAME subroutine can be used to:

- Concatenate two strings to form a pathname. The second string must be a relative path.
- Obtain the fully qualified pathname of a file. Where only one of *path1* or *path2* is given, !MAKE.PATHNAME returns the pathname in its fully qualified state. In this case, any filename you specify does not have to be an existing filename.
- Return the current working directory. To do this, specify both *path1* and *path2* as empty strings.

*path1* (input) is a filename or partial pathname. If *path1* is an empty string, the current working directory is used.

*path2* (input) is a relative pathname. If *path2* is an empty string, the current working directory is used.

*result* (output) is the resulting pathname.

*status* (output) is the returned status of the operation. 0 indicates success. Any other number indicates either of the following errors:

IES\$NOTRELATIVE	<i>path2</i> was not a relative pathname.
IES\$PATHNOTFOUND	The pathname 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
```



### Syntax

CALL !MATCHES (*result*, *dynamic.array*, *match.pattern*)

### Description

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.

*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)
```

## !MATCHES subroutine

---

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)
```

### Syntax

CALL !MESSAGE (*key*, *username*, *usernum*, *message*, *status*)

### Description

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.

*key* (input) specifies the operation to be performed. You specify the option you require with the *key* argument, as follows:

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:

0	The operation was successful.
IE\$NOSUPPORT	You specified an unsupported <i>key</i> option.
IE\$KEY	You specified an invalid <i>key</i> option.
IE\$PAR	The <i>username</i> or <i>message</i> 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.

## **!MESSAGE subroutine**

---

### **Example**

```
CALL !MESSAGE (KEY,USERNAME,USERNUMBER,MESSAGE,CODE)
IF CODE # 0
THEN CALL !REPORT.ERROR ('MY.COMMAND', '!MESSAGE',CODE)
```

## !PACK.FNKEYS subroutine

---

### Syntax

CALL !PACK.FNKEYS (*trap.list*, *fable*)

### Qualifiers

*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.

*fable*          A bit-significant string of trap keys used in the *fable* parameter of the !EDIT.INPUT subroutine. This string should not be changed in any way before calling the !EDIT.INPUT subroutine.

### Description

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.

*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 <sup>1</sup>
FK\$LEFT	4	Left arrow <sup>1</sup>
FK\$RIGHT	5	Right arrow <sup>1</sup>
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

## !PACK.FNKEYS subroutine

---

Key Name	Field	Description
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) <sup>1</sup>
FK\$INSLIN	20	Insert line
FK\$INSTXT	21	Insert text, Toggle insert/overlay mode <sup>1</sup>
FK\$INSDOC	22	Insert document
FK\$DELCH	23	Delete character <sup>1</sup>
FK\$DELLIN	24	Delete line <sup>1</sup>
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

## !PACK.FNKEYS subroutine

---

Key Name	Field	Description
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 <sup>1</sup>
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

## !PACK.FNKEYS subroutine

---

Key Name	Field	Description
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

1. Indicates supported functionality.

If *f*table is returned as an empty string, an error in the *trap.list* array is detected, such as an invalid function number. Otherwise *f*table is a bit-significant string which should not be changed in any way before its use with the [!EDIT.INPUT](#) 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=""
```



## **!PACK.FNKEYS subroutine**

---

```
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

---

## Syntax

CALL !REPORT.ERROR (*command*, *subroutine*, *code*)

## Description

Use the !REPORT.ERROR subroutine to print explanatory text for a UniVerse or operating system error 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 IESSEND.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)
```

## **!REPORT.ERROR subroutine**

---

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

---

## Syntax

CALL !SET.PTR (*print.channel*, *width*, *length*, *top.margin*, *bottom.margin*,  
*mode*, *options*)

## Description

Use the !SET.PTR subroutine to set options for a logical print channel. This subroutine provides the same functionality as the UniVerse [SETPTR](#) command.

*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

---

### Syntax

CALL !SETPU (*key*, *print.channel*, *new.value*, *return.code*)

### Description

Use the !SETPU subroutine to set individual parameters of any logical print channel.

Unlike [!SET.PTR](#), 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 for a way to read individual *print.channel* parameters.

*key* is a number indicating the parameter to be set (see “[Equate Names for Keys](#)”).

*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 “[Equate Names for Return Code](#)”).

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 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.

### 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](#) compiler directive, see Chapter 3. 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).

Mnemonic	Key	Parameter
PU\$SPOOLFLAGS	7	Spool option flags (see “ <a href="#">The PU\$SPOOLFLAGS Key</a> ”).
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	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: <table><tr><th>Character</th><th>Meaning</th></tr><tr><td>0</td><td>Advances two lines.</td></tr><tr><td>1</td><td>Ejects to the top of the next page.</td></tr><tr><td>+</td><td>Overprints the last line.</td></tr><tr><td>Space</td><td>Advances one line.</td></tr><tr><td>–</td><td>Advances three lines (skip two lines). Any other character is interpreted as advance one line.</td></tr></table>	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.
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.												

## !SETPU subroutine

---

Bit	Description
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
ESBKEY	Bad key ( <i>key</i> is out of range)
ESBPAR	Bad parameter (value of <i>new.value</i> is out of range)
ESBUNT	Bad unit number (value of <i>print.channel</i> is out of range)
ESNRIT	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)
```



## **!SETPU subroutine**

---

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

---

### Syntax

CALL !TIMDAT (*variable*)

### Description

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.

*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).
6	Ticks <sup>1</sup> of last second since midnight (integer). Always returns 0.
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).

1. Tick refers to the unit of time your system uses to measure real time.

## !TIMDAT subroutine

---

Use the following functions for alternative ways of obtaining time and date information:

Use this function...	To obtain this data...
----------------------	------------------------

<code>DATE ()</code>	Data in fields 1, 2, and 3 of the dynamic array returned by the !TIMDAT subroutine
----------------------	--

<code>TIME ()</code>	Data in fields 4, 5, and 6 of the dynamic array returned by the !TIMDAT subroutine
----------------------	--

<code>@USERNO</code>	User number
----------------------	-------------

<code>@LOGNAME</code>	Login ID (user ID)
-----------------------	--------------------

### Example

```
CALL !TIMDAT(DYNARRAY)
FOR X = 1 TO 13
    PRINT 'ELEMENT ' : X : ', DYNARRAY
NEXT X
```

## !USER.TYPE subroutine

---

### Syntax

CALL !USER.TYPE (*type*, *admin*)

### Description

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](#).

*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
USNORM	1	Normal user
USPH	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](#) 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

---

### Syntax

CALL !VOC.PATHNAME (*data/dict*, *voc.entry*, *result*, *status*)

### Description

Use the !VOC.PATHNAME subroutine to extract the pathnames for the data file or the file dictionary of a specified VOC entry.

*data/dict* (input) indicates the file dictionary or data file, as follows:

IK\$DICT or 'DICT' returns the pathname of the file dictionary of the specified VOC entry.

IK\$DATA or ' ' returns the pathname (or pathnames 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 pathnames.

*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 `$INCLUDE SYSCOM INFO_KEYS.H` when you compile your program. (For PI/open compatibility you can specify `$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.
IESPAR	A bad parameter was used in <i>data/dict</i> or <i>voc.entry</i> .
IESRNF	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
```



# Index

## Symbols

– operator 2-11  
! statement 1-4, 6-2  
# operator 2-15  
&PH& file 3-1  
\* operator 2-11  
\* statement 1-4, 6-27  
\*\* operator 2-11  
+ operator 2-11  
+= operator 2-19  
/ operator 2-11  
: operator 2-13  
:= operator 2-19  
< > operator 2-15, 6-28, 6-175, 6-453  
< operator 2-15  
<= operator 2-15  
-= operator 2-19  
= operator 2-15, 2-19  
> operator 2-15  
>< operator 2-15  
>= operator 2-15  
@ expressions in INPUT  
    statements 6-273  
@ function 6-29  
@variables 4-10, E-1–E-5  
[ ] operator 6-49  
^ operator 2-11  
~ (tilde) 6-338

## Subroutines

!ASYNC subroutine F-4  
!EDIT.INPUT subroutine F-6  
!ERRNO subroutine F-13  
!FCMP subroutine F-14

!GET.KEY subroutine F-15  
!GET.PARTNUM subroutine F-17  
!GET.PATHNAME subroutine F-19  
!GET.USER.COUNTS subroutine F-24  
!GETPU subroutine F-20  
!INLINE.PROMPTS subroutine F-25  
!INTS subroutine F-27  
!MAKE.PATHNAME subroutine F-28  
!MATCHES subroutine F-29  
!MESSAGE subroutine F-31  
!PACK.FNKEYS subroutine F-33  
!REPORT.ERROR subroutine F-38  
!SET.PTR subroutine F-40  
!SETPU subroutine F-42  
!TIMDAT subroutine F-46  
!USER.TYPE subroutine F-48  
!VOC.PATHNAME subroutine F-49

## Compiler Directives

#INCLUDE statement 3-5, 6-3  
\$\* statement 6-4  
\$CHAIN statement 3-5, 6-5  
\$COPYRIGHT statement 6-6  
\$DEFINE statement 3-6, 6-7  
\$EJECT statement 6-9  
\$IFDEF statement 3-8, 6-10  
\$IFNDEF statement 3-8, 6-11  
\$INCLUDE statement 3-5, 6-12  
\$INSERT statement 3-3, 3-5, 6-13  
\$MAP statement 6-15  
\$OPTIONS statement 3-6, 6-16  
    default settings 6-22  
    options 6-17–6-21  
    STATIC.DIM option 2-6  
    VEC.MATH option 2-13, 2-20

\$PAGE statement 6-25  
\$UNDEFINE statement 3-6, 6-26

## @Variables

@ABORT.CODE variable E-1  
@ACCOUNT variable E-1  
@AM variable E-1  
@ANS variable E-1  
@AUTHORIZATION variable 6-65,  
E-1  
@COMMAND variable E-1  
@COMMAND.STACK variable E-1  
@CONV variable E-1  
@CRTHIGH variable E-2  
@CRTWIDE variable E-2  
@DATA.PENDING variable E-2  
@DATE variable E-2  
@DAY variable E-2  
@DICT variable E-2  
@FILE.NAME variable E-2  
@FILENAME variable E-2  
@FM variable E-2  
@FORMAT variable E-2  
@HDBC variable E-2  
@HEADER variable E-2  
@HENV variable E-2  
@HSTMT variable E-2  
@ID variable E-2  
@IM variable E-2  
@ISOLATION variable 4-10, E-2  
@LEVEL variable E-2  
@LOGNAME variable E-2  
@LPTRHIGH variable E-2  
@LPTRWIDE variable E-2  
@MONTH variable E-3  
@MV variable E-3  
@NB variable E-3  
@ND variable E-3  
@NEW variable E-3  
@NI variable E-3

@NS variable E-3  
@NULL variable 2-3, 2-13, 6-62, E-3  
@NULL.STR variable 2-4, 2-13, 6-62,  
6-95, 6-96, E-3  
@NV variable E-3  
@OLD variable E-3  
@OPTION variable E-3  
@PARASENTECE variable E-3  
@PATH variable E-3  
@RECCOUNT variable E-3  
@RECORD variable E-3  
and ITYPE function 6-294  
@RECUR0 variable E-3  
@RECUR1 variable E-4  
@RECUR2 variable E-4  
@RECUR3 variable E-4  
@RECUR4 variable E-4  
@SCHEMA variable E-4  
@SELECTED variable E-4  
@SENTENCE variable 6-489, E-4  
@SM variable E-4  
@SQL.CODE variable E-4  
@SQL.DATE variable E-4  
@SQL.ERROR variable E-4  
@SQL.STATE variable E-4  
@SQL.TIME variable E-4  
@SQL.WARNING variable E-4  
@SQLPROC.NAME variable E-4  
@SQLPROC.TX.LEVEL variable E-4  
@STDFIL variable 6-380, 6-389, E-4  
@SVM variable E-4  
@SYS.BELL variable E-4  
@SYSTEM.RETURN.CODE  
variable E-4  
@SYSTEM.SET variable E-4  
@TERM.TYPE variable E-4  
@TIME variable E-5  
@TM variable E-5  
@TRANSACTION variable 4-10, E-5  
@TRANSACTION.ID variable 4-10,  
E-5



- @TRANSACTION.LEVEL
  - variable [4-10](#), [E-5](#)
- @TRUE variable [E-5](#)
- @TTY variable [E-5](#)
- @USER.NO variable [E-5](#)
- @USER.RETURN.CODE variable [E-5](#)
- @USER0 variable [E-5](#)
- @USER1 variable [E-5](#)
- @USER2 variable [E-5](#)
- @USER3 variable [E-5](#)
- @USER4 variable [E-5](#)
- @USERNO variable [E-5](#)
- @VM variable [E-5](#)
- @WHO variable [E-5](#)
- @YEAR variable [E-5](#)
- @YEAR4 variable [E-5](#)

## A

- A conversion [C-4](#)
- ABORT statement [6-52](#)
- ABORTE statement [6-21](#), [6-52](#)
- ABORTM statement [6-21](#), [6-52](#)
- ABS function [6-53](#)
- ABSS function [6-54](#)
- ACID properties [4-7](#)
- ACOS function [6-55](#)
- ADDS function [6-56](#)
- algebraic functions [C-4](#)
- ALPHA function [6-57](#)
- alphabetic characters [1-6](#)
- AND operator [2-17](#), [6-58](#), [6-70](#)
- ANDS function [6-58](#)
- angle brackets (< >) [6-28](#), [6-175](#), [6-453](#)
- Arabic numeral conversion [C-40](#)
- arguments, passing to subroutines [1-3](#)
- arithmetic operators [2-11-2-13](#)
  - and dynamic arrays [2-20](#)
  - and multivalued data [2-13](#)
  - and the null value [2-12](#)
- array variables [2-5-2-9](#)
- arrays
  - assigning values to [6-335](#)

- dimensioned [2-5-2-6](#), [6-146](#)
- dynamic [2-7-2-9](#)
- matrices [2-6](#)
- passing to subroutines [6-87](#), [6-147](#), [6-527](#)
- standard [2-6](#)
- vectors [2-6](#)

## ASCII

- characters [2-2](#)
  - CHAR(0) [2-2](#)
  - CHAR(10) [2-2](#)
  - CHAR(128) [2-3](#)
- codes [B-1](#)
- conversion [C-39](#)
- function [6-59](#)
- strings [1-2](#), [2-15](#)

## ASIN function [6-60](#)

## ASSIGNED function [6-61](#)

- assigning variables [6-62](#), [6-313](#)

- assignment operators [2-19](#), [6-28](#), [6-62](#)
  - and substrings [2-14](#)

- assignment statements [1-4](#), [6-62](#), [6-313](#)

## ATAN function [6-64](#)

- atomicity property [4-7](#)

- AUTHORIZATION statement [6-65](#)

- AUXMAP statement [6-67](#)

## B

- B-tree files [6-79](#)

- BASIC character set

- alphabetic [1-6](#)

- numeric [1-6](#)

- special [1-6](#)

- BASIC command [3-1-3-4](#)

- options [3-2-3-4](#)

- BASIC compiler

- keywords [1-2](#)

- variables [1-2](#)

- BASIC language

- application [1-1](#)

- extensions of [1-1](#)

- reserved words [D-1](#)

- BASIC programs
  - cataloging 3-10–3-13
  - compiling 3-1–3-9
  - conditional compiling 3-6, 6-258
  - definition 1-2
  - editing 1-7
  - flavors compatibility 3-6
  - global cataloging 3-11
  - listing 3-2, 3-3
  - local cataloging 3-11
  - normal cataloging 3-11
  - object code 3-9
    - listing 5-12
  - printing 3-2, 3-4
  - running 3-10
  - storing 1-7
- BASIC statements 1-2
- BB conversion C-8
- BEGIN CASE statement 6-89
- BEGIN TRANSACTION
  - statement 6-69
- binary conversion C-37
- bit conversion C-8
- BITAND function 6-70
- BITNOT function 6-71
- BITOR function 6-72
- BITRESET function 6-73
- BITSET function 6-74
- BITTEST function 6-75
- BITXOR function 6-76
- blank spaces, *see* spaces
- brackets
  - angle (< >) 6-28, 6-175, 6-453
  - square (| |) 6-49
- Break key 5-3
- BREAK statement 6-77
- BSCAN statement 6-79
- BX conversion C-8
- BYTE function 6-82
- BYTELEN function 6-83
- BYTETYPE function 6-84
- BYTEVAL function 6-85

## C

- C conversion C-9
- CALL statement 1-3, 6-86
  - and RETURN statement 6-457
  - and SUBROUTINE
    - statement 6-527
- calling subroutines 6-86
- CASE
  - option 6-17
  - statement 6-89
- CAT operator 2-13
- CATALOG command 3-12
- catalog shared memory 3-13
- cataloging BASIC programs
  - globally 3-11
  - locally 3-11
  - normally 3-11
- CATS function 6-92
- CHAIN statement 6-93
- CHANGE function 6-94
- CHAR function 6-95
- CHAR(0) 2-2
- CHAR(10) 2-2
- CHAR(128) 6-95, 6-96, 6-181
- CHAR(252) 2-7
- CHAR(253) 2-7
- CHAR(254) 2-7
- character conversion C-24
- character set conversion C-18
- character strings 2-1–2-2
  - ASCII 2-2
  - constants 2-2
  - empty 2-3, 2-4
  - numeric 2-12
  - substrings 2-13
- characters
  - alphabetic 1-6
  - numeric 1-6
  - special 1-6
- CHARS function 6-96
- CHECKSUM function 6-97

- CLEAR statement [6-98](#)
- CLEARDATA statement [6-99](#)
- CLEARFILE statement [6-100](#)
- clearing
  - in-line prompts [6-102](#)
  - select lists [6-103](#)
- CLEARPROMPTS statement [6-102](#),  
[6-262](#)
- CLEARSELECT statement [6-103](#)
- CLOSE statement [6-105](#)
- CLOSESEQ statement [6-107](#)
- COL1 function [6-109](#)
- COL2 function [6-110](#)
- commas in numeric constants [2-3](#)
- comments [1-4](#), [6-2](#), [6-4](#), [6-27](#), [6-447](#)
- COMMIT statement [6-112](#)
- common
  - clearing [6-98](#)
  - unnamed, saving variable  
values [6-93](#)
- COMMON statement [6-114](#)
- common variables [2-5](#)
- COMPPRECISION option [6-17](#)
- COMPARE function [6-115](#)
- compiler directives [3-4](#)–[3-9](#)
- concatenation
  - conversion [C-9](#)
  - and the null value [2-13](#)
  - operator [2-13](#)
- conditional compiling [3-6](#), [6-258](#)
- configurable parameters
  - ISOMODE [4-13](#)
  - MAXRLOCK [4-15](#)
  - OPENCHK [6-251](#), [6-352](#), [6-381](#),  
[6-605](#)
- consistency property [4-7](#)
- constants
  - character string [2-2](#)
  - definition [2-4](#)
  - fixed-point numeric [2-3](#)
  - floating-point [2-3](#)
  - numeric [2-3](#)
- CONTINUE statement [6-217](#), [6-330](#)
- control keys [6-297](#), [6-302](#), [6-305](#)
  - defining [6-297](#)
- control statements, *see* statements
- conventions, documentation [xvi](#)
- conversion codes [C-1](#)–[C-51](#)
  - A [C-4](#)
  - BB [C-8](#)
  - BX [C-8](#)
  - C [C-9](#)
  - D [C-11](#)
  - DI [C-17](#)
  - ECS [C-18](#)
  - F [C-19](#)
  - G [C-22](#)
  - in format expressions [6-204](#)
  - in ICONV function [6-254](#)
  - in ICONVS function [6-256](#)
  - in OCONV function [6-373](#)
  - in OCONVS function [6-376](#)
  - L [C-23](#)
  - MB [C-37](#)
  - MC [C-24](#)
  - MD [C-26](#)
  - ML [C-29](#)
  - MM [C-32](#)
  - MO [C-37](#)
  - MP [C-34](#)
  - MR [C-29](#)
  - MT [C-35](#)
  - MU0C [C-37](#)
  - MX [C-37](#)
  - MY [C-39](#)
  - NL [C-40](#)
  - NLSmapname [C-41](#)
  - NR [C-42](#)
  - P [C-43](#)
  - Q [C-44](#)
  - R [C-46](#)
  - S (Soundex) [C-47](#)
  - S (substitution) [C-48](#)
  - T [C-49](#)

- Tfile* C-50
  - TI C-52
- CONVERT
  - function 6-118
  - statement 6-119
- COS function 6-120
- COSH function 6-121
- COUNT function 6-122
- COUNT.OVLP option 6-17, 6-122, 6-124, 6-131, 6-265
- COUNTS function 6-124
- CREATE statement 6-126
  - and NOBUF statement 6-367
- CRT statement 6-127
- cursors, positioning 6-29

## D

- D conversion C-11
- data
  - anomalies 4-12
  - character string 2-1-2-2
  - null value 2-3
  - numeric 2-2
  - preventing loss of 4-1
  - visibility 4-6
- DATA statement 6-128
  - and INPUT statements 6-273
- data types 2-1-2-4
  - logical 2-17
  - null value 2-3
- date conversion C-11, C-17
- date format, default C-11
- DATE function 6-130
- dates, internal system 6-130
- DCOUNT function 6-131
- deadlocks 4-5
- DEBUG statement 5-3, 6-132
- debugger 5-1, 6-132
- DECATALOG command 3-13
- decimal equivalents B-5
- DEFFUN statement 6-134, 6-222

- defining
  - control keys 6-297
  - escape keys 6-298
  - function keys 6-298
  - identifiers 3-6
  - unsupported keys 6-300
- DEL statement 6-136
- DELETE
  - function 6-138
  - statement 6-140
- DELETE.CATALOG command 3-12
- DELETEDLIST statement 6-144
- DELETEDU statement 6-141, 6-145
- delimiters, system 2-7
- DI conversion C-17
- DIM statement 6-146
- DIMENSION statement 6-146
- dimensioned arrays 2-5-2-6
- dirty reads 4-12
- display length 2-10
- DISPLAY statement 6-149
- distributed files, status 6-517
- DIV function 6-150
- DIVS function 6-151
- documentation conventions xvi
- DOWNCASE function 6-152
- DQUOTE function 6-153
- DTX function 6-154
- durability property 4-7
- dynamic arrays 2-7-2-9
  - and arithmetic operators 2-20
  - and the null value 2-25
  - and operators 2-19-2-25
  - and REUSE function 2-24
  - creating 2-8
- dynamic arrays, passing to subroutines 6-87

## E

- EBCDIC function 6-155
- ECHO statement 6-156

- ECS conversion [C-18](#)
- edit* editor [1-7](#)
- editors
  - edit* [1-7](#)
  - vi* [1-7](#)
- effective UID [6-65](#)
  - AUTHORIZATION statement [6-65](#)
- empty strings [2-3](#), [2-4](#)
  - and pattern matching [2-17](#)
- END CASE statement [6-89](#)
- END statement [6-157](#)
- END.WARN option [6-17](#), [6-157](#)
- ENTER statement [6-160](#)
- entering external subroutines [6-160](#)
- EOF(ARG.) function [6-161](#)
- EQ operator [2-15](#)
- EQS function [6-162](#)
- EQUATE statement [6-163](#)
- EREPLACE function [6-165](#)
- ERRMSG
  - codes [6-166](#)
  - file [6-166](#), [6-406](#)
    - and STOPE statement [6-520](#)
  - statement [6-166](#)
    - and STOP statement [6-520](#)
- error messages [3-9](#), [6-52](#), [6-166](#), [6-406](#), [6-520](#)
- escape keys [6-298](#), [6-302](#), [6-305](#)
  - defining [6-298](#)
- EXCHANGE function [6-168](#)
- exclusive file locks [4-5](#)
- EXEC.EQ.PERF option [6-17](#), [6-171](#)
- EXECUTE statement [6-170](#)
- EXIT statement [6-173](#)
- EXP function [6-174](#)
- expressions [2-9](#)
  - @ [6-273](#)
    - format [2-10](#), [6-201](#)–[6-204](#)
- extended character set
  - conversion [C-18](#)
- external subroutines, entering [6-160](#)

- EXTRA.DELIM option [6-17](#), [6-286](#), [6-289](#), [6-454](#)
- EXTRACT function [6-28](#), [6-175](#)
- extracting substrings [6-49](#)

## F

- F conversion [C-19](#)
- FADD function [6-177](#)
- FDIV function [6-178](#)
- FFIX function [6-179](#)
- FFLT function [6-180](#)
- FIELD function [6-181](#)
  - and COL1 function [6-109](#)
  - and COL2 function [6-110](#)
- field marks [2-7](#)
- FIELDS function [6-183](#)
- file locks, types [4-2](#), [6-192](#)
- file translation [C-50](#)
- file variables [2-9](#)
- FILELOCK statement [6-192](#)
- files
  - &PH& [3-1](#)
  - B-tree [6-79](#)
  - closing [6-105](#)
  - configuration information [6-186](#)
  - distributed [6-517](#)
  - ERRMSG [6-166](#), [6-406](#)
  - locking [6-195](#)
  - part [6-517](#)
  - sequential processing [6-107](#), [6-126](#), [6-200](#), [6-367](#), [6-386](#), [6-392](#), [6-423](#), [6-430](#), [6-478](#), [6-562](#), [6-613](#), [6-616](#)
  - type 1 [1-7](#), [3-1](#)
  - type 19 [1-7](#), [3-1](#)
- FILEUNLOCK statement [6-195](#)
- FIND statement [6-197](#)
- FINDSTR statement [6-198](#)
- FIX function [6-199](#)
- fixed-point constants [2-3](#)

- flavors 6-16
  - compatibility 3-6
- floating-point constants 2-3
- floating-point numbers 2-2
- FLUSH statement 6-200
  - after WRITESEQ statement 6-613
- FMT function 6-201
- FMTDP function 6-206
- FMTS function 6-207
- FMTSDP function 6-208
- FMUL function 6-209
- FOLD function 6-210
- FOLDDP function 6-211
- FOOTING statement 6-212
- FOR statement 6-216
- FOR.INCR.BEF option 6-17, 6-217
- format expressions 2-10, 6-201–6-204
  - in INPUT statements 6-274
- FORMAT.OCONV option 6-17
- formatting numbers C-29
- FORMLIST statement 6-220
- FSELECT option 6-17
- FSUB function 6-221
- function keys 6-298, 6-302, 6-305, F-15, F-33
  - defining 6-298
- FUNCTION statement 6-222
- functions
  - intrinsic 1-2
  - numeric 1-2
  - range C-46
  - string 1-2
  - user-written 6-459
  - vector 2-20

## G

- G conversion C-22
- GE operator 2-15
- GES function 6-224
- GET statement 6-225
- GET(ARG.) statement 6-231

- GETLIST statement 6-233
- GETLOCALE function 6-234, 6-315
- GETREM function 6-235
- GETX statement 6-225, 6-230
- global cataloging 3-11
- GOSUB statement 1-3, 6-236
  - with ON statement 6-377
  - and RETURN statement 6-457
- GOTO statement 6-238
  - with ON statement 6-378
- granularity 4-1
- group extraction C-22
- GROUP function 6-239
- GROUPSTORE statement 6-241
- GT operator 2-15
- GTS function 6-243

## H

- HEADER.BRK option 6-17
- HEADER.DATE option 6-18, 6-215, 6-247
- HEADER.EJECT option 6-18, 6-245
- HEADING statement 6-244
- HEADINGE statement 6-21, 6-245
- HEADINGN statement 6-21, 6-245
- hexadecimal conversion C-37
- hexadecimal equivalents B-5
- host name and SYSTEM
  - function 6-536
- HUSH statement 6-249

## I

- ICHECK function 6-251
- ICONV function 6-254, C-1, C-8, C-37
- ICONVS function 6-256, C-6
- identifiers
  - removing 3-6
  - replacing 3-6

## IF

- operator [2-17](#)
- statement [6-257](#)
- IFS function [6-260](#)
- ILPROMPT function [6-261](#)
- IN2.SUBSTR option [6-18](#)
- include files, UVLOCALE.H [6-234](#),  
[6-315](#)
- INCLUDE statement [3-5](#), [6-264](#)
- INDEX function [6-265](#)
- INDEXS function [6-267](#)
- INDICES function [6-268](#)
  - and secondary indexes [6-268](#)
  - in transactions [6-270](#)
- INFO.ABORT option [6-18](#)
- INFO.CONVERT option [6-18](#)
- INFO.ENTER option [6-18](#)
- INFO.INCLUDE option [6-18](#)
- INFO.LOCATE option [6-18](#), [6-316](#),  
[6-320](#)
- INFO.MARKS option [6-18](#)
- INFO.MOD option [6-18](#)
- in-line prompts, clearing [6-102](#)
- INMAT function [6-272](#)
  - after MATPARSE statement [6-343](#)
  - after MATREAD statement [6-346](#)
  - after OPEN statement [6-381](#)
- INPUT @ statement [6-273](#)
- INPUT statements [6-273](#)
  - and DATA statement [6-128](#)
- INPUT.ELSE option [6-19](#), [6-275](#)
- INPUTAT option [6-19](#)
- INPUTCLEAR statement [6-278](#)
- INPUTDISP statement [6-279](#)
- INPUTDP statement [6-280](#)
- INPUTERR statement [6-281](#)
- INPUTIF statement [6-273](#)
- INPUTNULL statement [6-283](#)
- INPUTTRAP statement [6-284](#)
- INS statement [6-285](#)

## INSERT function [6-288](#)

- and LOCATE statement [6-317](#),  
[6-321](#), [6-325](#)

## INT function [6-291](#)

## INT.PRECISION option [6-19](#)

## intent file locks [4-4](#)

## internal system date [6-130](#)

## international date conversion [C-17](#)

## international time conversion [C-52](#)

## intrinsic functions, *see* functions

## ISNULL function [2-3](#), [2-16](#), [6-292](#)

- with CASE statement [6-89](#)

- with IF statement [6-258](#)

## ISNULLS function [2-3](#), [2-16](#), [6-293](#)

## isolation levels [4-14](#), [6-69](#)

- minimum locks for [4-14](#)

- types [4-11](#)

## isolation property [4-7](#)

## ISOMODE parameter [4-13](#)

## ITYPE function [6-294](#)

## K

## KEEP.COMMON keyword [6-93](#)

## keyboard keys

- control [6-297](#), [6-302](#), [6-305](#)

- escape [6-298](#), [6-302](#), [6-305](#)

- function [6-298](#), [6-302](#), [6-305](#), [F-33](#)

- unsupported [6-300](#)

## KEYEDIT statement [6-296](#)

## KEYEXIT statement [6-302](#)

## KEYIN function [6-304](#)

## KEYTRAP statement [6-305](#)

## keywords [1-2](#)

- definition [1-3](#)

## L

## L conversion [C-23](#)

## labels, statement [1-5](#)

## LE operator [2-15](#)

## LEFT function [6-307](#)

- LEN function [6-308](#)
- LENDP function [6-309](#)
- length function [C-23](#)
- LENS function [6-310](#)
- LENSDP function [6-311](#)
- LES function [6-312](#)
- LET statement [6-313](#)
- levels, *see* isolation levels
- line number table, suppressing [3-2](#), [3-4](#)
- list variables, *see* select list variables
- listing object code [5-12](#)
- LN function [6-314](#)
- local cataloging [3-11](#)
- local variables [6-93](#)
- LOCALEINFO function [6-315](#)
- LOCATE statement [6-316](#), [6-320](#), [6-324](#)
- LOCATE.R83 option [6-19](#)
- lock escalation, example [4-15](#)
- LOCK statement [6-327](#)
- locks
  - compatibility [4-1](#)
  - deadlocks [4-5](#)
  - exclusive file lock [4-5](#)
  - file lock [6-195](#)
  - granularity [4-1](#)
  - intent file lock [4-4](#)
  - and MATREADL statement [6-348](#)
  - and MATREADU statement [6-348](#)
  - process lock [6-598](#)
  - and READL statement [6-420](#)
  - and READU statement [6-354](#), [6-420](#), [6-607](#)
  - and READVL statement [6-421](#)
  - releasing [6-354](#), [6-444](#), [6-598](#), [6-607](#)
  - semaphore lock [6-327](#)
  - shared file lock [4-4](#)
  - shared record lock [4-2](#)
  - transactions and [4-8](#)
  - types [4-2](#), [6-192](#)
  - update record lock [4-3](#)
  - well-formed writes and [4-13](#)

- logical operators [2-17](#)
  - AND [2-17](#)
  - NOT [2-17](#)
  - and the null value [2-17](#)
  - OR [2-17](#)
- LOOP statement [6-329](#)
- loops
  - FOR...NEXT [6-216](#), [6-366](#)
  - LOOP...REPEAT [6-329](#)
- lost updates [4-12](#)
- LOWER function [6-332](#)
- LT operator [2-15](#)
- LTS function [6-334](#)

## M

- masked character conversions [C-24](#)
- MAT statement [6-335](#)
- MATBUILD statement [6-337](#)
- MATCH operator [2-16](#), [6-338](#)
- MATCHFIELD function [6-340](#)
- mathematical functions [C-4](#), [C-19](#)
- MATPARSE statement [6-342](#)
  - and INMAT function [6-272](#)
- MATREAD statement [6-345](#)
  - and INMAT function [6-272](#)
- MATREADL statement [6-348](#)
  - and INMAT function [6-272](#)
- MATREADU statement and INMAT function [6-272](#)
- matrices [2-6](#)
  - zero element [2-6](#)
- MATWRITE statement [6-352](#)
- MATWRITEU statement [6-354](#)
- MAXIMUM function [6-357](#)
- MAXRLOCK parameter [4-15](#)
- MB conversion [C-37](#)
- MC conversion [C-24](#)
- MD conversion [C-26](#)
- messages
  - error [3-9](#), [6-52](#), [6-166](#), [6-406](#), [6-520](#)
  - warning [3-9](#)



- MINIMUM function 6-358
- ML conversion C-29
- MM conversion C-32
- MO conversion C-37
- MOD function 6-359
- MODS function 6-360
- monetary conversion C-32
- MP conversion C-34
- MR conversion C-29
- MT conversion C-35
- MU0C conversion C-37
- MULS function 6-361
- multivalues and arithmetic
  - operators 2-13
- MX conversion C-37
- MY conversion C-39

## N

- named common variables 2-5
- names of variables 1-2, 2-5
- NAP statement 6-362
- NE operator 2-15
- NEG function 6-363
- NEGS function 6-364
- NES function 6-365
- nested transactions 4-6
  - committing 6-112
  - example 4-9
  - properties 4-7
- newlines 1-5
- NEXT statement 6-216, 6-366
- NL conversion C-40
- NLS monetary conversion C-32
- NLS*mapname* conversion C-41
- NO.CASE option 6-19
- NO.RESELECT option 6-19
- NOBUF statement 6-367
  - with TTYSET statement 6-592
- nonrepeatable reads 4-12
- normal cataloging 3-11

## NOT

- function 6-368
- operator 2-17
- NOTS function 6-369
- NR conversion C-42
- NULL statement 6-370
- null value
  - in arithmetic expressions 2-12
  - and concatenation operators 2-13
  - definition 2-3
  - and dynamic arrays 2-25
  - and logical operators 2-17
  - stored representation 2-4
- NUM function 6-371
- numbers, floating-point 2-2
- numeric
  - character strings 2-12
  - characters 1-6
  - constants 2-3
  - data 2-2
  - functions 1-2
- NUMS function 6-372

## O

- object code 1-2, 3-9
  - listing 5-12
- OCONV function 6-373, C-1, C-8, C-37
- OCONVS function 6-376, C-6
- octal conversion C-37
- ON statement 6-377
- ONGO.RANGE option 6-19, 6-378
- OPEN statement 6-380
  - and INMAT function 6-272
- OPENCHECK statement 6-251, 6-384
- OPENCHK parameter 6-251, 6-352,  
6-381, 6-605
- OPENDEV statement 6-386
- OPENPATH statement 6-389
- OPENSEQ statement 6-392
  - and CREATE statement 6-126
  - and NOBUF statement 6-367

- operators [2-11-2-25](#)
  - arithmetic [2-11-2-13](#)
  - assignment [2-14, 2-19](#)
  - concatenation [2-13](#)
  - and dynamic arrays [2-19-2-25](#)
  - logical [2-17](#)
  - pattern matching [2-16](#)
  - relational [2-15-2-16](#)
  - string [2-13](#)
  - substring [2-14](#)
- operators, assignment [6-28, 6-62](#)
- OR operator [2-17, 6-72, 6-397](#)
- ORS function [6-397](#)

## P

- P conversion [C-43](#)
- packed decimal conversion [C-34](#)
- PAGE statement [6-398](#)
- part files, status [6-517](#)
- part numbers, status [6-517](#)
- passing
  - arrays to subroutines [6-87, 6-147, 6-527](#)
  - variables to subroutines [6-147, 6-527](#)
- pattern matching [2-16, 6-338, C-43](#)
  - and empty strings [2-17](#)
  - codes [6-338](#)
- PCLOSE.ALL option [6-19](#)
- PERFEQ.EXEC option [6-19, 6-399](#)
- PERFORM statement [6-399](#)
- PHANTOM command [3-1](#)
- phantom writes [4-12](#)
- PIOPEN.EXECUTE option [6-19](#)
- PIOPEN.INCLUDE option [6-19](#)
- PIOPEN.MATREAD option [6-20](#)
- PIOPEN.SELIDX option [6-20, 6-485](#)
- pointer (REMOVE) [6-235, 6-462, 6-496](#)
- PRECISION statement [6-401](#)
- PRINT statement [6-402](#)
  - and INPUT statements [6-274](#)

- and TABSTOP statement [6-538](#)
- PRINTER CLOSE statement [6-404](#)
- PRINTER statement [6-404](#)
- PRINTER statement [6-406](#)
- process locks [6-598](#)
- PROCREAD statement [6-408](#)
- PROCWRITE statement [6-409](#)
- PROGRAM statement [6-410](#)
- PROMPT statement [6-411](#)
  - and INPUT statements [6-274](#)
- prompts, *see* in-line prompts
- PWR function [6-412](#)

## Q

- Q conversion [C-44](#)
- quotation marks in character strings [2-2](#)
- QUOTE function [6-413](#)

## R

- R conversion [C-46](#)
- RADIANS option [6-20](#)
- RAID [5-5](#)
- RAID (debugger) [6-132](#)
  - commands [5-4, 6-132](#)
  - description [5-1](#)
- RAID command [5-2](#)
  - options [5-2](#)
  - suppressing execution of [3-2](#)
- RAISE function [6-414](#)
- RANDOMIZE statement [6-416](#)
  - and RND function [6-466](#)
- range function [C-46](#)
- RAW.OUTPUT option [6-20](#)
- READ statement [6-417](#)
- READ.RETAIN option [6-20, 6-421](#)
- READBLK statement [6-423](#)
  - and TIMEOUT statement [6-562](#)
- READL locks, *see* shared record locks
- READL statement [6-417](#)

- READLIST statement [6-426](#)
- READNEXT statement [6-428](#)
  - and READLIST statement [6-426](#)
  - and SELECT statement [6-481](#)
- READSEQ statement [6-430](#)
  - and TIMEOUT statement [6-562](#)
- READT statement [6-432](#)
- READU locks, *see* update record locks
- READU statement [6-417](#)
- READV statement [6-417](#)
- READVL statement [6-417](#)
- READVU statement [6-417](#)
- REAL function [6-438](#)
- REAL.SUBSTR option [6-20](#)
- RECORDLOCKED function [6-442](#)
- RECORDLOCKL statement [6-439](#)
- RECORELOCKU statement [6-439](#)
- relational operators [2-15–2-16](#)
- RELEASE statement [6-444](#)
- REM
  - function [6-446](#)
  - statement [1-4, 6-447](#)
- REMOVE
  - function [6-448](#)
  - pointer [6-235, 6-462, 6-496](#)
  - statement [6-450](#)
- removing
  - identifiers [3-6](#)
  - spaces [6-576, 6-578, 6-579, 6-580, 6-581, 6-582, 6-596](#)
  - tabs [6-576, 6-578, 6-579, 6-580, 6-581, 6-582, 6-596](#)
- REPEAT statement [6-329](#)
- REPLACE function [6-28, 6-175, 6-453](#)
- reserved words in BASIC [D-1](#)
- RETURN (*value*) statement
  - and DEFFUN statement [6-134](#)
- RETURN (*value*) statement [6-222](#)
- RETURN statement [1-3, 6-87, 6-236, 6-457](#)
- RETURN TO statement [6-236](#)
- REUSE function [2-24, 6-460](#)

- REVREMOVE statement [6-462](#)
- REWIND statement [6-464](#)
- RIGHT function [6-465](#)
- RND function [6-466](#)
  - and RANDOMIZE statement [6-416](#)
- RNEXT.EXPL option [6-20, 6-428](#)
- ROLLBACK statement [6-467](#)
- Roman numeral conversion [C-42](#)
- RPC.CALL function [6-469](#)
- RPC.CONNECT function [6-471](#)
- RPC.DISCONNECT function [6-473](#)
- RUN command [3-10](#)
  - options [3-10](#)

## S

- S (soundex) conversion [C-47](#)
- S (substitution) conversion [C-48](#)
- SADD function [6-474](#)
- saving variables in unnamed
  - common [6-93](#)
- scientific notation [2-3](#)
- SCMP function [6-475](#)
- SDIV function [6-476](#)
- secondary indexes and BASIC
  - INDICES function [6-268](#)
- SEEK statement [6-477](#)
- SEEK(ARG.) statement [6-479](#)
- select lists
  - clearing [6-103](#)
  - variables [2-9, 6-103, 6-484, 6-508](#)
- SELECT statement [6-481](#)
- SELECTE statement [6-484](#)
- SELECTINDEX statement [6-485](#)
- SELECTINFO function [6-487](#)
- SELECTN statement [6-22, 6-483](#)
- SELECTV statement [6-21, 6-482](#)
- semaphore locks [6-327](#)
- SEND statement [6-488](#)
- SENTENCE function [6-489](#)
- SEQ function [6-490](#)
- SEQ.255 option [6-20, 6-490](#)

- SEQS function 6-491
- sequential I/O 1-5
- sequential processing 6-107, 6-126, 6-200, 6-367, 6-386, 6-392, 6-423, 6-430, 6-478, 6-562, 6-613, 6-616
- serializability
  - lost updates and 4-13
  - property 4-8
- SET TRANSACTION ISOLATION
  - LEVEL statement 4-8, 6-492
- SETLOCALE function 6-494
- SETPTR command F-40
- SETREM statement 6-496
- shared file locks 4-4
- shared memory 3-13
- shared record locks 4-2
- SIN function 6-497
- SINH function 6-498
- SLEEP statement 6-499
- SMUL function 6-500
- soundex conversion C-47
- SOUNDEX function 6-501
- source code 1-2
  - comments 1-4
  - spaces in 1-5
  - syntax 1-3
  - tabs in 1-5
- SPACE function 6-502
- spaces
  - in numeric constants 2-3
  - in source code 1-5
  - removing 6-576, 6-578, 6-579, 6-580, 6-581, 6-582, 6-596
- SPACES function 6-503
- special characters 1-6
- SPLICE function 6-504
- SQRT function 6-505
- square brackets ( [ ] ) 6-49
- SQUOTE function 6-506
- SSELECT statement 6-507
- SSELECTN statement 6-509
- SSELECTV statement 6-508
- SSUB function 6-510
- standard arrays 2-6
  - matrices 2-6
  - vectors 2-6
- statement labels
  - cross-reference table of 3-2, 3-3
  - definition 1-5
- statements 1-2
  - assignment 1-4
  - control 1-4
  - types 1-4
- STATIC.DIM option 2-6, 6-20, 6-147
- STATUS function 6-511
  - after BSCAN statement 6-80, 6-511
  - after DELETE statement 6-511
  - after FILELOCK statement 6-511
  - after FMT function 6-512
  - after GET statement 6-512
  - after GETX statement 6-512
  - after HUSH statement 6-249
  - after ICONV function 6-254, 6-512
  - after ICONVS function 6-256
  - after INPUT @ statement 6-512
  - after MATWRITE statement 6-512
  - after OCONV function 6-373, 6-512
  - after OCONVS function 6-376
  - after OPEN statements 6-513
  - after OPENPATH statement 6-390
  - after READ statement 6-513
  - after READBLK statement 6-514
  - after READL statement 6-514
  - after READSEQ statement 6-430, 6-514
  - after READT statement 6-433, 6-464, 6-514, 6-601, 6-619
  - after READU statement 6-514
  - after READVL statement 6-514
  - after READVU statement 6-514
  - after RECORDLOCKED
    - statement 6-443
  - after REWIND statement 6-514

- after RPC.CALL function 6-469, 6-514
- after RPC.CONNECT
  - function 6-471, 6-514
- after RPC.DISCONNECT
  - function 6-473, 6-514
- after SELECTINDEX
  - statement 6-485
- after WEOF statement 6-514
- after WRITE statements 6-512
- after WRITESEQ statement 6-613
- after WRITET statement 6-514
- STATUS statement 6-516
  - values 6-516
- STOP statement 6-520
- STOPMSG option 6-20, 6-52, 6-520
- STOPE statement 6-22, 6-520
- STOPM statement 6-22, 6-520
- STORAGE statement 6-522
- STR function 6-523
- string functions 1-2, C-19
- string operators 2-13
- strings
  - see also* character strings
  - comparing 2-15
  - determining length 2-17
- STRS function 6-524
- SUBR function 6-525
- SUBROUTINE statement 1-3, 6-86, 6-527
  - and SUBR function 6-525
- subroutines
  - calling 6-86
  - definition 1-3
  - entering external 6-160
  - list of F-2
  - passing arguments to 1-3
  - passing arrays to 6-87, 6-147
  - returning from 6-457
  - returning values from 6-525
  - vector functions 2-20
- SUBS function 6-528
- substitution C-48
- substring operator 2-14
- substrings 2-13–2-15
  - and assignment operators 2-14
  - definition 2-13
  - extracting 2-14, 6-49
- SUBSTRINGS function 6-529
- subtransactions 4-6
- subvalue marks 2-7
- SUM function 6-530
- SUMMATION function 6-532
- SUPP.DATA.ECHO option 6-20, 6-275
- symbol table, suppressing 3-2, 3-4
- syntax, source code 1-3
- system date 6-130
- system delimiters 2-7
- SYSTEM function 6-533
  - host name 6-536
  - values 6-533

## T

- T conversion C-49
- tabs
  - removing 6-576, 6-578, 6-579, 6-580, 6-581, 6-582, 6-596
  - in source code 1-5
- TABSTOP statement 6-538
  - and PRINT statement 6-402, 6-600
  - and TPRINT statement 6-567
- TAN function 6-539
- TANH function 6-540
- TERMINFO function 6-541
  - table of EQUATEs 6-541–6-559
- text extraction C-49
- Tfile conversion C-50
- TI conversion C-52
- tilde (~) 6-338
- time conversion C-35, C-52
- TIME function 6-560
- TIME.MILLISECOND option 6-21
- TIMEDATE function 6-561

- TIMEOUT statement 6-562
- TPARM function 6-564
- TPRINT statement 6-567
- TRANS function 6-569
- TRANSACTION ABORT
  - statement 6-572
- TRANSACTION COMMIT
  - statement 6-574
- TRANSACTION START
  - statement 6-575
- transaction statements 6-571
- transaction variables
  - @ISOLATION 4-10
  - @TRANSACTION 4-10
  - @TRANSACTION.ID 4-10
  - @TRANSACTION.LEVEL 4-10
- transactions 4-5
  - @variables 4-10
  - active 4-6
  - and data visibility 4-6
  - example 4-9
  - isolation levels 4-8
  - locks and 4-8
  - nested 4-6
    - committing 6-112
  - properties 4-7
  - and RELEASE statements 6-444
  - restrictions 4-10
  - subtransactions 4-6
  - and UV/Net 6-608
- TRIM function 6-576
- TRIMB function 6-578
- TRIMBS function 6-579
- TRIMF function 6-580
- TRIMFS function 6-581
- TRIMS function 6-582
- TTYCTL statement 6-583
- TTYGET statement 6-585
  - values 6-586
- TTYSET statement 6-591
- type 1 files 1-7, 3-1
- type 19 files 1-7, 3-1

## U

- UID, effective 6-65
- ULT.FORMAT option 6-21
- UNASSIGNED function 6-593
- UNICHAR function 6-594
- UNICHARS function 6-595
- UNISEQ function 6-596
- UNISEQS function 6-597
- UNIX *vi* editor 1-7
- UNLOCK statement 6-598
  - and LOCK statement 6-327
- unnamed common variables 2-5
- unnamed common, saving variable
  - values 6-93
- unsupported keys, defining 6-300
- UNTIL statement 6-216, 6-329
- UPCASE function 6-599
- update record locks 4-3
- UPRINT statement 6-600
- USE.ERRMSG option 6-21, 6-407
- user ID 6-65
- user-written functions 6-459
- UV/Net
  - AUTHORIZATION statement 6-65
  - RPC.CONNECT function 6-471
  - SYSTEM function 6-536
  - TIMEOUT statement 6-562
  - and transactions 6-608
  - WRITE statement 6-608
  - writing to remote files 4-10, 6-608
- UVLOCALE.H include file 6-234, 6-315

## V

- value marks 2-7
- VAR.SELECT option 6-21, 6-103, 6-233, 6-426, 6-427, 6-482, 6-483, 6-508, 6-509
- variables 1-2, 2-4-2-9
  - array 2-5-2-9

- assigning 6-62, 6-313
- common 6-114
- definition 1-2
- file 2-9
- in RAID 5-3
- in user-written functions 6-222
- local 6-93
- named common 2-5
- names 1-2, 2-5
- passing to subroutines 6-147, 6-527
- saving in unnamed common 6-93
- select list 2-9, 6-103, 6-484, 6-508
- transaction 4-10
- unnamed common 2-5, 6-93
- VEC.MATH option 2-13, 2-20, 6-21
- vector functions 2-20
  - as subroutines 2-20
- vectors 2-6
  - zero element 2-6
- vi editor 1-7
- VLIST command 5-12
  - suppressing execution of 3-2

## W

- warning messages 3-9
- well-formed write 4-13
- WEOF statement 6-601
- WEOFSEQ statement 6-601, 6-602
- WHILE statement 6-216, 6-329
- WIDE.IF option 6-21
- WRITE statement 6-604
- WRITEBLK statement 6-610
- WRITELIST statement 6-612
- WRITESEQ statement 6-613, 6-616
- WRITESEQF statement 6-616
- WRITET statement 6-618
- WRITEU statement 6-604
- WRITEV statement 6-604
- WRITEVU statement 6-604

## X

- XLATE function 6-623
- XTD function 6-625

## Z

- zero element 2-6





To help us provide you with the best documentation possible, please make your comments and suggestions concerning this manual on this form and fax it to us at **508-366-3669, attention Technical Publications Manager**. All comments and suggestions become the property of Ardent Software, Inc. We greatly appreciate your comments.

## Comments

Name: \_\_\_\_\_ Date: \_\_\_\_\_

Position: \_\_\_\_\_ Dept: \_\_\_\_\_

Organization: \_\_\_\_\_ Phone: \_\_\_\_\_

Address: \_\_\_\_\_

\_\_\_\_\_

Name of Manual: **UniVerse BASIC**

Part Number: **70-9002-952**

