

# MultiUser BASIC®

---

## *Language Reference*

*Version 2.1*



Third Edition: January, 2001

Copyright © 1995, 2001 by THEOS Software Corporation.

All rights reserved.

The software described in this document is furnished under a license agreement or nondisclosure agreement. The software may be used only in accordance with the terms of the agreement. Information in this document is subject to change. No part of this manual may be reproduced, transmitted, transcribed or translated into any language in any form by any means without the written permission of THEOS Software Corporation.

THEOS Software Corporation  
1801 Oakland Boulevard, Suite 315  
Walnut Creek, CA 94596-7000

Telephone: 925 935-1118  
Fax: 925 935-1177  
E-mail: [sales@theos-software.com](mailto:sales@theos-software.com)  
WWW: <http://www.theos-software.com>

THEOS®, the THEOS logo, THEOS MultiUser BASIC® and WindoWriter® are registered trademarks and THEOS C™ are trademarks of THEOS Software Corporation. Microsoft®, Windows® and Windows NT® are registered trademarks of Microsoft Corporation.

Printed in the United States of America

Documentation by THEOS Software Corporation

# Table of Contents

---

<b>Preface</b> .....	13
THEOS MultiUser BASIC .....	13
Notation Conventions .....	14
 <b>THEOS MultiUser BASIC Features</b> .....	15
The Interpreter .....	15
The Compiler .....	15
Program Segmentation .....	15
Data Files .....	16
Sequential files .....	16
Direct files .....	16
Indexed files .....	16
Keyed files .....	17
C Language Subroutines .....	17
Multitasking .....	17
Compatibility .....	17
Other Features .....	18
Features New to MultiUser BASIC .....	20
 <b>1 MultiUser BASIC Command</b> .....	23
 <b>2 Fundamentals</b> .....	25
Line Numbers .....	26
Line Labels .....	27
Global Labels .....	27
Local Labels .....	28
Constants .....	28
Numeric Constants .....	28
Size .....	28
Scientific Notation .....	29
Hexadecimal Integer Format .....	29
Punctuation .....	29
String Constants .....	29
Named Constants .....	31
Named Constant Types .....	31
Named Constant Names .....	31
Variables .....	32
Integer and Numeric Variables .....	33
String Variables .....	33
Assigning Values to Variables .....	33
Array variables .....	34
Creating Arrays .....	35
Using the Dimensioning Statements .....	35

Variable Scope and Duration . . . . .	36
Scope . . . . .	36
Duration . . . . .	37
Global Variables . . . . .	37
SHARED Variables . . . . .	37
COMMON Variables . . . . .	38
Local Variables . . . . .	38
LOCAL Variables . . . . .	38
STATIC Variables . . . . .	38
Automatic Variables . . . . .	38
Expressions . . . . .	39
Expressions . . . . .	39
Expression Evaluation . . . . .	39
Operator Precedence . . . . .	40
Arithmetic Expressions . . . . .	41
String Expressions . . . . .	42
Logical or Boolean Expressions . . . . .	44
Binary Expressions . . . . .	45
Relational Expressions . . . . .	48
Functions . . . . .	49
Intrinsic Functions . . . . .	49
User-defined Functions . . . . .	49
Subroutines . . . . .	50
Subprograms . . . . .	51
Include Files . . . . .	52
 <b>3 Commands . . . . .</b>	 53
THEOS MultiUser BASIC Program Editor . . . . .	56
Using Other Text Editors . . . . .	56
Using the MultiUser BASIC Interpreter . . . . .	57
Command Text Delimiters . . . . .	57
AUTO . . . . .	59
BOTTOM . . . . .	61
BREAK . . . . .	62
CHANGE . . . . .	66
COMPILE . . . . .	68
CONTINUE . . . . .	69
COPY . . . . .	70
DELETE . . . . .	72
DOWN . . . . .	74
HELP . . . . .	75
INDENT . . . . .	76
LENGTH . . . . .	79
LIST . . . . .	80
LOAD . . . . .	82
LOCATE . . . . .	83

LPLIST .....	84
LPXREF .....	86
MERGE .....	87
MODIFY .....	89
MOVE .....	91
NAME .....	92
NEW .....	93
OLDMOD .....	94
QUIT .....	97
RECALL .....	99
RENUMBER .....	101
RUN .....	103
SAVE .....	105
SHOWSTACK .....	110
STEP .....	111
TOP .....	113
TRACE .....	114
UNBREAK .....	118
UNTRACE .....	120
UP .....	121
VARs .....	122
VIEW .....	124
XREF .....	131

<b>4 Functions and Statements .....</b>	<b>139</b>
ABS .....	150
ACOS .....	151
ACOT .....	153
ACSC .....	154
ACTIVATE .....	155
ADDROF .....	158
ANGLE .....	160
ASC .....	161
ASEC .....	162
ASIN .....	163
AT\$ .....	164
ATAN, ATN .....	165
AVAIL.WINDOWS .....	166
BIN, BINOF\$ .....	167
BREAK .....	169
CALL .....	170
CALL.RETURN.VALUE .....	171
CASE .....	172
CEIL .....	175
CEND .....	176
CHAIN .....	177

CHR\$ .....	178
CLEAR .....	179
CLOSE .....	180
CLS\$ .....	181
CMDARG\$ .....	182
COLOR .....	183
COMMON .....	184
CONSTANT .....	187
CONTINUE .....	188
COS .....	189
COSH .....	191
COT .....	192
COTH .....	193
CRT\$ .....	194
CSC .....	200
CSCH .....	201
CSI .....	202
CSI.RETURN.CODE .....	204
DATA .....	206
DATE\$ .....	207
DAY .....	209
DECLARE CALL .....	211
DEF FN .....	212
DEG .....	217
DEL\$ .....	218
DELETE .....	219
DIM .....	220
DTE\$ .....	222
ELSE .....	225
END .....	226
END SUB .....	228
EOF .....	229
EPS .....	230
ERL .....	231
ERR .....	233
ERRMSG\$ .....	234
EVENT .....	236
EXP .....	237
EXT\$ .....	238
FILL .....	239
FILL BAR .....	241
FILL CIRCLE .....	243
FILL PIE .....	246
FIX .....	251
FLOAT .....	252
FLOOR .....	253

FNEND .....	254
FOR.....	255
FORMAT\$.....	259
FP .....	262
GCD .....	263
GET.....	264
GET COMMON.....	266
GOSUB.....	268
GOTO .....	269
HEX, HEXOF\$ .....	271
IF.....	272
IFEND .....	275
INCLUDE .....	276
INF .....	278
INP .....	279
INPUT .....	280
INS\$ .....	285
INT .....	287
IOLIST.....	288
IP.....	290
KILL.....	291
UBOUND.....	293
LCASE\$ .....	294
LEFT\$.....	295
LEN.....	296
LET .....	298
LINE .....	303
LINK .....	305
LINPUT.....	306
LINPUT USING.....	310
LOCAL .....	314
LOCKED.BY .....	317
LOG, LOG2, LOG10 .....	318
LPAD\$.....	319
LRL, LRR, LSL, LSR.....	320
LTRIM\$.....	323
MAT.....	324
MAT INPUT.....	329
MAT PRINT.....	333
MAT READ .....	336
MAT READNEXT .....	339
MAT READPREV .....	342
MAT SORT .....	345
MAT WRITE .....	349
MATCH.....	351
MAX .....	352

MID\$ . . . . .	353
MIN . . . . .	354
MOD . . . . .	355
MOUNT . . . . .	356
MOUSE.BUTTON, .COL, .FRAME, .ROW, .WINDOW . . . . .	357
MSEC . . . . .	362
NBR . . . . .	363
NEXT . . . . .	364
NEXT.FILE . . . . .	365
OCT, OCTOF\$ . . . . .	366
ON ERROR . . . . .	367
ON EVENT . . . . .	369
ON GOSUB . . . . .	372
ON GOTO . . . . .	374
ON KEY . . . . .	375
ON.KEY.TOKEN . . . . .	380
ON MOUSE . . . . .	381
ON TIMEOUT . . . . .	385
OPEN . . . . .	388
OPTION . . . . .	396
ORD . . . . .	398
OTHERWISE . . . . .	399
OVR\$ . . . . .	400
PAGE . . . . .	401
PI . . . . .	403
PLOT . . . . .	404
PLOT ARC . . . . .	406
PLOT BAR . . . . .	410
PLOT CIRCLE . . . . .	412
PLOT PIE . . . . .	414
POS . . . . .	418
PRINT . . . . .	419
PRINT USING . . . . .	427
PUBLIC . . . . .	439
PUT . . . . .	441
PUT COMMON . . . . .	443
QUIT . . . . .	444
RAD . . . . .	446
RANDOMIZE . . . . .	447
READ, READNEXT, READPREV . . . . .	448
REDIM . . . . .	455
REM . . . . .	457
REP\$ . . . . .	458
RESET . . . . .	459
RESTORE . . . . .	461
RESUME . . . . .	462



RETURN	465
RIGHT\$	467
RND	468
ROUND	470
RPAD\$	471
RPT\$	472
RTRIM\$	473
RUN	474
SCH	475
SEC	477
SECH	478
SECOND	479
SELECT	481
SEMAPHORE	483
SET	485
SET FILL	486
SET LINE	489
SET MARKER	491
SET TEXT	493
SGN	496
SHARED	497
SIN	500
SINH	501
SLEEP	502
SPACE\$	503
SQR	504
STATIC	505
STOP	508
STR\$	510
STRTIME\$	511
SUB	514
SWAP	517
SYS.ENV\$	518
SYSTEM	520
TAB	522
TAN	523
TANH	524
TEXT	525
THEN	528
TIME\$	529
TIMER	530
TOTAL.WINDOWS	532
TRIM\$	533
UCASE\$	534
UNGET	535
UNLOCK	537

USED.WINDOWS .....	539
VAL .....	540
VDI .....	541
WAIT .....	558
WAIT EVENT .....	560
WEND .....	562
WHILE .....	563
WINDOW CHOICE .....	565
WINDOW CLEAR .....	576
WINDOW CLIP .....	578
WINDOW CLOSE .....	580
WINDOW COPY .....	581
WINDOW EDIT .....	583
WINDOW FRAME .....	588
WINDOW GET .....	591
WINDOW GET TITLE .....	593
WINDOW INVERT .....	594
WINDOW LOCATE .....	595
WINDOW MOVE .....	596
WINDOW OPEN .....	597
WINDOW REFRESH .....	605
WINDOW REMOVE .....	606
WINDOW RESTORE .....	608
WINDOW SAVE .....	610
WINDOW SELECT .....	612
WINDOW STATUS .....	617
WINDOW TAKE .....	621
WINDOW TITLE .....	623
WRITE .....	625
YESNO\$ .....	627

## Appendices

<b>A Language Summary .....</b>	<b>629</b>
<b>B THEOS Character Set .....</b>	<b>638</b>
<b>C Multinational Characters .....</b>	<b>639</b>
<b>D Error Codes and Messages .....</b>	<b>640</b>
<b>E MultiUser BASIC Files .....</b>	<b>648</b>
<b>F Tables .....</b>	<b>649</b>
Program Control Statement Interactions .....	649

Color Codes ..... 649

VDI Color Codes ..... 650

VDI Fill Patterns ..... 650

VDI Line Styles ..... 651

VDI Marker Styles ..... 651

MATCH Function Pattern Specifications ..... 652

ON KEY Terminal Key Codes ..... 652

ANSI Forms Control Characters ..... 653

Formatted Record Field Codes ..... 653



# Preface

---

Beginner's All-Purpose Symbolic Instruction Code (BASIC) is an English-like computer language that's easy to learn and use. It was developed in 1965 at Dartmouth College as a teaching tool. It has since become the most widely used computer language in the world.

Using BASIC, it is easy to learn to write computer programs without needing to know how a computer actually works. Its simplicity is appreciated by even experienced programmers who have found that BASIC can be used to write large, complex programs. BASIC is used for businesses, the scientific community, manufacturers, the home market, and for many other applications.

A standardized version of BASIC is controlled by the American National Standards Institute and is known as ANSI Minimal BASIC. As the name implies, ANSI Minimal BASIC, by itself, has very limited capabilities. For this reason, software developers add enhancements of their own to extend the range of capabilities available in the version of BASIC they produce and sell. Most developers, including THEOS, strive to make their enhancements conform to the style and stated goals of the ANSI standard.

## ■ THEOS MultiUser BASIC

The extensions added to THEOS MultiUser BASIC make it a full-featured language, supporting all of the capabilities and enhancements of other BASIC dialects. THEOS MultiUser BASIC also includes commands to generate VDI graphic displays, as well as support for the floating-point math coprocessor, screen colors, and screen windowing.

## ■ Notation Conventions

Throughout this manual, syntax is used that looks like:

**RUN** *program-name starting-line*


These symbols are used to show the correct syntax for typing the MultiUser BASIC commands and statements.

**BOLD** Words and characters shown capitalized and in boldface must be entered exactly as shown.

*italics* Italicized words show parameters whose value is supplied by the programmer.

[ ] Text inside of square brackets is optional and may be omitted. The square brackets should not be entered.

DIRECT Underlined portions of a word indicate the minimum spelling allowed for abbreviations of the word.

 Specific keys on the keyboard are indicated by icons representing the key.

| A vertical bar symbol is used to separate two or more choices, only one of which may be chosen.

... An ellipsis indicates that the preceding term may be repeated several times.

When necessary, standard terms are used in the syntax definition of the commands, functions, and statements. For example, *numeric-expression* or *string-variable*. These terms are defined in Chapter 2 “[Fundamentals](#),” starting on page [25](#).

In many cases, the terms used in the syntax definitions are indicative of the meaning of the parameter. For example: *input-mask*, *length*, and *semaphore*. These terms are defined in the descriptive text for the command, function, or statement.

When referring to operating system files, the term *nnn* will be used instead of a specific number. For example, the reference `SYSTEM.TEOSnnn` indicates a reference to the library `SYSTEM.TEOS386` or `SYSTEM.TEOS32`, depending upon the version of the THEOS operating system on the computer.

# THEOS MultiUser BASIC Features

---

Because it is a full-featured BASIC language, THEOS MultiUser BASIC includes many enhancements for writing programs. These features also improve the performance of the resulting programs.

## ■ The Interpreter

THEOS MultiUser BASIC programs are developed and tested using the MultiUser BASIC interpreter. In this mode the computer translates one statement to machine code, executes it, translates the next statement, executes it, *etc.* If a line cannot be executed, an error message describing what is wrong with that line is displayed and execution terminates, returning to the command mode with the current line pointer set to the problem line. With the interpreter, it is very easy to test and revise the program until it runs error-free from beginning to end.

## ■ The Compiler

Although a BASIC interpreter program will run properly, it does not perform as quickly as it would if it were written in machine code and did not need to be translated line by line. The THEOS MultiUser BASIC compiler converts MultiUser BASIC source programs into machine code and stores the machine code as a command file. The program operates the same way, interpreted or compiled, but compiled programs execute much faster.

## ■ Program Segmentation

When the computer is instructed to run a program, it loads the program into memory where it is held while the program instructions are processed. In previous versions of THEOS BASIC, the program size had to be small enough to fit in the small memory model (about 50K). THEOS MultiUser BASIC allows programs, and data used by programs, to be as large as the computer system's available memory, up to a maximum of 4Mb.

The only reason to break up a program into multiple modules is because of programming style or the dictates of the application design.

## ■ **Data Files**

In order to write programs that process large amounts of data, data files are used. THEOS, and THEOS MultiUser BASIC support four types of data files: Sequential, Direct, Indexed and Keyed—named according to the way the records are stored and retrieved.

### ■ **Sequential files**

Sequential files contain records that can only be accessed sequentially. In order to obtain the 20th record in a file, records 1—19 must be read first. The disk space used to store the file and the records is variable in length.

This file organization is frequently used in situations where the entire file is read into memory before being processed. Sequential files are ideal for work files and text data files because they make optimal usage of disk storage.

### ■ **Direct files**

Direct files contain records that may be read from or written to directly, by record number. All records in a direct file must be of the same length, but the file itself can be variable in length and can be expanded by THEOS as necessary.

Direct files are most useful where the data records are keyed by a simple numeric value. A good usage of the direct file is error message text. When a program wants the text associated with error number five, it merely reads record five.

### ■ **Indexed files**

Indexed files contain records that are keyed by a single alphanumeric field. Indexed files have fixed record lengths, but the file itself can be variable in length and can be expanded by THEOS as necessary. Individual records may be accessed by the computer without having to read through previous records.

Indexed files are read from the file, or written to the file by a record key, rather than by record number. THEOS maintains the keys in alphabetical order, using a modified B+ tree algorithm, which facilitates reading them in sorted order if necessary.



## ■ Keyed files

Keyed files are accessed by key (like indexed files), but the keys are not maintained in alphanumeric order. This arrangement enables MultiUser BASIC to search a KEYED file more quickly than an INDEXED file. Keyed files can be dynamically expanded by THEOS, and use fixed record lengths.

## ■ C Language Subroutines

MultiUser BASIC allows programs to use subroutines written in the THEOS C language. The CALL statement is used to reference routines written and compiled in the C language. Data and file pointers may be passed to the C language routine and the C language routine may modify data in the BASIC program and perform file input and output.

## ■ Multitasking

MultiUser BASIC includes statements and functions that support multitasking programs. Multitasking programs start other programs operating independently of the original program.

## ■ Compatibility

THEOS MultiUser BASIC is upward compatible with Dartmouth BASIC, ANSI minimal BASIC, THEOS 8 BASIC, THEOS 286 BASIC, and THEOS 386 BASIC.

## ■ Other Features

THEOS MultiUser BASIC contains many other features, including:

- ▶ Multiple statements on one line
- ▶ Line length up to 4,096 characters
- ▶ Line labels
- ▶ Local line labels
- ▶ Up to one million lines per program
- ▶ Multiple-line, user-defined functions
- ▶ An extended set of string functions
- ▶ An extended set of numeric functions
- ▶ An extended set of trigonometric functions
- ▶ Commands allowing the loading, saving and merging of unnumbered program source
- ▶ Bit-manipulating logic functions
- ▶ 13-digit precision Binary Coded Decimal (BCD) arithmetic
- ▶ Floating point values in the range of ten raised to the power of  $\pm 126$
- ▶ Integer arithmetic (-32768 to +32767)
- ▶ Extended length variable and label names
- ▶ String length up to 4GB
- ▶ String arrays
- ▶ Array input/output and assignment
- ▶ Direct sorting of single-dimension arrays
- ▶ Error trapping (`ON ERROR GOTO`)
- ▶ Timer and other event trapping (`ON EVENT GOTO`)
- ▶ Special key trapping (`ON KEY GOTO`)
- ▶ Complex `IF-THEN-ELSE` statements
- ▶ Multiple-line `IF-IFEND` structures
- ▶ Multiple-line `WHILE-WEND`
- ▶ Multiple-line `SELECT-CEND` structure
- ▶ Structured programming statements `BREAK` and `CONTINUE`

- ▶ Formatted output ([PRINT USING](#) and [FORMAT\\$](#))
- ▶ A standard interface for console cursor control
- ▶ Control of screen colors ([COLOR](#))
- ▶ Access to THEOS Window Manager capabilities ([WINDOW OPEN](#), [WINDOW CHOICE](#), [WINDOW SELECT](#), *etc.*)
- ▶ Cross-reference listing of variables
- ▶ Interface to THEOS operating system commands ([CSI](#) and [SYS-TEM](#))
- ▶ Access to environment values using the [SYS.ENV\\$](#) function
- ▶ Interface to VDI graphics capabilities
- ▶ Automatic recall of last command entered
- ▶ Automatic line number entry
- ▶ Syntax checking as statements are entered
- ▶ Program debugging commands
- ▶ Automatic use of floating point math coprocessor, when available

## ■ Features New to MultiUser BASIC

The following features are new to THEOS MultiUser BASIC Version 1.0:

- ▶ Large memory model. Source programs and compiled programs may be as large as available memory.
- ▶ Extended line number range. Prior versions were limited to 9,999 lines in each program. New limit is now 999,999 lines.
- ▶ Ability to save, load, and merge unnumbered source code.
- ▶ Data file records maintained from BASIC increased from the previous record length limit of 2,048 bytes to 32,767 bytes.
- ▶ New command to **COPY** or duplicate a section of the source program.
- ▶ New command to **MOVE** a portion of the program to another location.
- ▶ The **NEW** command enhanced to allow program name assignment.
- ▶ New **ON KEY** statement allowing a program to detect and service a key press event.
- ▶ New structured programming statements **BREAK** and **CONTINUE**. These statements abort or repeat a program structure such as **FOR-NEXT**, **WHILE-WEND**, *etc.*
- ▶ New statement, **SYSTEM**, allowing the execution of a program command without closing data files.
- ▶ New statements providing windowing capability. As many as eleven windows may be open and displayed at one time.
- ▶ New function, **SYS.ENV\$**, provides access to system environment values and utility functions such as keywords, system messages, spooled printer status, temporary file names, console session control, *etc.*
- ▶ New math package providing higher performance and more accurate trigonometric and math operations.
- ▶ **OPTION VERSION** allows developer to insert version number and copyright notice into a program.
- ▶ **COMPILE** command allows the loaded program to be compiled from the interpreter.
- ▶ More error messages are now trappable.
- ▶ New MultiUser BASIC Language Reference manual with better organization, descriptions, examples, and illustrations.

- ▶ New MultiUser BASIC Programmer's Guide providing procedures and working examples of programming techniques for THEOS MultiUser BASIC.

The following features are new to THEOS MultiUser BASIC Version 2.0:

- ▶ Line lengths extended from 255 characters to 4,096 characters.
- ▶ String lengths extended from 255 characters to 4Gb.
- ▶ The number of open files increased to 999.
- ▶ Variables declared with the new **LOCAL** statement are local to the current function definition (**DEF FN-FNEND**) or subroutine definition (**SUB-END SUB**).
- ▶ New **INCLUDE** statement that merges an external source program into the current program at compile time or interpreter execution time.
- ▶ Line labels defined with a double colon (::) are local to the current include program.
- ▶ New **SUB** and **END SUB** statements that define callable subprogram within a program. Line numbers and labels are local to the subprogram.
- ▶ New **IOLIST** statement that defines a sequence of variable references that can be used in all file input and output statements.
- ▶ New **ON MOUSE** statements that define actions to be taken when mouse activity is detected. Works similar to the **ON KEY** statement.
- ▶ New **CALL.RETURN.VALUE** function returns the value set by the last called C language function.
- ▶ New **ON TIMEOUT** statement providing a timed input capability to programs.
- ▶ New **NEXT.FILE** function returns the next unused file channel.
- ▶ New **WINDOW EDIT** statement allowing a "full screen edit" of an array of strings.
- ▶ Enhanced **WINDOW CHOICE** statement allowing program to control the size of the window used by the choice list. (Previously, the size was solely determined by the contents of the list.)
- ▶ Enhanced **LINPUT USING** statement allowing specification of a starting column number for input.

The following features are new to the current release of THEOS MultiUser BASIC Version 2.0:

- ▶ The function `STRTIME$` is provided to format a date and time for display and printing.
- ▶ The function `LOCKED.BY` provides a means of determining which user is locking a file or record needed by a program.
- ▶ Standard input and output redirection is supported allowing utility-type programs to be written to use input and output from files or devices other than the console.
- ▶ The new system environment variables `DATEIN` and `DATEOUT` are supported by the date functions `DATE$`, `DAY` and `DTE$`. These variables provide a means of allowing some applications to support year 2000 dates without major revisions.
- ▶ The new command `SHOWSTACK` is provided to display the current subroutines, subprograms and defined functions that are executing at the time program execution is interrupted.

# 1 MultiUser BASIC Command

THEOS MultiUser BASIC is a dual mode language product. It is both a program editor and interpreter as well as a compiler.

1 **BASIC32** ( **SIZE** *nnn*

2 **BASIC32** *program-name...* ( *options*

<i>options</i>	»	DISK	LINK	NOLINK	PRT <i>nn</i>
		DEBUG	LIST	NOOBJ	RUN
		EXPAND	LOAD	NOTYPE	STACKSIZE
		FILES	NOASM	NOWARNING	TYPE
		HEAPSIZE	NOBOUND	OLDER	

## Operation

**Mode 1**—Invokes the MultiUser BASIC program editor and interpreter.

**Mode 2**—Unless the option **LOAD** or **RUN** is specified, compiles the program or programs specified by *program-name*. The *program-name* specification may be explicit with file-name, file-type and member-name specification, or it may be a simple program name specification with implied file-type or member-name and default library name.

Alternately, the *program-name* specification may use wild cards. With wild cards, each file matching the specification is compiled.

The programs referred to by *program-name* may be ASCII or compressed MultiUser BASIC source programs. If the program is ASCII, the source may be numbered or unnumbered. When the first non-space character is a digit, the entire program is treated as a numbered source program and unnumbered lines are ignored. Otherwise, the entire program is treated as an unnumbered source program and the line numbers of numbered source lines are ignored.

<b>Options</b>	<b><u>DEBUG</u></b>	Used when compiling a program to create a DEBUGSYM file. This file is then used by the source-level debugger to assist you in debugging a compiled program.
	<b><u>LOAD</u></b>	Instead of compiling the program the interpreter is invoked and the requested source program is loaded.
	<b><u>RUN</u></b>	Instead of compiling the program, the interpreter is invoked, the requested source program is loaded and executed in interpretive mode.

**Notes** When the program editor/interpreter is invoked ([Mode 1](#) or [Mode 2](#) with option [LOAD](#)), the MultiUser BASIC prompt character is displayed. The prompt character is a dash (-).

Source program files on disk are either *flat* files or library members. A ***flat file*** is a file whose name is a simple fn.ft (*file-name.file-type*) such as CUSTOMER.BASIC, INVOICES.BASIC, STANDARD.SETUP, *etc.* The default file-type of BASIC or BAS should be used for flat file names to make it easier when specifying the name in compiles, loads, saves, merges, *etc.* These operations will use the default file-type of BASIC or BAS whenever it is not specified explicitly. For example, the command

```
>BASIC32 MYPROG
```

compiles the program MYPROG.BASIC or, if this is not found, then MYPROG.BAS.

**See also** LINK32, Make and WindoWriter commands

**Examples**

```
>B21 TEST (LOAD
MultiUser BASIC® Version 2.1
© 2001 by THEOS Software Corporation
-LEN
Main program size:      363
Included program size:  0
Total data size:       217090
Free data size:        42368
-
```



## 2 Fundamentals

---

THEOS MultiUser BASIC language programs are composed of lines of source code. Each line of the program has the following syntax:

*line-number* [ *line-label* ] [ *statement* [\ *statement* ] ... ] [! *comment* ]

Every line of a program has a line number, an optional line-label, and zero or more statements separated by the backslash character.

```
1000 REPORT:
1010
1020   GOSUB STATUS.BOX \ REM Display program status box
1030
1040   GOSUB OPEN.FILES \ REM Open files used for report
1050   GOSUB HEADING \ REM Print page one heading
1060   GOSUB DO.REPORT \ REM Create and print report
1070
1080   IF NOT ABORT% THEN GOSUB TOTALS
1090
1100   RETURN
```

Although each of the possible statements in a program has their own syntax, there are many common elements used by all the statements and functions. Chapter 4 “[Functions and Statements](#)” describes the syntax for each statement and function. The common elements used throughout a MultiUser BASIC program include:

**Line number** Every line starts with a line number. The value of the number determines the location of the line within the source program. (A source program created outside of the interpreter does not have to have line numbers. However, when it is loaded by the interpreter or the compiler, line numbers are automatically assigned to each line.)

**Statement keyword** Every statement starts with a keyword that identifies the type and function of the statement.

**Constants** Numeric and alphanumeric values, specified at the time the program is written, that never change.

**Variables** Numeric and alphanumeric fields whose value can change during the execution of the program.

**Operators** Symbols indicating an operation on one or more constants, variables, or functions.

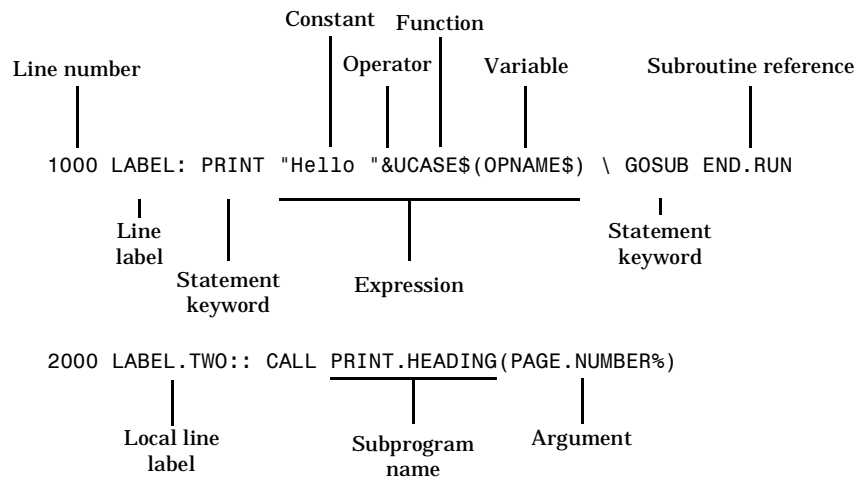
**Expressions** The combination of one or more constants, variables, functions or operators.

**Functions** A predefined or user-defined procedure that has a single value determined at the time of execution.

**Subroutine and subroutine references** A collection of statements defined once in a program but executed from many locations in the program.

**Subprogram and subprogram references** Similar to a subroutine except that it can be passed different information each time it is executed and it has variables and line labels local to itself.

**Line references** A line number or a line label.



## ■ Line Numbers

All of the lines in a MultiUser BASIC program must have a line number.

Line numbers serve two purposes: the line number determines the position of the line within the program and serves as a label. For instance, line number 20 precedes line number 30 and follows line number 10. The line number may also be used as the object or destination of a GOTO, GOSUB, RETURN, RESUME, ON ... GOTO and ON ... GOSUB statement.

The line numbers may have a range of values between 1 and 999999.

## ■ Line Labels

A line label is a descriptive identifier for a program line. For instance:

```
1000 READ.NEXT.RECORD:
1010
1020 READNEXT #1, KEY$: NAME$
1030
```

Line labels may be specified on the same line as a statement or on a separate line by itself.

```
1000 READ.NEXT.RECORD: READNEXT #1, KEY$: NAME$
```

Line labels may be as long as 64 characters and may contain letters, digits and the period character. Space characters are not allowed as part of a line label.

Line label definitions are terminated with either a single colon character (:) or two colon characters (::). When a line label is used in a statement as an object, the label is specified without any terminator. For instance:

```
4000 GOSUB CALCULATE.TOTALS
```

## ■ Global Labels

**Global line labels** are labels that have meaning globally, throughout the entire program. They are defined with a single colon character terminator. For instance:

```
2000 PRINT.TOTALS:
2010
2020 PRINT #16, USING TMASK$: TOTAL
2030
2040 RETURN
```

The environment from which a line label has meaning is referred to as its **scope**. Thus, the scope of a global line label is throughout the entire program, including any subroutines, function definitions and subprogram definitions. (Exceptions occur if the global line label is defined within a function definition or a subprogram definition. Refer to the explanations for each of these types of program objects.)

A global line label is the normal type of line label.

## ■ Local Labels

A **local line label** is a line label whose scope is only within a restricted environment. It is defined with a double colon character terminator. For instance:

```
2000 PRINT.TOTALS::
2010
2020 PRINT #16, USING Tmask$: TOTAL
2030
2040 RETURN
```

A local line label may only be defined as part of an *include program* file. Local line labels are useful for specifying a location with a name that might be in use by the main program or another included program file.

When a local line label is spelled the same as a global line label, and a reference is made to the label from within the local line label's environment, the local line label location is used.

Some line labels defined with a single colon are actually local line labels if they are defined with a subprogram or user-defined function.

## ■ Constants

All programming languages have the ability to represent problem-solving routines that manipulate logical and mathematical relationships in data. This section describes the terminology needed to construct THEOS MultiUser BASIC programs that can solve simple math problems, large equations, or particular operations on a series of alphabetical characters.

### ■ Numeric Constants

Constants are specific numbers used in MultiUser BASIC programs, such as 3, -7, or 0.49. A constant is a value that does not change during program execution. A program that calculates the diameter of a circle, might use the formula:

$$D = 3.14 \times R$$

The number 3.14 is a numeric constant in this formula.

#### • Size

The value of a numeric constant has a range of  $\pm 9.999999999999 \times 10^{\pm 126}$ . Numeric values have a maximum of 13 significant digits. Numbers speci-

fied with more than 13 digits are truncated to that length (the digits following the 13<sup>th</sup> digit are dropped).

- **Scientific Notation**

It is possible to write larger numbers in MultiUser BASIC by using scientific or exponential notation, which expresses a number as a constant multiplied by a power of ten. The power of ten can be anything in the range of +126 to -126. In MultiUser BASIC, this notation is indicated by the letter E, indicating exponentiation. The following list illustrates this format:

Constants	Scientific Notation
.0000009	.9E-6
0.00013	1.3E-4
0.0013	1.3E-3
7000008.00	7.000008E6
300000.00	3.0E5

MultiUser BASIC always tries to express numbers in normal notation. If it can't, it normalizes the entry by placing the decimal after the first digit. For example, the value 123456700000000 is normalized to 1.234567E+14.

- **Hexadecimal Integer Format**

Integer constants are entered in base ten (decimal) format, or in base 16 (hexadecimal) format. A hexadecimal value must start with a digit (0 if necessary), and end with the letter H. The following examples represent valid hexadecimal integer constants:

1243H      -1243H      0ACH      0AH

- **Punctuation**

Since commas separate numeric values, they are never used within MultiUser BASIC numbers. Negative numbers must use a leading minus sign. Positive numbers may have a leading plus sign, but its use is not mandatory.

- **String Constants**

A string constant is one or more letters, numbers or other characters enclosed within quotation marks. String constants, like numeric constants, retain their value throughout the operation of the program. The value of a string constant is determined by all the characters, including

spaces, that appear between the quotation marks. The only character that cannot be used in a string constant is a line terminator (or CR character).

Either single or double quotation marks define the start and end of a string constant, but they may not be mixed except to indicate a quotation within the string. The following examples show legal string constants:

```
"This is a legal string constant"
'And so is this'
'She said, "This is a way to indicate quotes."'
'He replied, 'And this is another.'"
```

These string constants are also called string literals, because the `PRINT` statement prints this data literally. The above examples display as:

```
This is a string constant
And so is this
She said, "This is a way to indicate quotes."
He replied, 'And this is another.'
```

It is possible to define a string constant with a pair of quotation marks and to have the same type of quotation mark embedded within the string. This is done by doubling the embedded quotation mark:

```
"She said, ""This is a way to indicate quotes."""
```

prints as:

```
She said, "This is a way to indicate quotes."
```

## ■ Named Constants

A named constant is a programming convenience that allows a programmer to assign a name to a frequently-used constant value. The name and value of a named constant is defined with the `CONSTANT` declaration.

```
CONSTANT PI = 3.14259256
CONSTANT FALSE% = 0
CONSTANT COMPANY.NAME$ = "THEOS Software Corporation"
```

Named constants are defined by a precompiler or preexecution declaration and do not, by themselves, use any memory. References to a named constant perform and operate exactly like a reference to a normal constant. The advantage of a named constant over a normal constant is:

- ▶ Symbolic name can indicate the meaning of the constant value.
- ▶ The value of all references to the constant can be easily changed by changing the single declaration for the named constant.
- ▶ The named constant is effectively *public in scope* meaning that it can be referenced from any location in the program.

### • Named Constant Types

The name of the constant indicates its type of value:

- ▶ Integer constant names terminate with a percent sign ( % ).
- ▶ String constant names terminate with a dollar sign ( \$ ).
- ▶ Numeric constant names end with a letter, digit or period.

### • Named Constant Names

MultiUser BASIC allows constant names up to 64 characters in length, not counting the terminating % or \$ character. The naming conventions are the same for all named constant types:

- ▶ The first character must be a letter (A—Z).
- ▶ Subsequent characters are optional, and may include letters (A—Z), digits (0—9) or periods.
- ▶ Spaces are not permitted as part of a constant name.
- ▶ The constant name must not be a MultiUser BASIC keyword, unless it is terminated with a % or \$ character.
- ▶ A constant name may not be the same as a predefined or user-defined variable or function name.

## ■ Variables

A variable is different from a constant because its value may change as the program executes.

A variable is known by its name rather than its value. The name given to a variable in a MultiUser BASIC program is assigned a location in memory; each value assigned to the variable is stored in that location.

The name of the variable indicates its type of variable:



- ▶ Integer variable names terminate with a percent sign ( % ).
- ▶ String variable names terminate with a dollar sign ( \$ ).
- ▶ Numeric variables end with a letter, digit or period.

MultiUser BASIC allows variable names up to 64 characters in length, not counting the terminating % or \$ character. The naming conventions are the same for all variable types:



- ▶ The first character must be a letter (A—Z).
- ▶ Subsequent characters are optional, and may include letters (A—Z), digits (0—9) or periods.
- ▶ Spaces are not permitted as part of a variable name.
- ▶ The variable name must not be a MultiUser BASIC keyword, unless it is terminated with a % or \$ character.
- ▶ A variable name may not be the same as a predefined or user-defined function name.

Examples of variable names that MultiUser BASIC recognizes:

Name	Variable Type
A	Numeric
A%	Integer
A\$	String
RECORD.NO	Numeric
CODE%	Integer
NAME\$	String



## ■ Integer and Numeric Variables

Integer variables can contain integer values in the range of +32767 to -32768. Assigning a value to an integer variable that is greater than the allowable size generates an “overflow” condition and causes the integer variable to be assigned the value -32768.

Numeric variables contain floating point values. The range of valid floating point numbers depends upon the presence and usage of a floating point processor. When a floating point processor is not used numeric values range from  $9.99999999999 \times 10^{+126}$  to  $1 \times 10^{-126}$ . When a floating point processor is used numeric values range from  $1.797693134862 \times 10^{+308}$  to  $2.225073858507 \times 10^{-308}$ .

Integer and numeric variables are automatically initialized to zero when the variable is first defined. To initialize the variable to a different value, use the **LET** statement to assign the value.

## ■ String Variables

A string variable is a named location in which a set or string of alphanumeric characters is stored. Strings have both a value and a length. As with integer and numeric variables, when a string variable is first referenced, MultiUser BASIC initializes the location to a length of zero. (This is known as a *null string*.)

During the execution of a program, the length of the string stored in that location can vary from zero to a maximum length of 4 billion characters or one-half of the currently available memory on your system.

## ■ Assigning Values to Variables

There can be many variables in one program. In the following example program, memory locations A, B%, C\$, and D are used. The **LET** statement is used to assign values to these variables. For example:

```

10      A = 5.5
20      B% = 10
30      C$ = "THE VALUE OF A + B% IS:"
40      D = A + B%
50      PRINT C$ \ PRINT D
60      END

```

```

-RUN
THE VALUE OF A + B% IS: 15.5

```

## ■ Array variables

The variables discussed so far in this chapter are restricted to storing only one data value per variable. This type of variable is sometimes referred to as a **scalar** variable. It is difficult to manipulate large amounts of data in a program unless many variable names are used.

MultiUser BASIC provides a special type of variable known as an array variable. An array variable reserves a set of memory locations. These locations can be as large as the program requires to store the data that will be used in numerical calculations or string manipulations.



**Note:** An array variable name cannot be used as a scalar variable name. Attempting to do so results in an “Inconsistent usage,” “Array variable used as a scalar” or “Scalar variable used as an array” error message.

An array can be visualized by thinking of a grid composed of rows and columns with one data element in each cell. If the array is set up with only one row, it is called a **one-dimensional array**. When the array has two or more rows, it is known as a **two-dimensional array**.

One-dimensional array:

0	1	2	3	4
---	---	---	---	---

Two-dimensional array:

0,0	0,1	0,2	0,3	0,4
1,0	1,1	1,2	1,3	1,4
2,0	2,1	2,2	2,3	2,4

To refer to a specific item in an array, use the array name followed by the cell coordinate reference numbers enclosed in parentheses. These reference numbers are called **subscripts** and the array variables are often referred to as **subscripted variables**.

This illustration shows a two-dimensional array of four rows and five columns named EXAMPLE%. The data shown in each cell is the coordinate references for that location.

		Columns				
		0	1	2	3	4
R o w s	0	(0,0)	(0,1)	(0,2)	(0,3)	(0,4)
	1	(1,0)	(1,1)	(1,2)	(1,3)	(1,4)
	2	(2,0)	(2,1)	(2,2)	(2,3)	(2,4)
	3	(3,0)	(3,1)	(3,2)	(3,3)	(3,4)

Notice the first row and the first column is each numbered 0. This means that, even though there are five columns and four rows, the largest subscript in this array is `EXAMPLE%(3,4)`. Using a subscript larger than the size of the array results in the error message “Subscript range error.”

It may be awkward to work with arrays numbered in this manner. For your convenience, MultiUser BASIC allows arrays to begin with row and column number 1 by using the `OPTION BASE 1` statement in your program. The default option is `BASE 0`.

The same array with `OPTION BASE 1` would be:

		Columns				
		1	2	3	4	5
Rows	1	(1,1)	(1,2)	(1,3)	(1,4)	(1,5)
	2	(2,1)	(2,2)	(2,3)	(2,4)	(2,5)
	3	(3,1)	(3,2)	(3,3)	(3,4)	(3,5)
	4	(4,1)	(4,2)	(4,3)	(4,4)	(4,5)

■ Creating Arrays

There are two ways to create arrays. An array can be implicitly created by simply using the `LET` statement to assign a name, followed by the subscript(s). Arrays created by this method are limited to subscripts 0–10 in `BASE 0`, or subscripts 1–10 in `BASE 1`.

For example:

```
NAME$(3) = "Jonah"
```

Creates a one-dimensional array called `NAME$` and assigns “Jonah” to location three.

```
ITEM.COST%(0,9) = 11.65
```

Creates a two-dimensional array called `ITEM.COST%` and assigns the value 11.65 to row zero, column nine.

■ Using the Dimensioning Statements

Arrays created with the `LET` statement are of a fixed size. One-dimensional arrays hold either 10 or 11 variables, depending upon the `OPTION BASE` in effect. Two-dimensional arrays have spaces reserved for either 121 variables (`BASE 0`), or 100 variables (`BASE 1`). To create larger or smaller

arrays the **DIM**, **COMMON**, **SHARED**, **LOCAL** or **STATIC** statements must be used to assign the array name and establish the dimensioned size.

To create the array called **EXAMPLE%** in **BASE 1**, use this statement:

```
DIM EXAMPLE%(4,5)
```



**Note:** It is good programming practice to create arrays by always using one of the dimensioning statements. This habit avoids wasted storage space, and provides visible documentation of the arrays and their size.

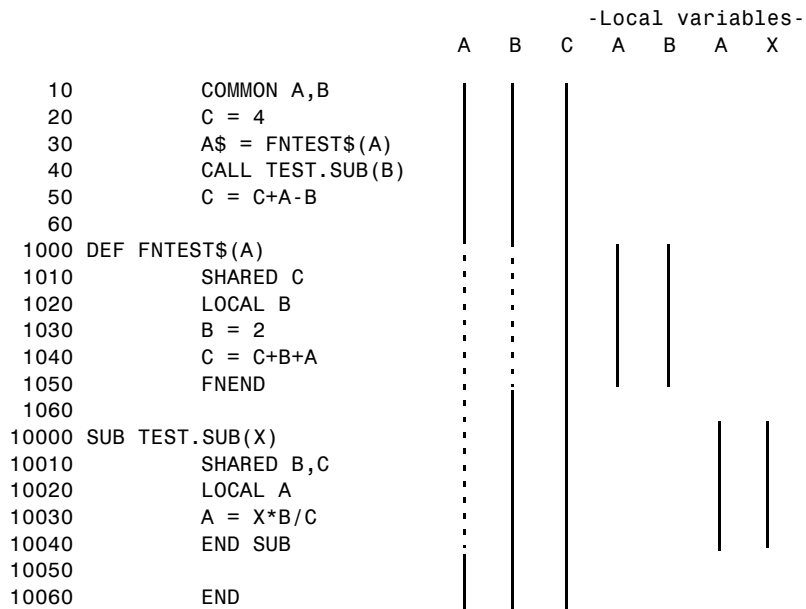
### ■ Variable Scope and Duration

Variables have **scope** and **duration** attributes. The scope of a variable refers to where the variable can be referenced. The duration of a variable refers to how long a variable and its value exist.

#### • Scope

Variable scope may be either global or local. If a variable does not have scope in a specific environment, then a reference to that variable name refers to a different variable that is local in scope.

The following example and diagram illustrates variable scope.



Although there are only five different variable names used in this program example, there are in fact eight different variables used. The additional variables come about because some of the variables are local in scope and therefore refer to a different variable than the variable name that is global in scope. For instance, the B variable name in the function definition at line 1030 and 1040 refers to a different variable than the B variable name used at lines 40 and 50.

The scope of a variable can be explicit by declaring the variable name with the **COMMON**, **LOCAL**, **SHARED** and **STATIC** statements or implicit by having the variable name take on the default scope for the environment.

The default scope for subprograms is local; the default in all other situations is global. Functions and subprograms receive passed values as local, automatic variables.

- **Duration**

Variable duration may be either static or temporary and refers to how long the value stored in a variable, and how long the variable itself, exists. A static variable continues to exist and have value even when it doesn't have scope. A temporary variable ceases to exist when it no longer has scope. Temporary variables are initialized to zero or an empty string when they begin to have scope. Static variables are not re-initialized.

The duration of a specific variable is directly tied to its scope declaration.

- **Global Variables**

There are actually two levels of global variables: global within a program or global between programs that are linked together in execution. A global variable may be referenced and assigned values from any location in a program, including user-defined functions and subprograms.

- **SHARED Variables**

Variable names used in the main portion of a program are global within the program. A subprogram can use these global variables by first declaring them with the **SHARED** statement.

Shared, global variables have temporary duration: They are created and initialized when the program is first started and exist until the program exits. Each time that the program is executed the variables are re-initialized.

- **COMMON Variables**

A program may declare certain variables with the **COMMON** statement. A common variable is a special type of shared, global variable that has scope between programs, as long as both programs declare them as **COMMON**.

Common, global variables have static duration: They are created and initialized when the program is first started with the **RUN** statement or command and exist until the program exits or a program is linked to that does not declare them as **COMMON**.

- **Local Variables**

There are two types of local variables: temporary duration local and static duration local. A local variable may only be referenced and assigned values from a location within the scope of the variable. Local variables may only be declared inside of a user-defined function or subprogram.

- **LOCAL Variables**

Variable names used in a subprogram are local in scope. A user-defined function can use local variable names by first declaring them with the **LOCAL** statement.

Local variables have temporary duration: They are created and initialized each time the subprogram or function is invoked and they exist until the subprogram or function is exited (**END SUB** or **FNEND** statement).

- **STATIC Variables**

Variable names declared with the **STATIC** statement are local variables that have static duration: They are created and initialized when a program is first started and exist until the program exits. Local, static variables are not re-initialized each time that the function or subprogram is called.

- **Automatic Variables**

Automatic variables are a special type of variable that is local in scope and temporary in duration. Automatic variables are the arguments passed to user-defined functions and subprograms and cannot be declared as local or global, they are always local. They exist and can be used only as long as the function or subprogram is executing.

## ■ Expressions

A powerful feature of programming languages is the ability to use expressions that specify a series of operations to be performed using constants and variables. Expressions, or formulas, can perform arithmetic and string manipulations, binary logic and comparative relationships. Prewritten expressions, called functions, perform many standard operations. The following sections provide a review of terms, expressions, operators and functions.

### ■ Expressions

An expression consists of terms and operators. A term may be as small as a single constant or it may be a combination of several terms, which may be expressions in themselves. The five types of expressions (arithmetic, string, logical, binary and relational) may be combined into complex expressions, except that string expressions may not be combined with arithmetic, logical, or binary expressions.

Most expressions resemble normal arithmetic equations or expressions. The following are typical expressions in MultiUser BASIC:

```
43
3+2*A
PI*R^2
```

### ■ Expression Evaluation

When an expression contains more than one operator, MultiUser BASIC performs the operations in the order of the established precedence of the operators. Here are some elementary rules to keep in mind when working with expressions:

- ▶ When the operators are of equal precedence, MultiUser BASIC performs the operations in the order of their occurrence — from left to right. Otherwise, the operations are performed in order of precedence. For example, in the expression  $3*2/4.5$ , the multiplication and division operators have equal precedence. This expression is evaluated in a left to right manner: the product of  $3*2$  is divided by 4.5.
- ▶ Parentheses are assigned the highest priority. Any term or expression inside of the parentheses is evaluated first before any term or expression which is not within the parentheses. When nested parentheses are encountered, MultiUser BASIC works from the inside out. The terms inside the innermost pair of parentheses are evaluated first, then those in the next outward layer. For example,

in the expression  $3 * (2 / 4.5)$  the quotient of  $2 \div 4.5$  is multiplied by 3.

- Arithmetic operations containing both integer and numeric data will have all the integer data converted to numeric prior to any operations being performed.

### ■ Operator Precedence

The following table shows the operators available in THEOS MultiUser BASIC in their order of precedence:

Operator	Description
()	Grouping
^	Exponentiation (raise to power)
[n:n]	Substring
<i>function</i>	Predefined functions
+ -	Unary Positive and Negative operators
* /	Multiplication & Division
+ -	Addition & Subtraction
&	String concatenation
> >= < <= = <>	Relational operators
NOT	Logical NOT
NOT	Binary NOT
AND	Logical AND
& AND	Binary AND
OR	Logical OR
OR	Binary OR
XOR	Binary Exclusive OR
EQV	Binary Equivalence
IMP	Binary Implies

Operators that affect only one term are called *unary* operators. The substring and NOT operators are always unary operators. “NOT” precedes the modified term, while the substring operator follows the affected term.

Operators that affect two terms are called *binary* operators. Most of the listed operators are binary operators. All of the binary operators are placed between the two terms they affect.

The plus and minus signs may be either binary or unary operators. As unary operators, they are used to specify the sign of the following single



term. As binary operators, they specify the addition or subtraction of two terms.

The grouping operator may be used to specify logical groupings of as many terms as desired. It is not limited to unary or binary operation.

■ Arithmetic Expressions

Most expressions are arithmetic. The syntax for these expressions is:

*term operator term*

or

*unary operator term*

Arithmetic terms can be:

- ▶ Numeric constants ..... 123.56
- ▶ Integer constants ..... 14
- ▶ Numeric variables ..... BALANCE
- ▶ Numeric arrays elements ..... MONTH(12)
- ▶ Integer variables ..... I%
- ▶ Integer arrays elements ..... ITEMS%(15)
- ▶ Numeric functions ..... FLOAT(2)
- ▶ Integer functions ..... AVAIL.WINDOWS
- ▶ Logical expressions ..... A OR B
- ▶ Binary expressions ..... B IMP C
- ▶ Relational expressions ..... A > B
- ▶ Other arithmetic expressions ..... A + (B\*C)^2

Arithmetic expressions use these operators:

Operation	Operator	Meaning
Exponentiation	^	Raises a number to a power
Multiplication	*	Multiplies two values
Division	/	Divides one value by another
Addition	+	Adds two values (or unary positive)
Subtraction	-	Subtracts two values (or unary negative)

For example:

$$X\% / Y\% + Z\%^2$$

The value of the integer variable Z% is squared. The integer variable X% is divided by the integer variable Y%. The result of this division is added to the previously calculated value of  $z\%^2$ .

$$SALARY - COLA(49) * SALARY + DIVIDEND$$

The numeric variable in location 49 of the array named COLA is multiplied by the numeric value of the variable named SALARY. The result is subtracted from the value of SALARY, and the value of DIVIDEND is added. The result is a numeric value.

$$X * (Y\% - .95)$$

The numeric constant .95 is subtracted from the numeric value of the variable Y%. The result is then multiplied by the numeric value of the variable X resulting in a numeric value.

### ■ String Expressions

String expressions use only string terms. The syntax for string expressions is:

*term operator term*

or

*term substring-operator*

String terms can be:

- ▶ String constants ..... "Page "
- ▶ String variables ..... HEADING\$
- ▶ String array elements ..... NAMES\$(4)
- ▶ String functions ..... LEFT\$(HEADING\$,4)
- ▶ String expressions ..... HEADING\$&TITLE\$

The string operators are:

Operator	Meaning
•&	•Concatenation: Joins two strings together
•[n:m]	•Substring: Extracts a subset from the string

The concatenation operator makes one long string from two shorter ones.

The substring operator makes a shorter string from a longer one. The designators *n* and *m* represent the location of the starting and ending characters. The length of the string will always be *m*−*n*+1.

In the following examples, NAME\$ = "THEOS" and MESSAGE\$ = "Have a good day."

```
"The operating system is "&NAME$
```

The result is: "The operating system is THEOS."

```
NAME$[1:3]&MESSAGE$[7:15]
```

The result is: "THE good day."



**Note:** The substring operator can appear on the left side of the assignment operator. Refer to the [LET](#) statement for a description of this special notation.

**Restrictions:** When the substring operator is used, the second number must always be greater than or equal to the first number. For example, the following is invalid:

```
NAME$[4:2]
```

Arithmetic operators cannot be used with string terms. For example:

```
NAME$ + MESSAGE$
```

The "&" operator could be used here to combine the two terms, but the "+" operator indicates an arithmetic operation and can only be used with numeric terms.

Arithmetic terms cannot be used with string operators. For example:

```
MESSAGE$ & A%
```

The **STR\$** function can be used to convert a numeric term to a string term, so the following would be acceptable:

```
MESSAGE$ & STR$(A%)
```

### ■ Logical or Boolean Expressions

Logical expressions are integer expressions using one or more of the logical operators. Logical expressions evaluate to a “True/False” or **Boolean** value. A false value is a zero; a true value is a –1 (binary 1111111111111111).



For comparison purposes, any non-zero value is treated as a true value. For example: 3 OR 2 evaluates to a true value because both the 3 and the 2 values are non-zero and are treated as true values during expression evaluation.

True/False logical expressions compare, combine or negate the logical “truth” or “falsehood” of the affected terms. There are three true/false logical operators:

Operator	Meaning
NOT	Negate term following
AND	Compares two terms; when both terms true then result is true, otherwise result is false.
OR	Compares two terms; when both terms false then result is false, otherwise result is true.

**Special Note:** . The operators NOT, AND, and OR are normally treated as binary operators. The **OPTION LOGICAL** directive **must** be specified to enable logical expression evaluation.

The syntax for a logical expression is:

*term logical-operator term*

or

NOT *term*

The following truth tables illustrate the results of each of the logical operators for all combinations of term values:

NOT Operator	
X%	NOT X%
false	true
true	false

OR Operator		
X%	Y%	X% OR Y%
false	false	false
false	true	true
true	false	true
true	true	true

AND Operator		
X%	Y%	X% AND Y%
false	false	false
false	true	false
true	false	false
true	true	true

**Restrictions:** Logical operators cannot be used with string terms. Logical operators, other than NOT, cannot be combined. For example, the following is not allowed:

A% AND OR B%

## ■ Binary Expressions

Binary expressions are integer expressions using one or more of the binary operators. Binary expressions compare, combine or negate the actual “bits” of the affected integer terms. Each integer is composed of 16 bits; each bit is either a zero (0) or a one (1).

Binary operations always operate on integer values and yield integer results. When a numeric value is used, MultiUser BASIC converts the number to an integer before the expression is evaluated. Large numeric values (values larger than 32,767 or less than -32,768) are converted to the integer value -32,768. The fractional portion of any numeric value is ignored when it is converted to an integer. For instance, 23.678 is converted to the integer value 23; 100,000.0 is converted to the value -32,768.

There are six binary operators: NOT, AND ( & ), OR ( | ), XOR, IMP, and EQV. The first three (NOT, AND, and OR) are normally treated as binary operators. However, if the **OPTION LOGICAL** directive is specified in the program these three operators become logical operators. The symbolic representations of AND ( & ) and OR ( | ) are always binary operators, even when **OPTION LOGICAL** has been specified.

When **OPTION LOGICAL** is in effect, there is no NOT binary operator. To perform a binary negate operation use the subexpression:

*-term-1*

The syntax for a binary expression is:

*term binary-operator term*

or

NOT *term*

Integer terms can be:

- ▶ Integer constants .....22
- ▶ Integer variables .....INDEX%
- ▶ Integer array elements .....QUANTITY%(INDEX%)
- ▶ Integer functions .....USED.WINDOWS
- ▶ Integer arithmetic expressions.....YEAR% - 1900
- ▶ Logical expressions .....KEY\$ OR NOT EOF(4)
- ▶ Relational expressions .....KEY\$ > LAST.KEY\$
- ▶ Binary expressions .....FLAGS% XOR CURRENT%

The binary operators are defined as follows:

NOT	Each bit in the affected term is reversed. That is, all zeroes are changed to ones and all ones are changed to zeroes.
AND	If each corresponding bit in both terms is 1, the corresponding bit in the result is 1. Otherwise the bit in the result is 0 (zero).
OR	If either of the corresponding bits is 1, the bit in the result is a one. Otherwise, the bit in the result is a 0 (zero).
XOR	(Exclusive OR) If either, but not both, of the corresponding bits in both terms is a 1, the result is a 1. Otherwise, the resulting bit is 0 (zero).
IMP	(Implication) If each corresponding bit in both terms is a 1, the result is a 1. Otherwise, that bit in the result is the same as the corresponding bit in the second term.
EQV	(Equivalence) If each corresponding bit in both terms is the same (either both 1 or both 0), the result is a 1. If they are different, the bit in the result is a 0 (zero).

The following truth tables illustrate the results of each of these operations for every possible combination of bits:

NOT Operator	
X%	NOT X%
0	1
1	0

AND Operator		
X%	Y%	X% AND Y%
0	0	0
0	1	0
1	0	0
1	1	1

OR Operator		
X%	Y%	X% OR Y%
0	0	0
0	1	1
1	0	1
1	1	1

XOR Operator		
X%	Y%	X% XOR Y%
0	0	0
0	1	1
1	0	1
1	1	0

IMP Operator		
X%	Y%	X% IMP Y%
0	0	1
0	1	1
1	0	0
1	1	1

EQV Operator		
X%	Y%	X% EQV Y%
0	0	1
0	1	0
1	0	0
1	1	1

In the following examples, A% = 1000, B% = 100, and C% = -1217.

Expression	Decimal	Binary
A% AND B%	1000	0000 0011 1110 1000
	100	0000 0000 0110 0100
Result:	96	0000 0000 0110 0000
A% OR B%	1000	0000 0011 1110 1000
	100	0000 0000 0110 0100
Result:	1004	0000 0011 1110 1100
A% XOR C%	1000	0000 0011 1110 1000
	-1217	1111 1011 0011 1111
Result:	-1833	1111 1000 1101 0111
B% EQV C%	100	0000 0000 0110 0100
	-1217	1111 1011 0011 1111
Result:	1188	0000 0100 1010 0100
A% IMP B%	1000	0000 0011 1110 1000
	100	0000 0000 0110 0100
Result:	-905	1111 1100 0111 0111

**Restrictions:** Binary operators cannot be used with string data.

## ■ Relational Expressions

Relational expressions compare two terms to determine if a specified relationship exists. The results of that test will either be true (–1), or false (zero). Note that a true result has all bits in the integer set to one (1), while in a false result all bits in the integer are set to 0 (zero). The syntax for relational expressions is:

*numeric-term relational-operator numeric-term*

or

*string-term relational-operator string-term*

Relational terms may be either arithmetic terms or string terms. The relational operators are:

Operator	Meaning
>	Greater than
>=	Greater than or equal
<	Less than
<=	Less than or equal
=	Equal
<>	Not equal

In the following examples, NAME\$ = "Garcia", A% = 15, B = 15.0, and C = 5.

A% > B

The result is false (0) because 15 is not greater than 15.0.

C < B

The result is true (–1) because 5 is less than 15.0.

B <= A%

The result is true because 15.0 is less than or equal to 15.

NAME\$ = "GARCIA"

The result is false; "Garcia" and "GARCIA" are not the same string.

**Restrictions:** Data types cannot be mixed. That is, you cannot compare a string to a numeric value.



## ■ Functions

Functions are predefined expressions that perform a computation or other operation and return a value. Functions may be intrinsic functions or user-defined functions.

### ■ Intrinsic Functions

Intrinsic functions are those which are provided as a built-in (intrinsic) part of MultiUser BASIC. There are many intrinsic functions and they do such things as compute square roots, generate random numbers, perform special input/output operations, manipulate strings, etc. These functions are listed in Chapter 4 “[Functions and Statements](#),” starting on page 139.

### ■ User-defined Functions

User-defined functions are routines defined by the program using them. A user-defined function name always starts with the letters FN followed by one or more letters, digits and periods. For instance: FN.GET.FIELD.

A user-defined function must be defined within the program that uses the function. A user-defined function is defined with the [DEF FN](#) statement and may be a single-line definition or a multiple-line definition. Multi-line function definitions are terminated with the [FNEND](#) statement. For instance:

```
DEF FN.DIGIT.STRIP$(STRING$)

  LOCAL WORK$,I%,RETURN.STRING$

  FOR I% = 1 TO LEN(STRING$)
    WORK$ = STRING$(I%:I%)
    IF WORK$>="0" AND WORK$<="9"
      RETURN.STRING$ = RETURN.STRING$&WORK$
    IFEND
  NEXT
  LET FN.DIGIT.STRIP$ = RETURN.STRING$

FNEND
```

A user-defined function is executed by referencing its name as part of an expression anywhere in a program that defines it. Arguments may be passed to a user-defined function and these arguments are received as *automatic local variables* to the function. The function also has access to all of the global variables, line numbers and labels defined in the program. A function may define additional variables with the [LOCAL](#) or [STATIC](#) statements. These additional variables are local to the function definition only.

```
PRINT FN.DIGIT.STRIP$(INPUT.STRING$)
```

Line labels and line numbers defined between the `DEF FN` statement and the `FNEND` statement are *local line labels* and *local line numbers*. That is, they may be referenced only from within the function definition.

You cannot use a user-defined function as an argument passed to another user-defined function.

## ■ Subroutines

A subroutine is a series of statements that is defined once in a program but may be executed from many locations within a program. For instance:

```
1000  GOSUB READ.RECORD
1010  GOSUB CALCULATE.TOTALS
1020  GOSUB ACCEPT.CHANGES
1030  GOSUB CALCULATE.TOTALS
...
```

In this sequence, the subroutine `CALCULATE.TOTALS` is referenced and executed twice, although it is only defined once elsewhere in the program. The subroutine may be a single statement (plus the `RETURN` statement) or it may be several hundreds or thousands of statements.

A subroutine is executed by the `GOSUB` statement and exited with the `RETURN` statement. Although no arguments may be passed to a subroutine it has access to all of the global variables, line numbers and labels defined in the program. A subroutine does not have local variables, unless it is defined as part of an *subprogram*, in which case it has access to the local variables of the subprogram that it is defined within.

```
READ.RECORD:

  READNEXT #1,KEY$: IOLIST CUSTOMER.RECORD

  RETURN
```

## ■ Subprograms

A **subprogram** is a section of code that is somewhat similar to a subroutine but with some very significant differences:

- ▶ Subprograms can be called with argument passing; subroutines cannot.
- ▶ Subprograms can have *local variables*; subroutines cannot.
- ▶ Variables used in a subprogram are local to that subprogram only, unless they are explicitly stated as being **SHARED** variables.
- ▶ Subprograms can only be entered at the top and exited at the end; subroutines can be entered and exited anywhere in the routine.

A subprogram is also somewhat similar to a multi-line function definition with some significant differences:

- ▶ Subprograms operate as a statement; functions operate as expressions.
- ▶ Both subprograms and functions can be passed arguments (*call by value*) but subprograms can be passed arrays and pointers to variables (*call by reference*); functions cannot.
- ▶ Variables used in a subprogram are local to that subprogram only, unless they are explicitly stated as being **SHARED** variables; variables used in a function are global unless they are explicitly stated as being **LOCAL** or **STATIC**.

In general, subprograms provide a better tool for creating modular program designs than either functions or subroutines, particularly when the subprogram is defined as part of an *include file*.

## ■ Include Files

An *include file* is a source program file containing subroutines, functions, subprograms and other code that is incorporated into another source program file with the **INCLUDE** statement. The principal advantages and reasons for using included files in an application are:

- ▶ Creation and maintenance of common code and routines that can be used by more than one program in an application and even in several different applications.
- ▶ Use of *local line labels*. Local line labels defined with the double colon only have meaning when they are used as part of an included program file. The local line labels may only be referenced from code defined in the include file itself.

The code of an included program is an extension of the code in the program that includes it. For instance:

```
10 PRINT "Main program logic"
20 INCLUDE "PART2"
```

The next line executed after line 10 is the first line of the `PART2.BASIC` program.

Certain restrictions and features are provided with included program files. The line numbers of an included program are separate from the line numbers of the including program and cannot be referenced from locations outside of the included program itself. Line labels in the included program can be referenced from other parts of the program.

# 3 Commands

---

This section describes each of the commands in THEOS MultiUser BASIC in alphabetical sequence. The following is a list of those commands grouped by major function.

In the following lists some commands are marked with special symbols.

## General:

HELP	Display syntax of all commands, statements, and functions.
INDENT	Indent the entire program showing program structure.
LENGTH	Display size of program and data space used.
QUIT	Exit .
RECALL	Recall the last command entered.

## Program naming, saving, loading, and executing:

COMPILE	Compile the current program.
LOAD	Retrieve a saved program from disk.
NAME	Change the name of the program.
NEW	Clear memory for a new program.
RUN	Begin execution of a program.
SAVE	Write the program to disk in default save mode.
SAVEA	Write the program to disk as an ASCII, numbered text file.
SAVEC	Write the program to disk as a prescanned and compressed BASIC program.
SAVEU	Write the program to disk as an unnumbered text file.

**Program Editing:**

<b>AUTO</b>	New line entry with automatic line numbering.
<b>CHANGE</b>	Change text of one or more lines of the program.
<b>COPY</b>	Duplicate lines of the program with new line numbers.
<b>DELETE</b>	Remove lines of the program.
<b>MERGE</b>	Copy a portion of, or all of, another program into the current program.
<b>MERGEU</b>	Copy all of another unnumbered program into the current program.
<b>MODIFY</b>	Perform line editing on one or more lines of the program.
<b>MOVE</b>	Change the position of one or more lines of the program.
<b>OLDMOD</b>	Perform line editing on one or more lines of the program.
<b>RENUMBER</b>	Change the line numbers on one or more lines.

**Program Display and Positioning:**

<b>BOTTOM</b>	Position to last line of program.
<b>DOWN</b>	Position to the next line of the program.
<b>LIST</b>	Display a portion of, or all of, the program.
<b>LOCATE</b>	Find a line containing specified text.
<b>LPLIST</b>	Display a portion of, or all of, the program on the printer.
<b>LPXREF</b>	Display the cross reference listing on the printer.
<b>TOP</b>	Position to the first line of the program.
<b>UP</b>	Position to previous line of the program.
<b>VIEW</b>	Specify view modes for program listings and variable displays.
<b>XREF</b>	Display the cross reference listing on the console.

**Program Debugging:**

<b>BREAK</b>	Set debugging breakpoints
<b>CONTINUE</b>	Resume execution after a <b>STOP</b> statement or a break-point.
<b>SHOWSTACK</b>	Display list of lines and statements showing the path to the current statement being executed.
<b>STEP</b>	Single step through execution of the program but not user defined functions or subprograms.
<b>TRACE</b>	Enable line and variable change tracing.
<b>UNBREAK</b>	Disable breakpoints.
<b>UNTRACE</b>	Disable line and variable change tracing.
<b>VAR</b>	Display the current value of a variable or variables.

Note: Any of the statements described in the chapter on “Functions and Statements” can be executed in immediate mode, thus acting like a command.

## ■ THEOS MultiUser BASIC Program Editor

As described earlier, MultiUser BASIC is a BASIC language compiler and interpreter. A major function of the interpreter is the creation and maintenance of programs. Program editing is accomplished by using commands to enter, modify, display, and save program statements.

MultiUser BASIC is unique in that it is the only THEOS language product that includes its own, built-in, intelligent editor. It is an intelligent editor because it does not just accept lines of text and save them in a file, as a line or screen editor would do. It “knows” the proper syntax for each statement and it “knows” the keywords that are reserved for those statements.

## ■ Using Other Text Editors

Any other text editor can be used to create MultiUser BASIC source programs. For instance, WindoWriter is a good editor that provides many features useful for program editing such as: full screen, multiple windows, cut and paste operations, etc. The principal disadvantage of not using the MultiUser BASIC editor is that other text editors have no knowledge about the syntax acceptable to MultiUser BASIC.

When another text editor is used to create and maintain source programs the text can be entered without line number and in mixed case. When lines are created without line numbers, the entire source program must have no line numbers.

A program created or maintained by a text editor is loaded into the interpreter just like a program created or maintained by the MultiUser BASIC editor. It will take additional time to load the program because complete syntax analysis must be performed on the file as it is loaded. If any errors are found MultiUser BASIC automatically enters the **MODIFY** command for the suspect line, allowing you to make corrections.

As MultiUser BASIC loads a source program it adds line numbers and changes all key words, line labels, function names and subprogram names to upper case.

When you use another text editor to maintain your program source, do not use the interpreter's **SAVEC** command as it creates a non-editable file. You must use the **SAVEA** or **SAVEU** commands to save the program as an editable file usable by text editors.


Text files created by other text editors can be compiled without loading them into the interpreter first. Merely specify the text file as the name of




the program to compile. Unnumbered text files are assigned line numbers by the compiler and syntax analysis is performed on the file. Since you are not in the interpreter you cannot make changes to correct any errors.




## ■ Using the MultiUser BASIC Interpreter


When MultiUser BASIC is invoked, it enters its *command mode* indicated by the MultiUser BASIC prompt character '-'.

```
>basic32 
-
```

Whenever MultiUser BASIC displays the prompt character with the cursor next to it, MultiUser BASIC is waiting for a command.

Commands are entered by typing one of the commands described on the following pages, terminated by pressing .

```
>basic32 
-load custmnt 
-run 
```

To make the examples more readable, the  symbol will not be shown, but it must be pressed at the end of each command.

## ■ Command Text Delimiters

Many MultiUser BASIC commands include text specifications. For instance, the LOCATE and CHANGE commands need text specified. The start and end of the text must be identified with a *delimiter*. A delimiter is any character used for marking the start or end of some entity.

The delimiters that may be used include most of the non-alphabetic, printable characters that are not part of the text specified. The possible delimiters include:

!	%	*	.	<	\	'	~
"	&	+	/	>	]	{	
#	'	,	;	?	^		
\$	)	-	;	@	_	}	

When the single or double quotation characters are used for delimiters the enclosed text may be entered as a mixed-case string. However, due to the rules used by MultiUser BASIC the CHANGE command cannot specify that text is to be changed to lower case:

```
-CHANGE "SOME TEXT"some text"      Cannot enter this command  
-CHANGE "some text"SOME TEXT"
```

The same character must be used for all instances of a delimiter in a command. For instance, do not use the open brace for the first delimiter and the close brace for the second. For consistency, this manual always uses the slant character (/).

---

## AUTO Command

AUTO enables line input mode with automatic line numbering.

- 1 **AUTO** *start-line line-increment*
  - 2 **AUTO** *start-line*
  - 3 **AUTO**

### Operation

**Mode 1**—Line input begins with the first line number equal to *start-line* and with subsequent lines incremented by *line-increment*. The default increment value for subsequent AUTO commands is set to the *line-increment* value.

**Mode 2**—Line input begins with the first line number equal to *start-line* and with subsequent lines incremented by the current increment value. If AUTO command, [Mode 1](#), has not been used prior to this, the default increment value is 10.

**Mode 3**—Line input begins with the first line number equal to the current line number plus the current increment value. If AUTO command, [Mode 1](#), has not been used prior to this, the default increment value is 10.

### Notes

To exit from line input mode, enter an empty line. To enter a blank, numbered line into the program you must enter at least one space character.

### Restrictions

AUTO can be used to add lines to the current source program only, not for any included files.

AUTO cannot be used on *read-protected* or *write-protected* programs.

AUTO cannot replace lines. When the line number used by the automatic line input mode is the same as an existing line in the program, AUTO is exited with the message: "Auto cannot replace or merge lines."

AUTO cannot merge lines. For example, if you have lines numbered 10, 20, 30, *etc.* and then attempt to perform an AUTO 15,10, you will be allowed to enter line 15, however the increment to line 25 will not be allowed and AUTO is exited with the message: "Auto cannot replace or merge lines."

## Examples

The first **AUTO** command uses the default start and increment of 10.

Notice lines 20, 40, and 60. A space is entered prior to the **Enter ↵** to avoid exiting auto entry mode.

The second **AUTO** begins line entry with line 1000 and an increment of five. A space is entered prior to the **Enter ↵** on lines 1005 and 1015.

-NEW

-AUTO

```
10 REM Program: SAMPLE Enter ↵
20 Space Enter ↵
30 REM Version 1.0 02/14/93 Enter ↵
40 Space Enter ↵
50 REM Programmer: Susan B. Doe Enter ↵
60 Space Enter ↵
70 Enter ↵
```

-AUTO 1000 5

```
1000 GOSUB SETUP Enter ↵
1005 Space Enter ↵
1010 WINDOW SELECT 1
1015 Space Enter ↵
1020 Enter ↵
```

---

## BOTTOM Command

BOTTOM positions to the last line of the program.

1 **BOTTOM**

2 **End**

**Operation**      The last line in the program becomes the current line and is displayed. Entry of the **End** key is a shortcut: it is synonymous with entering BOTTOM **Enter ↵**.

**Restrictions**    BOTTOM cannot be used on *read-protected* programs.

**See also**          [DOWN](#), [TOP](#), [UP](#)

**Examples**          -LIST

*Using the program that was started in the AUTO example, the BOTTOM command is used to position to the end and resume entry of lines.*

```
10 REM Program: SAMPLE
20
30 REM Version 1.0 07/04/95
40
50 REM Programmer: Susan K. Doe
60
1000      GOSUB SETUP
1005
1010      WINDOW SELECT 1
1015
-BOTTOM
1015
-AUTO
1025
```

---

## BREAK Command

BREAK sets or clears *debugging breakpoints* in the program.

- 1 **BREAK** [AT] *line-number*
- 2 **BREAK** [AT] *line-number* [AFTER] *count*
- 3 **BREAK AT** *line-label*
- 4 **BREAK AT** *line-label* [AFTER] *count*
- 5 **BREAK** [ON] *variable*
- 6 **BREAK** [ON] *variable* CHANGED
- 7 **BREAK** [ON] *variable* [AFTER] *count*
- 8 **BREAK** [ON] *variable* CHANGED [AFTER] *count*
- 9 **BREAK** [ON] *relational-expression*
- 10 **BREAK**

**Operation**      A *breakpoint* is set, either at a specific line, or on a variable. A breakpoint is a debugging aid that causes program execution to be interrupted when the specified conditions are met. The BREAK statement specifies those conditions.

**Mode 1**—A breakpoint is set at *line-number*. A break will be performed when execution of the program gets to that line, but prior to executing the statements on the line.

**Mode 2**—A breakpoint is set at *line-number*. A break will be performed after the statements on the line have been executed *count* times. Program execution is interrupted prior to executing the line *count*+1 times.

**Mode 3**—Identical to [Mode 1](#) except the line is referenced by *line-label*. Note that the keyword AT must be included (otherwise the *line-label* will be interpreted as a variable name, as in [Mode 5](#)).

**Mode 4**—Identical to [Mode 2](#) except the line is referenced by *line-label*.

**Mode 5**—A break is performed the next time a statement is executed that uses *variable*. The break is performed after the statement using *variable* executes.

**Mode 6**—A break is performed the next time a statement is executed that changes the value of *variable*. The break is performed after the statement executes.

**Mode 7**—A break is performed after *count* statements have been executed that use *variable*. Statements that use *variable* multiple times are counted as one. The break is performed after the statement executes.

**Mode 8**—A break is performed after *count* statements have been executed that change the value of *variable*. Statements that use but do not change the value of *variable* are not counted. The break is performed after the statement executes.

**Mode 9**—A break is performed after a statement is executed that causes the *relational-expression* to be true. The *relational-expression* must be a simple expression with a variable name on the left, a relational operator such as =, <, >, <>, etc., and a constant expression on the right.

**Mode 10**—All of the currently set breakpoints are displayed.

## Notes

Whether the break occurs before or after the event depends upon the type of breakpoint:

- ▶ Breakpoints at line numbers or labels cause a break **before** the line is executed.
- ▶ Breakpoints on variables cause a break **after** a statement executes.

When using the AFTER clause of the BREAK statement, the *count* value is decremented each time that a statement or line is executed that meets the criteria. This will be noticed if another breakpoint or a Break, C causes the program to be interrupted and **Mode 10** of the BREAK statement is used. Additionally, when the “break after” conditions are met the *count* will be zero causing a break after execution of the next statement that uses or changes the variable.

RENUMBER rennumbers the BREAK AT line references.

When a break is performed (caused by any break conditions set or entry of Break, C) the message “Break at line *nnnnnn*.” is displayed.

**Include Files:** To specify a breakpoint in a subordinate include file use the *line-number* syntax of:

*main-program-line: include-file-line*

*main-program-line: include-file<sub>1</sub>-line: include-file<sub>2</sub>-line*

*etc.*

Similarly, line labels in the main source program or include file can be used to reference a breakpoint:

*main-program-label: include-file-label*

**Subprograms:** To specify that a breakpoint is to be set for a LOCAL or STATIC variable used in a subprogram or a user-defined function, use the following syntax for *variable*:

*variable-name* **IN** *subprogram-name*

*variable-name* **IN** *function-name*

**Defaults** When a program is first loaded with the LOAD or RUN commands, all breakpoints are cleared.

**Restrictions** Only one breakpoint may be set for a specific line-reference and only one breakpoint may be set for a specific variable. When a second breakpoint is set for a line or variable that already has a breakpoint set, the prior breakpoint is cleared.

A breakpoint set for a LOCAL or STATIC variable will apply to all instances of that local variable name. For instance, if two subprograms use the local variable I% and a breakpoint is set for:

```
-BREAK ON I% IN SUBPROGRAM1 AFTER 3
```

the program will break the third time that the local variable I% is referenced, even if the references are done in the other subprogram.

Loading, linking or chaining to another program, or running another program, causes all breakpoints to be cleared.

**See also** [CONTINUE](#), [SSTEP](#), [STEP](#), [UNBREAK](#)



## Examples

-LOAD CUSTMNT

*An existing program is loaded and six breakpoints are set.*

```
-B AT END.RUN
-B AT 10000:200 AFTER 5
-B AT SEARCH.TRAP AFTER 2
-B AT EXIT
-B EVENT.FLAG% CHANGED
-B QUIT.WIN% AFTER 2
-B END.RUN% = -1
-B ON I% IN FN.SAMPLE CHANGED AFTER 3
-B
Break on I% in FN.SAMPLE changed after 3
Break on END.RUN% = -1
Break on QUIT.WIN% after 2
Break on EVENT.FLAG% changed
Break at 802230(EXIT)
Break at 803000(SEARCH.TRAP) after 2
Break at 10000:200 after 5
Break at 1130(END.RUN)
-
```

*Notice that the BREAK AT line-label commands are converted and maintained as BREAK AT line-number with the line-label displayed in parentheses.*

---

## CHANGE Command

CHANGE performs global changes to the source program.

- 1 **CHANGE** / *from-text* / *to-text* / *line-range*
- 2 **CHANGE** / *from-text* / *to-text* / *line-number*
- 3 **CHANGE** / *from-text* / *to-text* /
- 4 **CHANGE**

<b>Operation</b>	<b>Mode 1</b> —All occurrences of <i>from-text</i> are changed to <i>to-text</i> for all of the lines in the <i>line-range</i> .
	<b>Mode 2</b> —All occurrences of <i>from-text</i> are changed to <i>to-text</i> for the indicated line only.
	<b>Mode 3</b> —All occurrences of <i>from-text</i> are changed to <i>to-text</i> for the current line only.
	<b>Mode 4</b> —Performs the last change specified on the current line only.
<b>Notes</b>	The ‘ / ’ character shown in the above syntax may be any single character that is not alphanumeric and not included in the <i>from-text</i> or <i>to-text</i> text.
	When a line is changed by this command, the modified line is displayed on the console.
<b>Caution</b>	The CHANGE command operates on characters, not tokens. A change of x to X% for the entire program will probably change many things that you did not intend (for example, the X in the keyword NEXT).
<b>Restrictions</b>	You cannot change uppercase characters to lowercase characters with this command. The MODIFY command must be used.
	Only the current source program can be changed, not any included files.
	CHANGE cannot be used on <i>read-protected</i> or <i>write-protected</i> programs.
<b>See also</b>	<a href="#">MODIFY</a> , <a href="#">OLDMOD</a>

## Examples

<i>An existing program is loaded and all instances of the variable name END.RUN% is changed to END.OF.RUN%.</i>	-LOAD CUSTMNT -C /END.RUN%/END.OF.RUN%/ 1 999999 1040 WHILE NOT END.OF.RUN% 1080 IF NOT END.OF.RUN% THEN GOSUB ENTER.DATA 1090 IF NOT END.OF.RUN% THEN GOSUB MODIFICATIONS 900080 END.OF.RUN% = FALSE% -TOP
<i>The first instance of FLD% is located.</i>	-L /FLD% 10050 FLD% = 1
<i>FLD% is changed to FIELD% for this line only.</i>	-CHANGE /FLD%/FIELD%/ 10050 FIELD% = 1
<i>The LOCATE is repeated.</i>	- <span style="border: 1px solid black; padding: 0 2px;">Again</span> 10090 KEY\$ = FN.INPUT\$(REC\$(FLD%), INX\$(FLD%), INX\$(FLD%), INL(FLD%), INCASE\$(FLD%), INPROMPT\$(FLD%), " ", INHELP\$(FLD%), INSPECIAL\$(FLD%), DUP\$(FLD%))
<i>The CHANGE is repeated for this line.</i>	-C 10090 KEY\$ = FN.INPUT\$(REC\$(FIELD%), INX\$(FIELD%), INX\$(FIELD%), INL(FIELD%), INCASE\$(FIELD%), INPROMPT\$(FIELD%), " ", INHELP\$(FIELD%), INSPECIAL\$(FIELD%), DUP\$(FIELD%)) -

---

# COMPILE Command

The COMPILE command compiles the program currently in memory.

COMPILE

Commands

Operation	The program in memory is saved to disk (unless no changes have been made since the program was last saved) and compiled.
Notes	<p>The compiled program is saved in the current command library or as a flat file with a file type of <code>COMMAND</code>. Command libraries are defined by the account environment or by the <code>SET</code> command. Refer to the <i>System Reference Manual</i>.</p> <p>When <code>COMPILE</code> saves the program it saves it using the <code>SAVEC</code> format. That is, it saves it as a non-text, non-editable file. To ensure that a program is saved in editable format you must <code>SAVE</code> the program prior to compiling.</p>
Restrictions	There must be a program in memory and that program must have a name. When either of these conditions is not met, the message “No program name to compile.” is displayed.
See also	<a href="#">SAVE</a> , <a href="#">SAVEC</a> , <a href="#">SAVEU</a>
Examples	<pre>-NEW -AUTO   10PRINT "Account = ";SYS.ENV\$(1);" (" ;SYS.ENV\$(2);)"   20PRINT "Logon at ";SYS.ENV\$(12)   30PRINT   40PRINT "Number of consoles: ";SYS.ENV\$(15)   50   60END   70 -<b>COMPILE</b> No program name to compile. -NAME SAMPLE -<b>COMPILE</b> "SAMPLE.BASIC:S" saved. Compile: "SAMPLE.BASIC:S" -</pre>

# CONTINUE Command

CONTINUE resumes execution of a program that was interrupted by a breakpoint, STOP statement, or a *program-cancel* (`(Break),C` or `(Break),Q`).

CONTINUE

Operation	Program execution resumes with the statement following the breakpoint.
Notes	When there has been no breakpoint encountered (for example, after a program load) the CONTINUE command acts like a RUN command.
Version 2.0:	This version of MultiUser BASIC will continue with the next statement even when the breakpoint was inside a user-defined function definition or subprogram.
See also	<a href="#">BREAK</a> , <a href="#">SSTEP</a> , <a href="#">STEP</a>

Examples

*An existing program is loaded and displayed to find a good place for a breakpoint.*

*Line 1120 is chosen for the breakpoint, after input is performed.*

*The breakpoint is encountered and displayed. Program execution is continued until the breakpoint is encountered again.*

```
-LOAD HELPTEST
-LIST

10 REM Program HELPTEST simple input with help
20
30 REM Version: 1.0          02/15/93
40
1000 GOSUB SETUP
1040 GENERAL.HELP$ = "GENERAL"
1050
1060 GET.INPUT:
1070
1080 PRINT AT$(5,3);"Field: ";
1100 FIELD1$ = FN.INPUT$(" ",12,3,8,"U","field1")
1110
1120 GOTO GET.INPUT
1130
1140 END.RUN:
~
-B 1120
-RUN
Field: SAMPLE
Break at line 1120
-CONTINUE
Field:
```

---

## COPY Command

COPY duplicates a series of lines in the program to a new location, giving the copied lines new line numbers.

**COPY** *line-range to-line* [*line-increment*]

### Commands

**Operation**      A copy of the *line-range* lines of the program is made and inserted into the program, renumbering the inserted lines starting with *to-line* with an increment of *line-increment*.

**Notes**            When the *line-range* of lines contains references to line numbers within itself, the copied set of lines are adjusted to refer to the new line number in the copy. The original lines are unmodified.

Line labels defined in the *line-range* of lines are not changed during the copy. When there are line labels defined in the copied section, the result will be duplicate label definitions. This error is not detected until the program is executed or compiled.

**Restrictions**    COPY can be used to copy lines to the current source program only, not to any included files.

COPY cannot be used on *read-protected* or *write-protected* programs.

COPY cannot replace lines. If any of the copied lines would have a line number that is the same as an existing line in the program, none of the lines are copied and the message “Line number range error.” is displayed.

COPY cannot merge lines. For example, a program has lines numbered 10, 20, 30, etc. If an attempt is made to copy two lines with COPY 100,110,15,10, none of the lines are copied. The message “Line number range error.” is displayed.

**See also**          [MERGE](#), [MOVE](#), [RENUMBER](#)

## Examples

-LIST 23000 23999

*To create a new validation routine for integer numbers, this routine is copied and placed immediately after the dollar validation routine.*

```
23000 VALIDATE.AMOUNT:
23010
23020     SELECT
23030         CASE NOT NBR(FIELD$)
23040             VALID% = FALSE%
23050             P2MSG$ = "NUMERIC VALUE REQUIRED"
23060         CASE VAL(FIELD$)<>ROUND(VAL(FIELD$),2)
23070             VALID% = FALSE%
23080             P2MSG$ = "FRACTIONAL PENNIES NOT ALLOWED"
23090         CASE VAL(FIELD$)>="10^(INL%(FLD$)-3)
23100             VALID% = FALSE%
23110             P2MSG$ = "DOLLAR VALUE TOO LARGE"
23120     CEND
23130
23140     RETURN
23150
```

*To avoid having duplicate labels the label of the new routine is changed and the routine is listed.*

```
-COPY 23000 23150 23200
-C /AMOUNT/NUMBER/23200
23200 VALIDATE.NUMBER:
-LIST 23200 23999
23200 VALIDATE.NUMBER:
23210
23220     SELECT
23230         CASE NOT NBR(FIELD$)
23240             VALID% = FALSE%
23250             P2MSG$ = "NUMERIC VALUE REQUIRED"
23260         CASE VAL(FIELD$)<>ROUND(VAL(FIELD$),2)
23270             VALID% = FALSE%
23280             P2MSG$ = "FRACTIONAL PENNIES NOT ALLOWED"
23290         CASE VAL(FIELD$)>="10^(INL%(FLD$)-3)
23300             VALID% = FALSE%
23310             P2MSG$ = "DOLLAR VALUE TOO LARGE"
23320     CEND
23330
23340     RETURN
23350
```

*Deletions and changes are made to transform the dollar validation into a number validation routine.*

```
-DELETE 23260 23280
23290     CASE VAL(FIELD$)>="10^(INL%(FLD$)-3)
-C /-3//
23290     CASE VAL(FIELD$)>="10^(INL%(FLD$))
-C /DOLLAR VALUE/NUMBER/23310
23310     P2MSG$ = "NUMBER TOO LARGE"
-
```

---

## DELETE Command

DELETE removes a line or series of lines from a program.

- 1 **DELETE** *line-range*
- 2 **DELETE** \*
- 3 **DELETE**

Commands

Operation	<b>Mode 1</b> —All lines in the <i>line-range</i> are deleted. Specifying a single line number instead of a range, deletes that single line.
	<b>Mode 2</b> —Deletes all lines from the current line to the end of the program.
	<b>Mode 3</b> —The current line is deleted.
Notes	The current line pointer is set to the line following the line or lines deleted.
Restrictions	DELETE can delete lines from the <u>current source program</u> only, not from any included files.
	DELETE cannot be used on <i>read-protected</i> or <i>write-protected</i> programs.
See also	<a href="#">NEW</a>



Examples

*The portion of the program that will be modified is displayed.*

```
-LIST 20180 20290

20180      CASE "STATE"
20190          ON KEY(F2%) GOTO SEARCH.TRAP
20200      CASE "ZIP"
20210          FIELD$ = FN.DIGIT.STRIP$(FIELD$)
20220          DFLD$ = FN.DIGIT.STRIP$(DFLD$)
20230      CASE "PHONE"
20240          FIELD$ = FN.DIGIT.STRIP$(FIELD$)
20250          DFLD$ = FN.DIGIT.STRIP$(DFLD$)
20260      CASE "DATE"
20270          FIELD$ = FN.DIGIT.STRIP$(FIELD$)
20280          DFLD$ = FN.DIGIT.STRIP$(DFLD$)
20290      CEND
-DELETE 20200 20250
```

*Six lines are deleted from the program and the section is listed again to confirm the action.*

```
20260      CASE "DATE"
-LIST 20150 20290
20180      CASE "STATE"
20190          ON KEY(F2%) GOTO SEARCH.TRAP
20260      CASE "DATE"
20270          FIELD$ = FN.DIGIT.STRIP$(FIELD$)
20280          DFLD$ = FN.DIGIT.STRIP$(DFLD$)
20290      CEND
-
```

---

# DOWN Command

DOWN positions to the next line in the program.

Enter ↵

↓

Commands

Operation	The current line is set to the next line of the program and displayed.
Notes	When positioned to the last line of the program, a down command displays: “End of file.”
Restrictions	DOWN cannot be used on <i>read-protected</i> programs.
See also	<a href="#">BOTTOM</a> , <a href="#">TOP</a> , <a href="#">UP</a>

**Example**

```
- 1000
    1000  GOSUB SETUP
- Enter ↵
    1010
- Enter ↵
    1020  WINDOW SELECT 1
- ↓
    1030
-
```

# HELP Command

HELP displays the help text for commands, statements, and functions.

- 1

HELP *keyword*
- 2

HELP
- 3

Help or Ctrl+F1

**Operation**

**Mode 1**—The help text associated with the command, statement or function that starts with *keyword* is displayed on the screen.

**Mode 2**—The help text showing the syntax of all commands, statements and functions is displayed on the screen.

Pressing the key defined as HELP (normally F1) displays the [Mode 2](#) help.

**Notes**

A multiple-page help display can be canceled by pressing Break, C or the quit key (normally F9).

The *keyword* may be a partial keyword. For example, HELP OP would display the help for all commands, statements and functions that start with the letters “OP” which includes the OPEN and the OPTION statements.

**Examples**

```
-HELP
                                Commands
AUTO  [<start>[,<line-increment>]]
BOTTOM
BREAK
BREAK [AT] <line-number> [[AFTER] <count>]
BREAK AT <line-label> [[AFTER] <count>]
BREAK [ON] <variable> [[CHANGED] [[AFTER] <count>]]
BREAK [ON] <variable> <relational operator> <value>
CHANGE [/<from>/<to>[/<linerange>]] | <line-number>]]
COMPILE
CONTINUE
COPY <line-range> <to-line>[ <line-increment>]
DELETE [<line-range> | *]
HELP [<command> | <statement> | <function>]
INDENT [<count>]
LENGTH
LIST [<line-range>]
LOAD <program-name>
LOADU <program-name>
LOCATE [/string[/<line-range> | <line-number>]]
```

---

## INDENT Command

INDENT performs automatic indentation of all lines in the program depending upon the program structure.

```
1  _INDENT  count
2  _INDENT
```

Commands

**Operation**      **Mode 1**—The entire program is reformatted with tab stops every *count* columns.

**Mode 2**—The entire program is reformatted with tab stops of five.

**Notes**            The rules for program indentation are:

1. Initial line indent is set to one tab.
2. If a line contains any of the following statements the indicated indentation is performed:

Statement	Current Line	Lines Following
<i>line-label</i>	Outdented	Indented
CASE	Outdented	Indented
CEND	Outdented	Outdented
DEF	Outdented	Indented
ELSE	Outdented	Indented
FNEND		Outdented
FOR		Indented
IF (multi-line)		Indented
IFEND		Outdented
NEXT		Outdented
OTHERWISE	Outdented	Indented
REM (starting only)	Outdented	Indented
SELECT		Double indented
SUB	Outdented	Indented
WEND		Outdented
WHILE		Indented

Multiple statement lines are analyzed for each of the statements in the line.

3. All other statements do not affect the indentation of the program.

Indentation is performed on the existing lines in the program. New lines added to the program are not automatically indented unless the `INDENT` command is used again.

An indent *count* of 0 may be used to remove all indentation from a program.

#### Defaults

The default indent tab is 5 columns.

#### Restrictions

The maximum indent *count* is 10. Values larger than 10 are ignored.

Should any indent amount for a line extend beyond column *line-length-40*, the indent spacing is changed to 1 for indent tabs beyond that column.

Should any indent amount for a line extend beyond column *line-length-10*, the indent spacing is changed to 0 for indent tabs beyond that column.

Only the current source program is indented, not any included source programs.

## Examples

*A new program is entered.*

*Note: This program is a simple routine that displays all of the color combinations available. It is useful for picking colors that contrast well with each other.*

```
-AUTO
10 COLOR 7,0
20 PRINT CLS$
30
40 FOR BG% = 0 TO 7
50 FOR FG% = 0 TO 7
60 COLOR FG%,BG%,BG%,FG%
70 IF BG%<4
80 PRINT AT(1+BG%*19,FG%+2);
90 ELSE PRINT AT(1+(BG%-4)*19,FG%+12);
100 FEND
110 PRINT " TEST ";CRT("RVON");" TEST ";CRT("RVOFF");
120 NEXT
130 NEXT
140
150 PRINT AT(1,22);
160 COLOR 7,0,0,7
170
```

*Standard indentation is performed on the program and it is listed.*

*Notice how the structure of the program comes out and how easy it is to see which statements are subordinate to others.*

```
-IND
-LIST
10      COLOR 7,0
20      PRINT CLS$
30
40      FOR BG% = 0 TO 7
50          FOR FG% = 0 TO 7
60              COLOR FG%,BG%,BG%,FG%
70              IF BG%<4
80                  PRINT AT$(1+BG%*19,FG%+2);
90                  ELSE PRINT AT$(1+(BG%-4)*19,FG%+12);
100             IFEND
110         PRINT " TEST ";CRT$("RVON");" TEST ";
            CRT$("RVOFF");
120     NEXT
130     NEXT
140
150     PRINT AT$(1,22);
160     COLOR 7,0,0,7
-
```

---

# LENGTH Command

LENGTH displays the length of the program source, included files and the length of the various data types.

LENGTH

**Operation**      The current length of the source program is shown along with the length of all of the included source programs. The amount of space currently used by data is shown along with the amount of free space remaining.

**Notes**            The size of included programs is only determined and shown after the program has been RUN during this session. The size of the data space used is updated at this same time so the initial display of the LENGTH command when a program is first loaded may not be correct.

                      The size of the free data space is misleading because the THEOS operating system will provide additional space for the program when it needs it.

## Examples

*An existing program is loaded and the initial length is displayed.*

```
-LOAD TEST
-LENGTH
Main program size:      3149
Included program size:  0
Total data size:       196610
Free data size: 23156
```

*The sizes change after the program is RUN one time because it is only then that the interpreter knows the size of the included programs and the data space used by those included programs.*

```
-RUN
Stop at line 420.
-LEN
Main program size:      3149
Included program size:  53360
Total data size:       344066
Free data size:       19340
-
```

---

## LIST Command

LIST displays the program on the console.

1 **LIST** *line-range*

2 **LIST**

3 *line-number*

**Operation**      **Mode 1**—Displays the *line-range* portion of the program on the screen. A single line number may be specified causing that line to be displayed.

**Mode 2**—Displays the entire program on the screen. A page heading is used at the top of each page of the listed program.

**Mode 3**—Displays the specified line on the screen.

**Notes**            The current line is always set to the last line displayed.

Page waits are performed at the bottom of each full screen of listing.

A long LIST display can be canceled with the **Quit** key or a **Break**, **C**.

When VIEW NOINCLUDE is set, LIST displays lines in the current source program only, not for any included files; when VIEW INCLUDE is set, LIST displays entire included files if a line in the current source program is listed that contains an INCLUDE statement.

**Restrictions**    LIST cannot be used on *read-protected* programs.

The *line-range* of **Mode 1** or the *line-number* of **Mode 3** refer only to line numbers in the current source program.

When VIEW INCLUDE is set and the program contains an endless, recursive INCLUDE, the listing is aborted with the message "INCLUDE file name "program" creates infinite include loop at nnnnn."

**See also**        [DOWN](#), [LPLIST](#), [LPXREF](#), [VIEW](#), [XREF](#)



## Examples

```
-LIST 1000 1180
1000      GOSUB SETUP
1010
1020      WINDOW SELECT 1
1030
1040      WHILE NOT END.RUN%
1050
1060      GOSUB GET.KEY
1070
1080      IF NOT END.RUN% THEN GOSUB ENTER.DATA
-
```

*The current VIEW settings indicate that INCLUDE files are to be viewed.*

```
-VIEW
NOLOCAL
INCLUDE
NOELEMENTS
NOCONSTANTS
FILEREFS
TRACELINE
WARNING
SAVE MODE - COMPRESSED
-LOAD BJ
```

*When a line is listed that contains an INCLUDE statement, the included file is displayed with a "+" at the beginning of each line. Included files are only displayed if they have already been loaded. This is normally only done when the program is RUN.*

```
-LIST
~
1170      INCLUDE "colors"! Define color names
+ 10      ! Begin COLORS include file
+ 20
+ 30      BLACK% = 0
+ 40      BLUE% = 1
+ 50      GREEN% = 2
+ 60      CYAN% = 3
+ 70      RED% = 4
+ 80      MAGENTA% = 5
+ 90      YELLOW% = 6
+ 100     WHITE% = 7
+ 110
+ 120     ! End COLORS include file
1180
1190      ON KEY(277) GOTO END.OF.GAME      ! Trap (F9)
1200
1210      GOSUB GET.OPTIONS! Read game settings from disk
~
```

# LOAD Commands

LOAD loads a previously saved program into memory.

**LOAD** *program-name*

**LOADU** *program-name*

Commands

**Operation** The program is loaded into memory, replacing any existing program.

**Notes** The LOADU command is a synonym to the LOAD command.

When there has been a change made to the program in memory, then, prior to clearing memory, a question is asked: "OK to forget changes? (Y/N)". You must respond with a 'Y' or an 'N'.

Program name specified:	A	A.B	A.B.C
Program loaded if the default library defined is LN,LT	LN,LT.A or A.BASIC	A.B	A.B.C
Default library not defined	A.BASIC	A.B	A.B.C

The program may be a MultiUser BASIC program, a BASIC program, or a program saved with [SAVEA](#) or [SAVEU](#). BASIC programs and [SAVEA/SAVEU](#) format programs will take longer to load as syntax checking and prescanning is performed for each statement. When an error is found the [MODIFY](#) mode is entered, allowing corrections to be made.

As a program is loaded all breakpoints are cleared, the [TRACE](#) mode is reset and all files are closed as if an END statement were executed.

A message is displayed when the program cannot be found. Memory is cleared of any program that was in memory.

Unnumbered source programs saved with the [SAVEU](#) command or created with a text editor such as WindoWriter can be loaded with this command. An unnumbered source program is numbered as it is loaded, assigning line number 10 to the first line and incrementing by 10 for each subsequent line of the program.

**Restrictions** If the loaded program has read-protection, it can be loaded and executed, but none of the program display, modification or save commands will be allowed. A write-protected program may be loaded and executed, but it may not be modified or saved.

**See also** [MERGE](#), [NAME](#), [RUN](#), [SAVEA](#), [SAVEC](#), [SAVEU](#)

---

## LOCATE Command

LOCATE positions to a line in the program that contains a specified sequence of characters.

- 1 **\_LOCATE** / *string* / *line-range*
- 2 **\_LOCATE** / *string* / *line-number*
- 3 **\_LOCATE** / *string* /
- 4 **\_LOCATE**
- 5 **Again** or **Ctrl** + **A**

Commands

### Operation

**Mode 1**—LOCATE positions to a line in the program that contains a specified sequence of characters. The first occurrence of *string* in *line-range* is located and the line becomes the current line.

**Mode 2**—The first occurrence of *string*, starting with *line-number*, is located and the line becomes the current line.

**Mode 3**—The first occurrence of *string*, starting with the current line, is located and that line becomes the current line.

**Mode 4**—The last LOCATE *string* is searched for again, starting with the current line.

**Mode 5**—The **Again** key (normally **F3**) is synonymous with [Mode 4](#).

### Notes

When the string is found, the current line is set to the line containing the string, and that line is displayed. When there are no more occurrences of *string* in the program, the message “String not found.” is displayed and the current line is not changed.

### Restrictions

LOCATE cannot be used on *read-protected* programs.

Only text in the current source program is searched by this command. Included source files are not searched.

### Examples

```
-LOAD CUSTMNT
-L /1992
    30 REM Version 1.0 02/15/1992
-L
String not found.
```

---

## LPLIST Command

LPLIST prints the program listing on the printer.

- 1 **LLIST**
- 2 **LLIST** *line-range*
- 3 **LP*n*LIST**
- 4 **LP*n*LIST** *line-range*

**Operation**      **Mode 1**—Displays the entire program on PRT1.  
**Mode 2**—Displays the *line-range* portion of the program on PRT1.  
**Mode 3**—Displays the entire program on PRT*n*.  
**Mode 4**—Displays the *line-range* portion of the program on PRT*n*.

**Notes**            The current line is always set to the last line printed.

The heading for each page of the listing contains the name and version number of BASIC, the complete file name of the program, the date and time of the listing and the page number.

When the VIEW NOINCLUDE mode is set, LPLIST displays lines in the current source program only, not for any included files; when VIEW INCLUDE mode is set, LPLIST displays entire included files if a line in the current source program is listed that contains an INCLUDE statement.

**Restrictions**    LPLIST cannot be used on *read-protected* programs.

**See also**        [LIST](#), [LPXREF](#), [VIEW](#), [XREF](#)

## Examples

*Assuming that CUSTMNT is the current program in memory, an LPLIST will print the entire program on the primary printer.*

```
10 REM Program: CUSTMNT Customer File Maintenance
20
30 REM Version 1.0 02/01/1998
40
50 REM Programmer: Susan Brown
60
70 REM Not copyrighted
80
1000 GOSUB SETUP
1010
1020 WINDOW SELECT 1, UPDATE ON
~
```

*The current VIEW settings indicate that INCLUDE files are to be viewed.*

```
-VIEW
NOLOCAL
INCLUDE
NOELEMENTS
NOCONSTANTS
FILeref
TRACELINE
WARNING
SAVE MODE - COMPRESSED
```

*When a line is listed that contains an INCLUDE statement, the include dfile is displayed with a "+" at the beginning of each line.*

```
-LOAD BJ
-LPLIST
~
1170 INCLUDE "colors"! Define color names
+ 10 ! Begin COLORS include file
+ 20
+ 30 BLACK% = 0
+ 40 BLUE% = 1
+ 50 GREEN% = 2
+ 60 CYAN% = 3
+ 70 RED% = 4
+ 80 MAGENTA% = 5
+ 90 YELLOW% = 6
+ 100 WHITE% = 7
+ 110
+ 120 ! End COLORS include file
1180
1190 ON KEY(277) GOTO END.OF.GAME ! Trap (F9)
1200
1210 GOSUB GET.OPTIONS ! Read game settings
~
```

---

## LPXREF Command

LPXREF prints a *cross-reference* listing of the program, showing all references to lines, line-labels, variables, and constants.

Commands

- 1 **LPXREF**
- 2 **LPXREF** *line-range*
- 3 **LP<sub>n</sub>XREF**
- 4 **LP<sub>n</sub>XREF** *line-range*

<b>Operation</b>	<b>Mode 1</b> —Displays the cross-reference listing for the program on PRT1.
	<b>Mode 2</b> —Displays the cross-reference listing for the <i>line-range</i> portion of the program on PRT1.
	<b>Mode 3</b> —Displays the cross-reference listing on PRT <sub>n</sub> .
	<b>Mode 4</b> —Displays the cross-reference listing for the <i>line-range</i> portion of the program on PRT <sub>n</sub> .
<b>Notes</b>	Refer to the XREF command for a description of the display of the LPXREF command. The LPXREF command displays identical information with the only difference being that the display is on a printer.
<b>See also</b>	<a href="#">LPLIST</a> , <a href="#">VIEW</a> , <a href="#">XREF</a>

---

## MERGE Command

MERGE and MERGEU copy program lines from another program into the current program.

- 1 **MERGE** *program-name*
- 2 **MERGE** *program-name line-range*
- 3 **MERGE** *program-name line-range to-line [line-increment]*
- 4 **MERGEU** *program-name to-line [line-increment]*

**Operation**      **Mode 1**—A copy of all of the lines in *program-name* is merged into the current program.

**Mode 2**—A copy of the *line-range* lines of *program-name* is made and merged into the current program.

**Mode 3**—A copy of the *line-range* lines of *program-name* is made, renumbered starting with *to-line* with increments of *line-increment* and merged into the current program. Omitting *line-increment* causes the default increment of 10 to be used.

**Mode 4**—The entire set of unnumbered program lines in *program-name* is merged into the current program and given line numbers starting with *to-line* and incrementing by *line-increment*. Omitting *line-increment* causes the default increment of 10 to be used.

**Notes**              Unlike the commands [AUTO](#), [COPY](#), and [MOVE](#), the MERGE and MERGEU commands can merge lines into your program.

**Caution:**          MERGE and MERGEU can replace lines. When any of the merged lines has a line number that is the same as an existing line in the program, the merged line replaces the existing line.

To avoid the problem with MERGE and MERGEU replacing lines try to standardize the line-numbering scheme used in all of your programs. With the six-digit line numbers available, it should be easy to reserve certain ranges of line numbers for specific purposes.

When the [Mode 3](#) form of the MERGE command is used to renumber the merged lines be aware that any line number references in the merged lines are **not renumbered**.

**Restrictions**

MERGE and MERGEU can copy lines into the current source program only, not into any program files included in the current source program.

MERGE and MERGEU cannot be used if the current program or the merged from program is *read-protected*.

MERGE cannot merge an unnumbered text file. Use the MERGEU command instead.

**See also**

[COPY](#) and [MOVE](#) commands, [INCLUDE](#) statement

**Examples**

*A new program is started using sections of an existing program.*

```
- NEW
-MERGE CUSTMNT 800000 802220
-MERGE CUSTMNT 700000 703370
```

*The entire program "STANDARD.SETUP" is merged into the current program at line 9000 and renumbered.*

```
-MERGE STANDARD.SETUP 1 999999 9000 5
```

*A new program is started and the program blocks saved in various files are merged into the new program.*

```
- NEW
-MERGEU FNINPUT 700000
-MERGEU FNPROMPT 710000
-MERGEU GHLPKEY 800000 5
-MERGEU SHELPKEY 801000
-MERGEU QUITKEY 802000
```

```
-
```



---

## MODIFY Command

MODIFY invokes the line editor to allow modifications to be made to a program line.

1 **MODIFY** *line-range*

2 **MODIFY**

**Operation**      **Mode 1**—Allows modification to each line in *line-range*. You may specify a single line number, in which case that single line is modified.

**Mode 2**—Allows modification of the current line.

**Notes**            During modification mode, the line to be modified is displayed and the cursor is positioned to the first character (excluding the line number) of the line. At this point, you can insert characters, delete characters, or change existing characters.

**Restrictions**    MODIFY can only modify lines in the current source program, not in any included source programs.

MODIFY cannot be used on *read-protected* or *write-protected* programs.

**See also**        [CHANGE](#), [OLDMOD](#)

Key	Meaning
<b>Cursor movement</b>	
<b>Ctrl</b> + <b>L</b> or <b>→</b>	Move cursor right one character.
<b>Ctrl</b> + <b>H</b> or <b>←</b>	Move cursor left one character.
<b>Ctrl</b> + <b>G</b> or <b>BegLine</b>	Move to first character of line.
<b>Ctrl</b> + <b>E</b> or <b>EndLine</b>	Move to end of line.
<b>Character searching</b>	
<b>Ctrl</b> + <b>W</b> c or <b>SchFwd</b> c or <b>Ctrl</b> + <b>D</b> c or <b>Find</b> c	Position to next occurrence of character c.
<b>Ctrl</b> + <b>V</b> c or <b>SchBck</b> c	Position to previous occurrence of character c.
<b>Ctrl</b> + <b>A</b> or <b>Repeat</b>	Repeat prior search.
<b>Character deletion</b>	
<b>Ctrl</b> + <b>Z</b> or <b>Delete</b>	Delete the current character.
<b>BackSpace</b> or <b>Del</b>	Delete the prior character.
<b>Ctrl</b> + <b>N</b> or <b>Erase</b>	Delete characters from current position to end of line.
<b>Replacement, insertion</b>	
<b>Ctrl</b> + <b>R</b> or <b>Insert</b>	Toggle between character insert mode and character replacement mode. The initial mode is insert.
<b>Ctrl</b> + <b>C</b> or <b>Case</b>	Toggle the case mode of the current character and advance to the next character. Note, the case mode of the character is only changed if the character is part of a string literal or a REM statement.
<b>Ending modification</b>	
<b>Ctrl</b> + <b>M</b> or <b>Enter</b> <b>↵</b>	End modification of this line.
<b>Break</b> , <b>C</b> or <b>Quit</b>	Exit modification of lines; do not save changes made to this line.

# MOVE Command

MOVE relocates a sequence of program lines in the program to a new location, giving the copied lines new line numbers.

**MOVE** *line-range to-line* [*line-increment*]

**Operation**      The position of the lines in *line-range* is moved so that the lines are inserted into the program at *to-line* position. The moved lines are renumbered with an increment of *line-increment* or the default increment value of 10.

**Notes**            After the move is performed, the current line is set to the line following *line-range*. For example, after moving line 100 through 190 to some new location, the current line would be the line following 190.

**Restrictions**    MOVE can only move lines within the current source program, not from or to any included source programs.

MOVE cannot be used on *read-protected* or *write-protected* programs.

MOVE cannot replace lines. If any of the moved lines would have a line number that is the same as an existing line in the program, none of the lines are moved and the message “Line number range error.” is displayed.

MOVE cannot merge lines. For example, with lines numbered 10, 20, 30, etc., attempt to move two lines with MOVE 100,110,15,10. (This specifies that line 100 is to be moved to line 15 and line 110 is to be moved to line 25.) None of the lines are moved and the message “Line number range error.” is displayed.

**See also**            [COPY](#), [MERGE](#), [RENUMBER](#)

**Examples**            -LOAD CUSTMNT

*First, the current definitions of certain lines in the program are checked.*

```
-12000
12000    READ.NEXT.CUSTOMER:
-13000
13000    READ.PREV.CUSTOMER:
```

*Line 14000 is free.*

```
-14000
20000   ENTER.DATA:
```

*The routine located at line 12000 is moved to 14000.*

```
-MOVE 12000 12999 14000
13000    READ.PREV.CUSTOMER:
```

---

# NAME Command

NAME displays or changes the file name of the program.

- 1

**NAME** *program-name*
- 2

**NAME**

Commands

Operation	<b>Mode 1</b> —The program in memory is given the name <i>program-name</i> .
	<b>Mode 2</b> —The complete name of the current program is displayed.
Notes	The NAME command does not write the program to disk.
	The <i>program-name</i> may be any valid THEOS filename. Although the default file type for MultiUser BASIC programs is BASIC, any file type may be used.
	When a default library is set, specifying a simple name without a file type indicates that the name is a member of the default library.
See also	<a href="#">LOAD</a> , <a href="#">RUN</a> , <a href="#">SAVE</a> , <a href="#">SAVEA</a> , <a href="#">SAVEC</a> , <a href="#">SAVEU</a>

## Examples

<i>The current name of the program is confirmed.</i>	-LOAD CUSTMNT
	- <b>NAME</b>
	EXAMPLE.PROGRAMS.CUSTMENT:S
	-DELETE 1 899999
	-DELETE 980000 999999
<i>Changes are made to the program with the <b>DELETE</b> command and the program is renamed and saved.</i>	- <b>NAME STANDARD.SETUP</b>
	-SAVE
	"STANDARD.SETUP:S" saved.
	-

---

## NEW Command

NEW clears the current program from memory along with its name.

- 1

**NEW**
- 2

**NEW** *program-name*

### Operation

**Mode 1**—The current program in memory is cleared and the NAME is set to an empty string. All data and breakpoints are cleared.

**Mode 2**—Similar to [Mode 1](#) except that the NAME is set to *program-name*. This is identical to performing a NEW followed by a [NAME](#) command.

### Notes

Any files left open by the current program are closed.

All run-time options are set to their defaults.

If the current program has had any changes made to it and the program has not been saved, the message “OK to forget changes? (Y/N)” is displayed and you must respond with a ‘Y’ or ‘N’.

The *program-name* may be any valid THEOS filename. Although the default file type for MultiUser BASIC programs is BASIC, any file type may be used.

When a default library is set, specifying a simple name without a file type indicates that the name is a member of the default library.

### See also

[DELETE](#), [LOAD](#), [NAME](#)

### Examples

*A new program is started. Any previous program is cleared.*

```
-NEW SAMPLE
-AUTO
 10 REM Program: SAMPLE
 20
 30 REM Version 1.0 07/04/95
 40
 50 REM Programmer: Susan B. Doe
 60
 70
-NAME
SAMPLE.BASIC
-
```

---

## OLDMOD Command

OLDMOD invokes the line editor using old style (OASIS or THEOS 8) editing commands to allow modifications to be made to a program line.

- 1 OLDMOD *line-range*
- 2 OLDMOD

### Commands

**Operation**      **Mode 1**—Allows modification to each line in *line-range*. You may specify a single line number, in which case that single line is modified.

**Mode 2**—Allows modification of the current line.

**Notes**            The OLDMOD command is provided as a convenience to programmers who prefer the older style of modifications provided with old versions of THEOS BASIC. This command provides many of the same capabilities as the [MODIFY](#) command. The basic difference is that [MODIFY](#) is a text editor and OLDMOD is like a line editor with a command mode and a text modification mode.

Initially, when OLDMOD is entered, it is in command mode. This mode allows movement around the line, deletion of existing characters, and case changing of existing characters. Character insertion and replacement is not allowed unless the insert or replace mode is used. While in the insert or replace mode the command mode keys to move around the line, change case, etc., cannot be used.

**Restrictions**    OLDMOD can only modify lines in the current source program, not in any included source programs.

OLDMOD cannot be used on *read-protected* or *write-protected* programs.

**See also**        [CHANGE](#), [MODIFY](#)

Key	Meaning
<b>Cursor movement</b>	
<b>[Ctrl]+[L]</b> or <b>[→]</b> or <b>[Space]</b>	Move cursor right one character.
<b>[Ctrl]+[H]</b> or <b>[←]</b>	Move cursor left one character.
<b>[Ctrl]+[G]</b> or <b>[BegLine]</b> or <b>[B]</b>	Move to first character of line.
<b>[Ctrl]+[E]</b> or <b>[EndLine]</b> or <b>[E]</b>	Move to end of line.
<b>Character searching</b>	
<b>[Ctrl]+[W]c</b> or <b>[SchFwd]c</b> or <b>[Ctrl]+[D]c</b> or <b>[Find]c</b> or <b>[F]c</b>	Position to next occurrence of character c.
<b>Character deletion</b>	
<b>[D]</b> or <b>[Ctrl]+[Z]</b> or <b>[Delete]</b>	Delete the current character.
<b>Replacement, insertion</b>	
<b>[R]</b>	Begin replace mode. This mode is continued until <b>[Enter ↵]</b> is pressed. In replace mode the only editing key available is <b>[Ctrl]+[H]</b> or <b>[←]</b> which is a nondestructive backspace.
<b>[I]</b>	Begin insert mode. This mode is continued until <b>[Enter ↵]</b> is pressed. In insert mode the only editing key available is <b>[Ctrl]+[H]</b> or <b>[←]</b> which is a destructive backspace.

Key	Meaning
<b>Changing case</b>	
<b>Ctrl+C</b> or <b>Case</b>	Toggle the case mode of the current character and advance to the next character. The case mode of the character is only changed if the character is part of a string literal.
<b>L</b>	Change the current character to lowercase and advance to the next character. The case mode of the character is only changed if the character is part of a string literal.
<b>U</b>	Change the current character to uppercase and advance to the next character. The case mode of the character is only changed if the character is part of a string literal.
<b>Ending modification</b>	
<b>Ctrl+M</b> or <b>Enter ↵</b>	End modification of this line. If the current mode is insert or replace, that mode is ended and command mode is entered.
<b>Q</b> or <b>Ctrl+Q</b> or <b>Quit</b> or <b>Break</b> , <b>C</b>	Exit modification of lines; does not save changes to this line.



---

## QUIT Command

QUIT exits the interpreter/editor without saving the current program on disk.

- 1 **QUIT**
  - 2 **QUIT** *string-literal*
  - 3 **QUIT** *numeric-literal*

### Operation

**Mode 1**—The current program is exited with a return code of 0. Control is returned to the *calling environment*. The calling environment is the state or program running when MultiUser BASIC was entered. This might be the CSI, an EXEC program, WindoWriter, a compiled BASIC program, etc.

This mode of QUIT can be performed by pressing **Quit**.

**Mode 2**—The current program is exited and the command specified by *string-literal* is executed. *string-literal* may be any valid THEOS command or program name that you have written and compiled. For example,

-QUIT SHOW USERS

exits MultiUser Basic and executes the SHOW utility, giving it the argument USERS. After the command completes execution, control is returned to the *calling environment*.

The *string-literal* must be a literal (unquoted). When the argument is a quoted string the QUIT command is interpreted as the **QUIT** statement. (See differences, below.)

**Mode 3**—The current program is exited with a return code of *numeric-literal*. Control is returned to the *calling environment*.

### Notes

Any files or devices still open at the time of the QUIT are closed first.

Any windows open at the time the QUIT command is executed will be closed. (The **QUIT** statement leaves windows open.)

If there have been any changes made to the program source since the last save was performed, you are asked "OK to forget changes? (Y/N)." You must respond with a Y, N, **Quit**, or **Save** to answer this question.

- Y Exits program without saving changes.
- Quit** Exits program without saving changes.
- N Cancels the QUIT command, returning to the command mode.
- Save** Saves the program (in [SAVEC](#) format) and exits .

The QUIT command and the [QUIT](#) statement operate slightly differently:

	QUIT command	QUIT statement	
		Interpreter	Compiled
Close all files	✓	✓	✓
Clear all non-COMMON data	✓		✓
Clear all COMMON data	✓	✓	✓
Terminate open program structures	✓	✓	✓
Clears the current ON ERROR trap	✓	✓	✓
Clear all ON KEY and ON EVENT traps	✓	✓	✓
Executes a command	Optional	Optional	Optional
Reset all OPTIONS	✓	✓	✓
Close windows	✓		
Deactivate subtasks	N/A	N/A	✓
Exit	✓		✓

**See also** [SAVE](#), [SAVEA](#), [SAVEC](#), [SAVEU](#) commands, [QUIT](#) statement

### Examples

*A QUIT is performed after some changes have been made to the current program.*



**-QUIT**  
OK to forget changes? (Y/N)

*The second QUIT example exits BASIC and compiles a program.*

**-QUIT B32 SAMPLE**


## RECALL Command


The RECALL command recalls a prior command line entered.

- 1 
- 2 **!**
- 3 **!nnn**
- 4 








### Operation

**Mode 1**—The last command line entered is recalled and the cursor is positioned to the first character of that command line. You may now edit or change the command line before executing it. For instance, in the following example a program name is misspelled causing MultiUser BASIC to not find the desired program.

```
-LOAD /PROGRAMS/SOURCE/TESTX
File "/PROGRAMS/SOURCE/TESTX.BASIC" not found.
-
-LOAD /PROGRAMS/SOURCE/TESTX
```

The command is recalled and can be edited to correct the misspelling. After the correction merely press  and the corrected command is executed.

**Mode 2**—The command history is displayed in a list box window and you may select any of the last sixteen commands by using the  and  keys and then pressing .

```
-LOAD /PROGRAMS/SOURCE/TESTX
File "/PROGRAMS/SOURCE/TESTX.BASIC" not found
-LOAD /PROGRAMS/SOURCE/TEST
-INDENT
-VIEW FILEREF
-VIEW INCLUDE
-VIEW TRACEFILE TRACE.OUTPUT
-!
```

```

COMMAND HISTORY
LOAD /PROGRAMS/SOURCE/TESTX
LOAD /PROGRAMS/SOURCE/TEST
INDENT
VIEW FILEREF
VIEW INCLUDE
VIEW TRACEFILE TRACE.OUTPUT

```

**Mode 3**—A prior command is recalled and the cursor is positioned to the first character of that command line allowing you to change the command before executing it.

```

- !3[Enter]
- INDENT

```

**Mode 4**—A shortcut key to invoke the command history window.

### Notes

All of the editing keys described for the **MODIFY** command can be used to modify the command line prior to pressing [Enter].

### Examples

*A program is loaded and an attempt is made to MERGE another program. Due to an error in the line range specified, the program cannot be merged. The command is recalled and the line range is corrected without re-entering the entire line.*

```

-LOAD TEST
-MERGE SAMPLE.SOURCE.MYPROG 1000 999 10000 5
Line number range error
-[Ctrl]+[E]
-MERGE SAMPLE.SOURCE.MYPROG 1000 1999 10000 5

```

*Three more commands are entered and the command history is displayed.*

```

-RENUMBER 1000 100
-INDENT
-VIEW TRACEFILE TRACE.OUTPUT
-[PageUp]

```

```

COMMAND HISTORY
LOAD TESTX
MERGE SAMPLE.SOURCE.MYPROG 1000 999 10000 5
MERGE SAMPLE.SOURCE.MYPROG 1000 1999 10000 5
RENUMBER 1000 100
INDENT
VIEW TRACEFILE TRACE.OUTPUT

```

---

## RENUMBER Command

RENUMBER changes the line numbers of the program without changing their sequence.

- 1 **RENUMBER** *start line-increment line-range*
- 2 **RENUMBER** *start line-increment line-number*
- 3 **RENUMBER** *start line-increment*
- 4 **RENUMBER** *start*
- 5 **RENUMBER**

Commands

### Operation

**Mode 1**—The portion of the program specified by *line-range* is renumbered with the first line numbered *start* and succeeding lines incremented by *line-increment*.

**Mode 2**—The portion of the program from *line-number* to the end of the program is renumbered with the first line numbered *start* and succeeding lines incremented by *line-increment*.

**Mode 3**—The entire program is renumbered with the first line numbered with *start* and succeeding lines incremented by *line-increment*.

**Mode 4**—The entire program is renumbered with the first line numbered with *start* and succeeding lines incremented by the last *line-increment* value used. The default *line-increment* of 10 is used when *line-increment* has not been specified prior to this command.

**Mode 5**—The entire program is renumbered with the first line numbered with the last *line-increment* used and succeeding lines incremented by that amount. The default *line-increment* of 10 is used when *line-increment* has not been specified prior to this command.

### Notes

All references to lines that have been renumbered are updated to reflect the new line numbers. This affects statements ([GOTO](#), [GOSUB](#), [ON EVENT](#), [ON KEY](#), *etc.*) and breakpoints.

**Restrictions**

RENUMBER can only renumber lines in the current source program, not in any included source programs.

RENUMBER cannot be used on *read-protected* or *write-protected* programs.

RENUMBER cannot replace lines. If any of the renumbered lines would have a line number that is the same as an existing line in the program, RENUMBER is exited with the message: "Line number range error."

RENUMBER cannot merge lines. For example, with lines numbered 10, 20, 30, etc. and attempting to perform a RENUMBER 15, 10, 100, 190 renumbering will not be performed and RENUMBER is exited with the message "Line number range error."

Only the current source file is affected by the RENUMBER command. Files included in this source file are unaffected.

**See also**

[COPY](#), [MERGE](#), [MOVE](#)

**Examples**

```
-LIST
10    COLOR 7,0
20    PRINT CLS$
30
40    FOR BG% = 0 TO 7
50        FOR FG% = 0 TO 7
60            COLOR FG%,BG%,BG%,FG%
70            IF BG%<4
80                PRINT AT$(1+BG%*19,FG%+2);
90            ELSE PRINT AT$(1+(BG%-4)*19,FG%+12);
100        IFEND
110        PRINT " TEST ";CRT$("RVON");" TEST ";
            CRT$("RVOFF");
120    NEXT
130    NEXT
140
150    PRINT AT$(1,22);
-RENUMBER 1000 2
-LIST
1000    COLOR 7,0
1002    PRINT CLS$
1004
1006    FOR BG% = 0 TO 7
1008        FOR FG% = 0 TO 7
1010            COLOR FG%,BG%,BG%,FG%
1012            IF BG%<4
1014                PRINT AT$(1+BG%*19,FG%+2);
1016            ELSE PRINT AT$(1+(BG%-4)*19,FG%+12);
~
```

---

## RUN Command

RUN begins execution of a program.

- 1 **RUN** *program-name*
- 2 **RUN** *program-name line-number*
- 3 **RUN**
- 4 **RUN** *line-number*
- 5 **RUN** *program-name token ...*
- 6 **RUN** *\* token ...*

### Operation

**Mode 1**—The program indicated by *program-name* is loaded into memory and executed, starting with the first line in the program.

**Mode 2**—The program indicated by *program-name* is loaded into memory and executed, starting with *line-number*.

**Mode 3**—The program in memory is executed, starting with the first line in the program.

**Mode 4**—The program in memory is executed, starting with *line-number*.

**Mode 5**—The program specified by *program-name* is loaded into memory. The program's command arguments are set to the values of the *token* list and the program is executed, starting with the first line.

**Mode 6**—The program command arguments for the program in memory are set to the values of the *token* list and the program is executed, starting with the first line.

### Notes

Any files open from prior executions of this or other programs are closed before execution begins.

All data items are cleared (except those specified by [Mode 5](#) and [Mode 6](#)) before the program is executed.

All windows are closed before execution begins.

If the program contains any [INCLUDE](#) statements, the current version of the indicated files are loaded into memory.

[Mode 5](#) and [Mode 6](#) are useful when testing a program that needs command line arguments. To set command line arguments to numeric values or lower case text, enclose the arguments in quotation marks.

Prior to executing the program a pre-scan is performed locating and validating all [DIM](#), [COMMON](#), [LOCAL](#), [SHARED](#), and [STATIC](#) statements, all directive statements, and all program structures. Any errors or invalid structures are reported and program execution is not performed.

**Restrictions** The *line-number* in [Mode 2](#) and [Mode 4](#) refer to a line number in the main source program only, not to any line in an included source program.

**See also** [LOAD](#) command, [CHAIN](#), [LINK](#), & [RUN](#) statements, [CMDARG\\$](#) function.

### Examples

*The program APPOINTS is loaded into memory.*

**-RUN APPOINTS SAMPLE**

*CMDARG\$(1) is set to "SAMPLE" and the program is executed.*

*The program in memory is executed.*

**-RUN**

*The program CUSTMNT from the library EXAMPLE.PROGRAMS is loaded into memory and executed.*

**-RUN EXAMPLE.PROGRAMS.CUSTMNT**



---

## SAVE Commands

SAVE writes the current program to disk.

- 1 **SAVE** *program-name*
- 2 **SAVE**
- 3 **F10**
- 4 **SAVEA** *program-name*
- 5 **SAVEA**
- 6 **SAVEC** *program-name*
- 7 **SAVEC**
- 8 **SAVEU** *program-name*
- 9 **SAVEU**

### Operation

The SAVE command saves the current program in the default save format; the SAVEA command saves the program as a numbered ASCII text file; the SAVEC command saves the program as a coded BASIC file; the SAVEU command saves the program as an unnumbered ASCII text file.

**Mode 1**—The name of the current program is set to *program-name* and then written to disk using the current, default save mode.

**Mode 2**—The current program is written to disk with its current name using the current, default save mode.

**Mode 3**—Shortcut key for [Mode 2](#).

**Mode 4**—The name of the current program is set to *program-name* and then written to disk as a numbered, ASCII file. The [VIEW SAVE](#) option is set to VIEW SAVEA and the default save mode is set to SAVEA.

**Mode 5**—The current program is written to disk with its current name as a numbered, ASCII file. The [VIEW SAVE](#) option is set to VIEW SAVEA and the default save mode is set to SAVEA.

**Mode 6**—The name of the current program is set to *program-name* and then written to disk as a coded, BASIC program file. The [VIEW SAVE](#) option is set to VIEW SAVEC and the default save mode is set to SAVEC.

**Mode 7**—The current program is written to disk with its current name as a coded, BASIC program file. The [VIEW SAVE](#) option is set to VIEW SAVEC and the default save mode is set to SAVEC.

**Mode 8**—The name of the current program is set to *program-name* and then written to disk as an unnumbered, ASCII text file. The [VIEW SAVE](#) option is set to VIEW SAVEU and the default save mode is set to SAVEU. See “[Caution](#)” on page 108.

**Mode 9**—The current program is written to disk as an unnumbered, ASCII text file with its current name. The [VIEW SAVE](#) option is set to VIEW SAVEU and the default save mode is set to SAVEU. See “[Caution](#)” on page 108.

## Notes

[Mode 1](#), [Mode 2](#) and [Mode 3](#) of this command save the program in the current, default save mode. The default save mode is determined by the first condition that is true:

1. The last, explicit SAVEA, SAVEC or SAVEU command used since this program was loaded sets the default save mode.
2. If [VIEW SAVEA](#), [VIEW SAVEC](#) or [VIEW SAVEU](#) has been set then that is the default save mode.
3. If no [VIEW SAVE](#) option has been set but the VIEW\_SAVE environment variable is set prior to invoking the interpreter, then that is the default save mode.
4. If no [VIEW SAVE](#) option has been set and no VIEW\_SAVE environment variable is set then the type of program loaded determines the default save mode. The default save mode is SAVEA when the program loaded is a
  - ▶ BASIC386 compressed program file
  - ▶ MultiUser BASIC, Version 1.0 program file
  - ▶ Numbered, ASCII program file

If the program was an unnumbered ASCII program file the default save mode is SAVEU. If the program was a compressed MultiUser BASIC, Version 2.0 program file the default save mode is SAVEC.

The *program-name* may be a complete specification of the name including the file name, file type and file drive, the file name, file type, member name and file drive or it may be a simple file name. The path (directory) for the program may also be specified in the *program-name*.

Backup copies of the prior version of the program are saved. For details about backup files refer to the section on “[Backup Files](#),” below.

After the program is saved, a message displays indicating that the program has been saved and showing the complete file name used.

## Backup Files

Normally, MultiUser BASIC maintains one version of backup file for each source program. If special libraries are created before saving a program with MultiUser BASIC, you can have as many as nine versions of backup to a program. For example, if you create a program and save it, then over time make four changes to the program, saving it each time, it is possible to access any of the five versions of the program (the current version and the four previous versions).

**Single version backups.** Before MultiUser BASIC will maintain multiple backup versions, you must first create the libraries for these different versions. When no special libraries are created, MultiUser BASIC saves the prior version of a program as a file named PROGNAME.BACKUP. The PROGNAME is either the program’s file-name or its member-name, as appropriate.

You can create a backup library to keep all the backups for your programs in one place. This library has a file type of BACKLIB. The file name of the backup library should be the account name. For example, if you have programs in the account SOURCE, then create a library named SOURCE.BACKLIB.

The following table shows the backup file names that MultiUser BASIC chooses for the two basic types of program files (flat files and library member files). When a backup choice is a library and that library does not exist, then MultiUser BASIC goes to its next choice for the backup name.

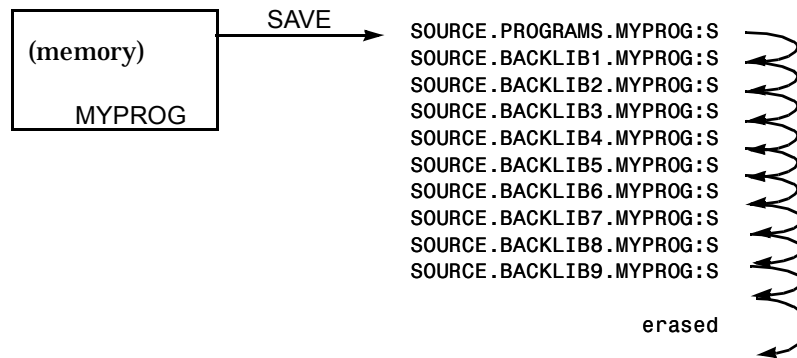
Account:	SOURCE	SOURCE
Program:	SAMPLE.BASIC	ACCNTING.PROGRAMS.SAMPLE
1st choice:	SOURCE.BACKLIB.SAMPLE	SOURCE.BACKLIB.SAMPLE
2nd choice:	SAMPLE.BACKUP	SAMPLE.BACKUP

**Multiple version backups.** To maintain multiple versions of backups for programs, first create special libraries to hold the versions. These special libraries have file names the same as the account name and file types of BACKLIB1, BACKLIB2, etc., through BACKLIB9. Create only as many libraries as backup versions desired. For example, to maintain three levels of backups for a file on the account ACCNTING, create the libraries ACCNTING.BACKLIB1\_, ACCNTING.BACKLIB2, and ACCNTING.\_BACKLIB3. Make sure that there are no libraries with the same file name and a file type of BACKLIB without the ending digit.

The following shows the complete sequence of files that MultiUser BASIC looks for when creating a backup file.

Account:	SOURCE	SOURCE
Program:	SAMPLE.MEMO	ACCNTING.PROGRAMS.SAMPLE
1st choice:	SOURCE.BACKLIB1.SAMPLE	SOURCE.BACKLIB1.SAMPLE
2nd choice:	SOURCE.BACKLIB.SAMPLE	SOURCE.BACKLIB.SAMPLE
3rd choice:	SAMPLE.BACKUP	SAMPLE.BACKUP

When MultiUser BASIC finds its backup choice and that choice is a library with filetype BACKLIB1, then the backup file found in that library is moved to the next numbered backup library. The previous backup file in that library is moved to the next library, and so on.



### Caution

When a program is saved in SAVEU format (unnumbered ASCII), any line number references in the file (for example, GOTO 1000), are not changed by this command. When the program is subsequently loaded or merged into another program, the line number reference will not be adjusted. Therefore, **DO NOT PERFORM A SAVEU ON A PROGRAM THAT HAS ANY LINE NUMBER REFERENCES.** Change them to line label references first.

**Restrictions** [Mode 2](#), [Mode 5](#), [Mode 7](#) and [Mode 9](#) cannot be used if the current program does not have a name. The message “File name missing.” is displayed.

As with any file in THEOS, if the *program-name* already exists on disk and is erase or write protected, an error occurs and the program is not saved.

Only the current source file is saved. Source programs included in this program with the [INCLUDE](#) statement are not saved.

Programs saved in SAVEC format (coded BASIC) cannot be edited with Edit or WindoWriter.

**See also** [NAME](#), [VIEW](#)

**Examples**

```
-LOAD CUSTMNT
-SAVE NEWPROG
"EXAMPLE.PROGRAMS.NEWPROG:S" saved.
-DELETE 1 899999
-SAVE STANDARD.SETUP
"STANDARD.SETUP:S" saved
```

---

## SHOWSTACK Command

SHOWSTACK displays the return location of the currently executing subprogram, function, subroutine or event handler.

### SHOWSTACK

#### Commands

**Operation** The “call stack” is displayed. This stack is a list, in order, of all of the currently executing subroutines, functions and event handlers and shows the name of the routine executing and the return location of that routine.

**Notes** The SHOWSTACK command is useful when a break point or trappable error is encountered during program execution. In this situation the SHOWSTACK displays which routines are executing and where they were called from. For instance:

```
-RUN
Break at line 1060.
-SHOWSTACK
GOSUB FROM: 230
GOSUB FROM: 150
-
```

The above display indicates that the code that was executing at line 1060 when the breakpoint was encountered was invoked from a GOSUB statement at line 230 which was in a subroutine that was invoked from line 150.

The types of messages that can be displayed by a SHOWSTACK command include:

```
DEF function-name CALLED FROM: nnn
GOSUB FROM: nnn
ON CENTER CLICK FROM: nnn
ON CENTER DCLICK FROM: nnn
ON ERROR FROM: nnn
ON EVENT(n) FROM: nnn
ON KEY(nnnn) FROM: nnn
ON LEFT CLICK FROM: nnn
ON LEFT DCLICK FROM: nnn
ON MOUSE FROM: nnn
ON RIGHT CLICK FROM: nnn
ON RIGHT DCLICK FROM: nnn
ON TIMEOUT FROM: nnn
SUB PROGRAM: program-name CALLED FROM: nnn
```

---

## STEP Command

STEP executes the next program line(s) performing an automatic breakpoint after they are executed.

```
5  STEP  count
6  STEP
7  SSTEP count
8  SSTEP
```

Commands

**Operation**      **Mode 1**—The next *count* number of lines are executed and a breakpoint is performed.

**Mode 2**—A single line is executed followed by a breakpoint.

**Mode 3**—The next *count* number of statements are executed and a breakpoint is performed.

**Mode 4**—A single statement is executed followed by a breakpoint.

**Notes**            The STEP and SSTEP commands are debugging aids used in conjunction with the **BREAK** command and **STOP** statement.

The *count* associated with STEP refers to lines of statements, not single statements. When a breakpoint has occurred on a statement in a multiple statement line, a STEP executes the remaining statements on the line. The *count* associated with SSTEP refers to statements, not lines. When a breakpoint has occurred on a statement in a multiple statement line, a SSTEP executes the next statement only.

The SSTEP command differs from the STEP command not just in whether it executes a statement or a line of statements. The STEP command does not step through user-defined functions or subprograms; the SSTEP command does.

When a break occurs during execution of a subprogram or a user-defined function and a STEP command is issued, the break is not encountered until the remainder of the subprogram or function is executed and then any remaining statements on the line calling the subprogram or referencing the function is executed.

**Examples**

*This is the same program used for the SSTEP example.*

```

10  OPTION PROMPT " "
20
30  INPUT "Enter a number: ",X$
35  X$ = FN.DIGIT.STRIP$(X$)
40
50  PRINT X$
60  STOP
70
1000 DEF FN.DIGIT.STRIP$(STRING$)
1010
1020  LOCAL I%
1030  LOCAL I$
1040  LOCAL X$
1050
1060  X$ = " "
1070
1080  FOR I% = 1 TO LEN(STRING$)
1090    I$ = STRING$(I%:I%)
1100    IF I$>="0" AND I$<="9" THEN X$ = X$&I$
1110  NEXT
1120
1130  FN.DIGIT.STRIP$ = X$
1140
1150  FNEND

```

*Compare this display with the SSTEP example.*

```

-S
Break at line 30.
-S
Enter a number: 1a2
Break at line 35.
-S
Break at line 50.
-S
12
Break at line 60.
-

```



---

## TOP Command

TOP positions to the first line of the program.

1 TOP

2

**Operation** The first line in the program becomes the current line and it is displayed. Entry of the  key is a shortcut—it is synonymous with entering TOP .

**Notes** When there is no program in memory and the TOP command is used, the message “End of file.” is displayed.

**Restrictions** TOP cannot be used on *read-protected* programs.

**See also** [BOTTOM](#)

**Examples**

```
-LOAD HELPTST
-TOP
  10 REM Program HELPTST field input with help
-
```

---

## TRACE Command

TRACE sets the line number and variable *debugging* trace modes on.

- 1 TRACE
- 2 TRACE VARS

### Commands

**Operation**      **Mode 1**—Statement tracing is enabled.

**Mode 2**—Statement and variable change tracing is enabled.

**Notes**            When TRACE or TRACE VARS mode is enabled, each statement executed is traced by displaying the line number of each statement executing. When the statement is part of a multiple statement line, the line number is followed by the statement number in the line.

TRACE VARS also displays the variable name and new value for each variable whose value changes during the execution of a statement.

The display from TRACE and TRACE VARS is distinguished from normal program display by enclosing the line numbers and variable assignments in angle brackets. For instance:

```
-LIST
  10 A = 1
  20 B = 2 \ C = 3
-TRACE VARS
-RUN
<10>
  <SHARED A = 1>
<20>
  <SHARED B = 2>
<20,2>
  <SHARED C = 3>
-
```

The TRACE or TRACE VARS output is displayed on the console unless the [VIEW](#) TRACEFILE command was used to redirect the output to a disk, tape, printer or com file. When the [VIEW](#) TRACEFILE command is used the output of the TRACE or TRACE VARS is appended to the end of any existing file.

When the trace display is on the console and the statement outputs to the console, the statement output is always displayed at the start of a new line in window 0.

Line numbers for statements in an included file are shown with the line number of the main program INCLUDE statement. For instance:

```
<210:150>
```

This refers to line 150 of a program included by an INCLUDE statement at line 210. When [VIEW](#) FILEREF is in effect, this same reference might be:

```
<210[TEST.BASIC]:150[KEYDEFS.BASIC]>
```

The [VIEW](#) TRACELINE and [VIEW](#) TRACESOURCE modes affect the display of the trace output. With [VIEW](#) TRACELINE enabled the trace display is as described above. With [VIEW](#) TRACESOURCE enabled the trace display for each line of the program is preceded by a listing of the line being executed. For instance:

```
-LIST
    10 A = 1
    20 B = 2 \ C = 3
-TRACE VARS
-VIEW TRACESOURCE
-RUN
***      10 A = 1
<10>
<SHARED A = 1>
***      20 B = 2 \ C = 3
<20>
<SHARED B = 2>
<20,2>
<SHARED C = 3>
-
```

The TRACESOURCE output is identified with three asterisk characters at the start of the line. Multiple statement lines (such as line 20 above) are displayed only once, before the first statement is traced.

#### Defaults

The output of the TRACE or TRACE VARS is displayed on the console, unless the [VIEW](#) TRACEFILE is enabled.

#### See also

[UNTRACE](#), [VIEW](#)

**Examples**

This example uses the same program as the [STEP](#) command. However, lines 30 and 35 are combined into a single, multiple-statement line.

```

-TRACE
-RUN
TRACE is enabled
and the short pro-
gram is executed.
Line 30: The trace
output is displayed
followed immedi-
ately by the pro-
gram's output.
-TRACE
-RUN
<10>
<30>
Enter a number: 1A2
<30, 2>
<1060>
<1080>
<1090>
<1100>
<1100, 2>
<1110>
<1090>
<1100>
<1110>
<1090>
<1100>
<1100, 2>
<1110>
<1130>
<1150>
<50>
12
<60>
Stop at line 60.
-TRACE VARS
-RUN
<10>
<30>
Enter a number: 1A2
<SHARED X$ = "1A2">
<30, 2>
<1060>
<LOCAL X$ = ">
<1080>
<LOCAL I% = 1>
<1090>
<LOCAL I$ = "1">
<1100>
<1100, 2>
<LOCAL X$ = "1">
~
LOCAL I% = 4>
Line 1130: Assign-
ments of function
names are always
identified with the
variable identifier
PARAMETER
VALUE.
<1130>
<PARAMETER VALUE FN.DIGIT.STRIP$ = "12">
<1150>
<SHARED X$ = "12">
<50>
12
<60>
Stop at line 60.

```

Notice that line 30 and sometimes line 1100 appear twice in the TRACE output. These two lines are multiple statement lines.

For each statement on a line, TRACE displays a line. The “,2” indicates that it is the second statement on the line. A “,3” would be used for a third statement on the same line, *etc.*

Line 1100 doesn’t always appear twice because it is an IF statement and the second statement on the line is not always executed.

TRACE VARS shows all variable assignments with the variable type shown. In this example, only SHARED, LOCAL and PARAMETER VALUE are used. TRACE VARS also will identify COMMON and STATIC variable assignments.

---

## UNBREAK Command

UNBREAK disables all breakpoints or specific breakpoints.

- 1 UNBREAK [AT] *line-number*
- 2 UNBREAK AT *line-label*
- 3 UNBREAK [ON] *variable*
- 4 UNBREAK

Operation	<b>Mode 1</b> —All breakpoints for <i>line-number</i> are cleared.
	<b>Mode 2</b> —All breakpoints for <i>line-label</i> are cleared.
	<b>Mode 3</b> —All breakpoints for <i>variable</i> are cleared.
	<b>Mode 4</b> —All breakpoints are cleared.
Notes	To remove breakpoints set for a line label you must use the AT keyword. When the AT keyword is missing it assumes the label is a variable name.
Restrictions	Breakpoints are automatically cleared when a program is loaded with the <a href="#">LOAD</a> command, the <a href="#">RUN</a> <i>program-name</i> command, or the <a href="#">CHAIN</a> or <a href="#">LINK</a> statements.
See also	<a href="#">BREAK</a> , <a href="#">STEP</a>

## Examples

### -LIST

```
10 INPUT "Number of squares",COUNT%
20 PRINT \ PRINT "Nbr Square SquareRoot"
30 FOR I% = 1 TO COUNT%
40 PRINT USING "### ##### ###.###", I%, I%*I%, SQR(I%)
50 NEXT
```

*Tracing is enabled  
and a breakpoint is  
set in the FOR-  
NEXT loop.*

### -TRACE VARS

#### -B 40

*When the program  
is executed the  
breakpoint is  
encountered.*

### -RUN

```
<10>
Number of squares? 2
<SHARED COUNT% = 3>
<20>
<20,2>
Nbr Square SquareRoot
<30>
<SHARED I% = 1>
Break at line 40.
```

*Clearing the break-  
point with the  
UNBREAK com-  
mand allows execu-  
tion to continue  
without interrup-  
tion.*

### -UNB

#### -CONT

```
<40,2>
1 1 1.0000
<50>
<SHARED I% = 2>
<40>
2 4 1.4142
<50>
<SHARED I% = 3>
-
```

# UNTRACE Command

UNTRACE disables line number and variable tracing.

UNTRACE

**Operation** Both line and variable change tracing are disabled.

**Notes** Tracing is not disabled by [CHAIN](#) or [LINK](#) statements nor is it disabled by the [RUN](#) command. [NEW](#) does reset the trace mode.

If [VIEW](#) TRACEFILE is in effect the file is closed with this command.

**See also** [TRACE](#), [VIEW](#)

**Examples**

```
-LIST
10 INPUT "Number of squares",COUNT%
20 PRINT \ PRINT "Nbr Square SquareRoot"
30 FOR I% = 1 TO COUNT%
40 PRINT USING "### ##### ###.###", I%,I%*I%,SQR(I%)
50 NEXT
```

Tracing is enabled  
and a breakpoint is  
set in the FOR-  
NEXT loop.

```
-TRACE VARS
-B 40

-RUN
<10>
Number of squares? 4
<SHARED COUNT% = 3>
<20>
<20,2>
Nbr Square SquareRoot
<30>
<SHARED I% = 1>
Break at line 40.
```

When the program  
is executed the  
breakpoint is  
encountered.

Clearing the break-  
points and trace  
mode allows execu-  
tion to continue  
without interrup-  
tion and without  
disrupting the dis-  
play.

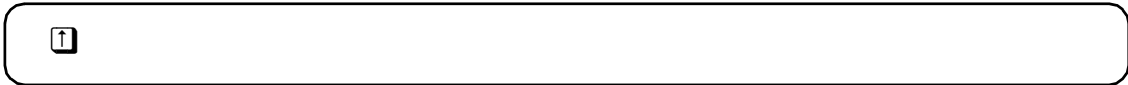
```
-UNB
-UNT
-CONT
1      1      1.0000
2      4      1.4142
3      9      1.7321
4     16      2.0000
-
```



---

# UP Command

UP positions to the previous line of the program.



**Operation**      The current line is set to the previous line of the program and displayed.

**Notes**            If you are currently positioned at the first line of the program, an up command displays: "Top of file."

**Restrictions**    UP cannot be used on *read-protected* programs.

**See also**         [BOTTOM](#), [DOWN](#), [TOP](#)

**Examples**        -LOAD HELPTXT

*In this program, trying to position to line 10000 displays line 700000, indicating that there are no lines between 10000 and 700000.*

*An UP command finds the first line prior to 700000.*

*The AUTO command with no arguments begins automatic line insert after the current line.*

-10000  
700000 REM Function definitions

- 1240

-A 1250

---

## VARs Command

VARs displays the name and contents of all variables or specific variables.

- 1 **\_VARs** *variable-name...*
- 2 **\_VARs**

### Commands

**Operation**      **Mode 1**—The specified *variable* or variables are displayed with their current values. Specifying multiple variable names causes those variables to be displayed in the same sequence as specified.

**Mode 2**—All variables currently defined are displayed with their current values. The sequence of the variables displayed is alphabetical by name.

**Notes**            The VARs command display uses the current [VIEW](#) settings:

- ▶ [VIEW](#) ELEMENTS or NOELEMENTS controls how arrays are displayed. NOELEMENTS causes the array name only to be displayed with its dimensioned size. When ELEMENTS is specified, each of the elements of the array is displayed.
- ▶ [VIEW](#) LOCAL, CURRENT, or NOLOCAL controls which variables are displayed: all (LOCAL), shared only (NOLOCAL), or just the variables that are accessible to the program at its current execution location (CURRENT).
- ▶ [VIEW](#) VARsFILE controls where the VARs display is output.

A long VARs display can be canceled with the **Quit** key or a **Break**, **C**.

VARs always clears the screen prior to displaying the variable values except when a single, nondimensioned variable is displayed.

**See also**        [TRACE](#), [VIEW](#)

## Examples

### -LIST

```
10  OPTION BASE 1
20  DIM ARRAY%(3,3)
30  MAT ARRAY% = (15)
40  READ ARRAY%(2,3),A$,B$,C%
50
60  DATA 5,"TEST 1","SAMPLE",999
```

### -RUN

### -VARS

```
A$ = "TEST 1 "
ARRAY%(3,3)
B$ = "SAMPLE"
C% = 999
```

*The VARS command displays all of the variables used in the program and their values.*

*The default VIEW is NOELEMENTS so the first VARS display only shows the array name.*

### -VIEW ELEMENTS

### -VARS

```
A$ = "TEST 1 "
ARRAY%(3,3)
  (1,1) = 15
  (1,2) = 15
  (1,3) = 15
  (2,1) = 15
  (2,2) = 15
  (2,3) = 5
  (3,1) = 15
  (3,2) = 15
  (3,3) = 15
B$ = "SAMPLE"
C% = 999
-
```

*After setting VIEW ELEMENTS the VARS display shows the value for each element of the array.*

---

## VIEW Command

The VIEW command controls how information is displayed by MultiUser BASIC.

1 **VIEW** *mode*[, *mode*]...

2 **VIEW**

---

<i>mode</i>	»	<b><u>CONSTANTS</u></b> <b><u>NOCONSTANTS</u></b>
	»	<b><u>ELEMENTS</u></b> <b><u>NOELEMENTS</u></b>
	»	<b><u>INCLUDE</u></b> <b><u>NOINCLUDE</u></b>
	»	<b><u>LINEREF</u></b> <b><u>FILEREF</u></b>
	»	<b><u>LOCAL</u></b> <b><u>NOLOCAL</u></b> <b><u>CURRENT</u></b>
	»	<b><u>TRACESOURCE</u></b> <b><u>TSOURCE</u></b> <b><u>TRACELINE</u></b> <b><u>TLINE</u></b>
	»	<b><u>SAVEA</u></b> <b><u>SAVECOMPRESS</u></b> <b><u>SAVEUNNUMBER</u></b>
	»	<b><u>WARNING</u></b> <b><u>NOWARNING</u></b>
	»	<b><u>VARFILE</u></b> [ <i>filename</i> ] <b><u>VFILE</u></b> [ <i>filename</i> ]
	»	<b><u>TRACEFILE</u></b> [ <i>filename</i> ] <b><u>TFILE</u></b> [ <i>filename</i> ]

## Operation

**Mode 1**—The viewing mode is set.

**Mode 2**—The current viewing modes are displayed.

For instance:

```
-VIEW
CURRENT
NOINCLUDE
ELEMENTS
NOCONSTANTS
LINEREF
TRACELINE
WARNING
SAVE MODE - COMPRESSED
```

## Notes

The view *mode* controls how and what information is displayed by a program list or during program debugging. The VIEW command itself doesn't display information, except for the view modes in effect ([Mode 2](#)).

## Modes

**CONSTANTS** Specifies whether or not numeric and string literals are displayed in cross-reference listings ([XREF](#) and [LPXREF](#)).

**CURRENT** See [LOCAL](#) description.

**ELEMENTS** Specifies that the individual elements of arrays are displayed in variable displays ([VARS](#)). This is a default setting unless the VIEW\_ELEMENTS environment variable has been defined. See “[Environment Variables](#)” on page 128.

**FILEREF** See the [LINEREF](#) description.

**INCLUDE** Specifies that included source files are displayed by the [LIST](#) and [LPLIST](#) commands. When INCLUDE is enabled the included source programs are displayed immediately following the line containing the [INCLUDE](#) statement. The included source lines are marked with a “+” at the beginning of each displayed line.

The default value is VIEW NOINCLUDE unless the VIEW\_INCLUDE environment variable has been defined. See “[Environment Variables](#)” on page 128.

**LINEREF** Controls how lines numbers are displayed in [BREAK](#), [SSTEP](#), [STEP](#), [TRACE](#) and [XREF](#) displays.

**LINEREF** Only line numbers are displayed.

```
-VIEW LINEREF
-STEP
```

```
<11390:560>
Break at line 11390:560.
-
```

**FILEREF** The line number and the source file name is displayed.

```
-VIEW FILEREF
-STEP
<1170[BJ.BASIC]>
Break at line 1170[BJ.BASIC]:30[CLR.BASIC].
-STEP
<1170[BJ.BASIC]:30[CLR.BASIC]>
  <SHARED BLACK% = 0>
Break at line 1170[BJ.BASIC]:40[CLR.BASIC].
-
```

The default value is VIEW LINEREF unless the VIEW\_LINEREF environment variable has been defined. See “[Environment Variables](#)” on page 128.

**LOCAL** Controls which variables are viewed with the [VARS](#) command.

**CURRENT** Only variables that have current scope are shown. This will include all [SHARED](#) variables and any [LOCAL](#) and [STATIC](#) variables defined in the current subprogram or function definition.

**LOCAL** All variables are displayed.

**NOLOCAL** Only global or [SHARED](#) variables are shown.

The default value for VIEW LOCAL is CURRENT unless the VIEW\_LOCAL environment variable has been defined. See “[Environment Variables](#)” on page 128.

**NOCONSTANTS** Indicates that numeric and string constants are not included in cross-reference listings. This is a default setting unless the VIEW\_CONSTANTS environment variable has been defined. See “[Environment Variables](#)” on page 128.

**NOELEMENTS** Specifies that the individual elements of arrays are not displayed in variable displays ([VARS](#)).

**NOINCLUDE** Specifies that included source files are not displayed by the [LIST](#) and [LPLIST](#) commands. When NOINCLUDE is enabled only the main program code is listed. This is a default setting unless the VIEW\_INCLUDE environment variable has been defined. See “[Environment Variables](#)” on page 128.

**NOLOCAL** See [LOCAL](#) description.

**NOWARNING** See [WARNING](#) description.

**SAVEA** Specifies that the [SAVE](#) command is a synonym to the [SAVEA](#) command. This is the default or initial setting unless the VIEW\_SAVE environment variable has been set prior to invoking BASIC.

**SAVECOMPRESS** Specifies that the [SAVE](#) command is a synonym to the [SAVEA](#) command.

**SAVEUNNUMBER** Specifies that the [SAVE](#) command is a synonym to the [SAVEU](#) command.

**TFILE** This is a synonym to [TRACEFILE](#).

**TLINE** This is a synonym to [TRACELINE](#).

**TRACEFILE** Specifies that the [TRACE](#) and TRACE VARS commands display their output on the console (default) or to a specific disk, tape or com file. Specifying VIEW TRACEFILE without a filename causes subsequent trace output to be displayed on the console.

The *filename* is opened when the next [TRACE](#) or TRACE VARS command is issued and closed when the UNTRACE command is used. Note that it is opened for output append and any existing stream file by that name will not be erased.

The default VIEW TRACEFILE can be set externally by defining the VIEW\_TRACEFILE environment variable. See “[Environment Variables](#)” on page 128.

**TRACELINE** Specifies that the [TRACE](#) and TRACE VARS display shows only the line number of the lines being executed. This is a default setting unless the VIEW\_TRACESOURCE environment variable has been defined. See “[Environment Variables](#)” on page 128.

**TRACESOURCE** Specifies that the [TRACE](#) and TRACE VARS commands will display the source line text along with the source line number.

**TSOURCE** This a synonym to [TRACESOURCE](#).

**VARFILE** Specifies that the [VARS](#) command displays its output on the console (default) or to a specific disk, tape or com file. Specifying VIEW VARFILE without a filename causes subsequent [VARS](#) output to be displayed on the console.

The *filename* is opened when the [VARS](#) command is used and is closed at the end of the variable display. Note that it is opened

for output append and any existing stream file by that name will not be erased.

The default VIEW VARSFILE can be set externally by defining the VIEW\_VARSFILE environment variable. See “[Environment Variables](#)” on page 128.

**VFILE** This is a synonym to [VARSFILE](#).

**WARNING** Enables warning message display. The NOWARNING mode disables all warning messages. The warning messages are listed at the end of appendix "D: Error Codes and Messages." This is a default setting unless the VIEW\_WARNING environment variable has been defined. See “[Environment Variables](#)” on page 128.

## Environment Variables

The initial settings for each of the viewing options can be set external to the BASIC interpreter by defining environment variables. Environment variables are defined by the SET utility (and also the ACCOUNT command in THEOS 32, Version 4).

For instance, to set the default include file viewing mode, use the command:

```
>SET VIEW_INCLUDE=INCLUDE
```

When this is done the include file viewing mode will be INCLUDE each time that you load the interpreter. After loading the interpreter you can change the mode with the VIEW command. This viewing mode default will be in effect until you log off of this account or until you change the VIEW\_INCLUDE environment variable value.



The various environment variables and their possible values are:

Variable name	Values
VIEW_CONSTANT	<u>C</u> ONSTANT <u>N</u> OCONSTANT
VIEW_ELEMENTS	<u>E</u> LEMENTS <u>N</u> OELEMENTS
VIEW_INCLUDE	<u>I</u> NCLUDE <u>N</u> OINCLUDE
VIEW_LINEREF	<u>L</u> INEREF <u>F</u> ILEREF
VIEW_LOCAL	<u>C</u> URRENT <u>L</u> OCAL <u>N</u> OLOCAL
VIEW_SAVE	<u>A</u> SCII <u>C</u> OMPRESSED <u>U</u> NNUMBERED
VIEW_TRACEFILE	<i>filename</i>
VIEW_TRACESOURCE	<u>T</u> RACESOURCE <u>T</u> RACE <u>L</u> INE <u>T</u> LINE <u>T</u> SOURCE
VIEW_VARSFILE	<i>filename</i>
VIEW_WARNING	<u>W</u> ARNING <u>N</u> OWARNING

Remember that the syntax for setting environment variables uses an equal sign between the variable name and the value.

**Defaults** The default viewing modes are: [CURRENT](#), [NOINCLUDE](#), [ELEMENTS](#), [NOCONSTANTS](#), [LINEREF](#), [TRACELINE](#), [SAVEA](#) and [WARNING](#), unless otherwise set with an environment variable.

**See also** [BREAK](#), [LPXREF](#), [SAVE](#), [SAVEA](#), [SAVEC](#), [SAVEU](#), [STEP](#), [TRACE](#), [VARS](#), [XREF](#)

**Examples**

Refer to the affected commands for examples of the effect of the various VIEW modes.

```
-VIEW  
  NOLOCAL  
  NOINCLUDE  
  NOELEMENTS  
  NOCONSTANTS  
  LINEREF  
  TRACELINE  
  WARNING  
  SAVE MODE - ASCII  
-VIEW CURRENT  
-VIEW FILEREF  
-VIEW TSOURCE  
-VIEW TFILE TRACE.OUTPUT  
-VIEW VARS VARS.OUTPUT  
-VIEW SAVEC  
-VIEW  
  CURRENT  
  NOINCLUDE  
  NOELEMENTS  
  NOCONSTANTS  
  FILEREF  
  TRACESOURCE  
  TRACEFILE TRACE.OUTPUT  
  VARSFILE VARS.OUTPUT  
  WARNING  
  SAVE MODE - COMPRESS  
-
```

---

## XREF Command

XREF displays the cross reference listing of the program, showing all references to lines, line-labels, variables, and constants.

- 1 **XREF**
  - 2 **XREF** *line-range*
  - 3 **XREF** *filename*
  - 4 **XREF** *line-range filename*

Commands

Operation	<b>Mode 1</b> —Displays the cross-reference listing.
	<b>Mode 2</b> —Displays the cross-reference listing for the <i>line-range</i> portion of the program.
	<b>Mode 3</b> —Generates the cross-reference listing for the entire program and outputs the result to <i>filename</i> .
	<b>Mode 4</b> —Generates the cross-reference listing for the <i>line-range</i> portion of the program and outputs the result to <i>filename</i> .
Restrictions	XREF cannot be used on <i>read-protected</i> programs.
Notes	<p>The heading for each page of the cross-reference listing contains the name and version number of MultiUser BASIC, the complete file name of the program, the date and time of the listing, and the page number.</p> <p>The cross-reference listing is composed of six sections:</p> <ul style="list-style-type: none"><li>▶ Line number references</li><li>▶ Label references</li><li>▶ Call references</li><li>▶ Variable references</li><li>▶ Constant references, numeric integers first, followed by floating point values and then strings. This is included only when <a href="#">VIEW</a> CONSTANTS is set.</li><li>▶ Include references</li></ul>

Each section is sorted in ascending sequence of the term being referenced. All references to a term are sorted by the line number of the statement making the reference.

When a *line-range* is specified, the cross-reference pertains only to that portion of the program. No references to variables or lines outside of *line-range* are resolved.

The current setting of [VIEW LINEREF](#) or [VIEW FILEREF](#) is used when displaying line numbers during the cross-reference listing.

A long cross-reference listing may be canceled with the [Quit](#) key or a [Break](#), [C](#).

Each section of the cross-reference listing contains different types of information:

#### ■ Line Number References

LINE REFERENCES	
Line Being Referenced	Line Doing the Referencing
890 (RE.START) SAMPLE.BASIC	875 SAMPLE.BASIC
900 SAMPLE.BASIC	1000 SAMPLE.BASIC
	1120 SAMPLE.BASIC
	1255 SAMPLE.BASIC

The line numbers that are referenced in the program are listed on the left side. If the line also has a line label it is displayed in parentheses immediately after the line number.

All of the lines that reference a particular line number are listed on the right.

Notice that the source program name is included for both the referenced line number and the referencing line number. These source program names are displayed independent of the current [VIEW LINEREF](#) | [FILEREF](#) mode.

■ Label References

LABEL REFERENCES		
Label references	Define in	Reference or
Line number referenced Defined		
DISPLAY.HELP.PAGE	HELP	
15300:5840(DISPLAY.HELP.PAGE)	HELP.BASIC	
	Defined	
15300:3550	HELP.BASIC	Reference
15300:4360	HELP.BASIC	Reference
END.HELP	HELP	
15300:5470(END.HELP)		
HELP.BASIC		Defined

Each label defined or used in the program is listed along with an identifier indicating if the label is defined in the `MAIN` program area or in a function definition or subprogram. In the above example, the label `DISPLAY.HELP.PAGE` is defined in the subprogram `HELP`. Labels defined in function definitions are identified with the function name, which always start with `FN`.

Following each label is an indented listing of all of the lines in the program that reference that line label. Listed first is the line that defines the label itself. Following this first line the other lines referencing the label are listed in line number sequence.

Notice that the source program name is included for the referencing line number. These source program names are displayed independent of the current [VIEW LINEREF | FILEREF](#) mode.

Because line labels can be local to an include file, a subprogram or a function definition, it is possible to have multiple labels with the same name.

## ■ Call References

### CALL REFERENCES

Name of program or functions	Type of program
Line number doing the call. First refer ... except for old C	
BACCESS	New Style C Call
15270:3940:330 LOOKFILE.BASIC	
15270:3970:450 LOOKFILE.BASIC	
15300:1360 HELP.BASIC	
FN.LOOKFILE	User Defined Function
15270:3940:350 LOOKFILE.BASIC	Defined
15270:190 SAMPLE1.BASIC	Referenced
HELP	Sub Program
15300:530 HELP.BASIC	
3630 SAMPLE.BASIC	
6510 SAMPLE.BASIC	

Each subprogram name, user-defined function or C language function used in the program is listed alphabetically with its type.

Below each name the indented references to the name are listed. Except for old-style C language functions, the first reference listed is always the definition of the name. For subprogram names this is the [SUB](#) statement; for user-defined functions this is the [DEF FN](#) statement; for new-style C language functions it is the [DECLARE CALL](#) CALL statement.

References to user-defined functions further identify the reference to be either a simple reference or an assignment. The assignment of a user-defined function is only performed inside of the function definition itself.

Notice that the source program name is included for the referencing line number. These source program names are displayed independent of the current [VIEW LINEREF](#) | [FILEREF](#) mode.

■ Variable References

VARIABLE REFERENCES			
Variable Name	Type of variable	Where defined	
Line number of reference		Type of reference	
AA%	Local	FORMAT.HELP.TEXT	
15300:7530	HELP.BASIC		Definition
ANS\$	MAIN		
2780	SAMPLE.BASIC		Read
4360	SAMPLE.BASIC		Read
4440	SAMPLE.BASIC		Assignment
4500	SAMPLE.BASIC		Reference
4500	SAMPLE.BASIC		Write
ANS\$	HELP		
15300:3770	HELP.BASIC		Assignment
15300:3780	HELP.BASIC		Reference

Each variable name used in the program is listed alphabetically along with the type of variable and the program, subprogram or function definition that defines the variable.

The variable types identified include:

Local	Variable name is local to a subprogram or user-defined function.
Shared	Variable name is global to the entire program.
Static	Variable name is local to a subprogram or user-defined function.

Array variables are indicated by specifying the array dimensions immediately following the variable type. For instance: “Shared 100” indicates that the variable is a **SHARED** access array name that is single-dimensioned for a size of 100.

Following each variable name is an indented listing of all of the lines that use that variable. These lines are listed in line-number sequence and specify how the variable name is used at that line.

The referencing types identified include:

Assignment	Indicates that the variable is assigned a value with a <a href="#">LET</a> statement.
Definition	Indicates that the variable name is defined at this line with a <a href="#">COMMON</a> , <a href="#">DIM</a> , <a href="#">LOCAL</a> , <a href="#">SHARED</a> or <a href="#">STATIC</a> statement.
Read	Indicates that the variable name is used in some type of input statement.
Reference	Indicates that the variable is merely referenced in a statement at this line.
Write	Indicates that the variable name is used in some type of output statement.

When a variable name is used multiple times in the same line it is listed multiple times in the cross-reference listing.

Because variable names can be local to a subprogram or a function definition, it is possible to have multiple instances of the same variable name listed in the cross-reference listing.

Notice that the source program name is included for the referencing line number. These source program names are displayed independent of the current [VIEW](#) LINEREF | FILEREF mode.



■ Constant References

CONSTANT REFERENCES	
Constant being referenced	Referencing line
-14	15270:3060 OPTIONS.BASIC
-12	15270:3010 OPTIONS.BASIC
-10	11090 SAMPLE.BASIC 15270:2910 OPTIONS.BASIC

If [VIEW](#) CONSTANTS is set, the cross-reference listing includes a listing of all numeric and string constants used in the program. These are listing in the following sequence:

- ▶ Integer constants
- ▶ Numeric constants
- ▶ String literals

Within each section, the constants are sorted in ascending order.

No other information is provided because constants can only be used, not assigned, read, or written.

Notice that the source program name is included for the referencing line number. These source program names are displayed independent of the current [VIEW](#) LINEREF | FILEREF mode.

■ Include References

INCLUDE REFERENCES	
Program being included	Line number of INCLUDE statement
COLORS.BASIC	1400 SAMPLES.BASIC
KEYDEFS.BASIC	1410 SAMPLES.BASIC
OPTIONS.BASIC	15270 SAMPLES.BASIC
LOOKFILE.BASIC	15270:3740 OPTIONS.BASIC
HELP.BASIC	15300 SAMPLES.BASIC

Each program file included in the program is listed along with the location where it is included in the program.

**See also**      [LIST](#), [LPLIST](#), [LPXREF](#), [VIEW](#)



# 4 Functions and Statements

---

This section describes each of the statements and functions that are available in THEOS MultiUser BASIC. The individual descriptions are given in alphabetical order. Below, the statements and functions are listed and grouped by their main function or operation.

## ■ Functions

All functions return a value. That is the definition of a function: a procedure operating on zero or more parameters that returns one, and only one, result.

Although most functions return a value without changing anything else, a few do. For example, the `LEFT$` function returns the leftmost portion of a string without changing the value of the string; the `SYS ENV$` function returns the value of a system environment variable and can change the value of that variable; the `SET` and `RESET` functions change the status of a semaphore, returning the prior status.

## ■ Statements

Statements do not return a value, although many change the value of one or more variables in the program.

**PROGRAM CONTROL:**

OPTION	Set general operating parameters.
REM	Define a program comment.

**EXECUTION CONTROL:**

SELECT CASE OTHERWISE CEND	Define a “choose one of” program structure.
IF THEN ELSE IFEND	Define condition and identifies the statements that are conditionally executed.
FOR NEXT	Define a repeat with increment program loop.
GOTO ON GOTO	Unconditional and conditional program branches.
GOSUB ON GOSUB RETURN	Subroutine control.
SUB END SUB	Subprogram control.
ON ERROR ON EVENT ON KEY ON TIMEOUT RESUME TIMER	Asynchronous event processing control.
WHILE WEND	Define a conditionally repeated program loop.
END QUIT STOP	Terminate execution of the program.
SLEEP	Suspend processing.
WAIT	Wait for operator input or device ready.
BREAK CONTINUE	Abort or repeat a <a href="#">FOR-NEXT</a> , <a href="#">WHILE-WEND</a> program structure.

---

**ASSIGNMENT & DECLARATION:**

CLEAR	Initialize variables.
COMMON	Define variables & arrays shared between programs.
DIM	Define arrays.
DATA RESTORE	Define data values.
DEF FN FNEND	Define user-defined function.
LET	Assign value to variable.
LOCAL SHARED STATIC	Declare variables or arrays as being local to a subprogram or function or shared with the main program.
MAT MAT SORT	Initialize, copy, or sort array contents.
SWAP	Exchange value between two variables.

---

**STRING MANIPULATION:**

LCASE\$ UCASE\$	Convert a string to lowercase or uppercase.
LEFT\$ MID\$ RIGHT\$	Return a portion of a string (left, middle, or right).
LEN	Return length of string.
LPAD\$ RPAD\$	Add spaces to the left or right end of a string.
LTRIM\$ RTRIM\$ TRIM\$	Remove spaces from the left or right end of a string, or remove leading spaces, trailing spaces, and embedded multiple spaces.
MATCH	Compare a string to a mask.
OVR\$	Overlay one string onto another.
RPT\$ SPACE\$	Create a string of repetitive strings or spaces.
SCH	Locate the position of a substring in a string.
DEL\$ EXT\$ INS\$ REP\$	Delete, extract, insert, or replace a subfield in a string.

---

**NUMERIC MANIPULATION:**

ABS	Return the unsigned value or the sign of a value.
SGN	
CEIL	Return the ceiling or floor of a value.
FLOOR	
EPS	Return the smallest or largest value maintained by the system.
INF	
EXP	Return the exponential value of an expression.
FP	Return the fractional or integer portion of a value.
IP	
INT	
GCD	Compute the Greatest Common Divisor.
LOG, LOG2, LOG10	Compute the natural, binary, or common logarithm.
MAX	Return the larger or smaller of two values.
MIN	
MOD	Compute the modulo or remainder function.
PI	Return the constant $\pi$ .
RANDOMIZE	Reseed the random number generator or return the next random number.
RND	
ROUND	Round a value.
SQR	Compute the square root.

---

**STRING & NUMERIC CONVERSION:**

ASC ORD CHR\$	Convert between character and ASCII value.
BIN, BINOF\$ HEX, HEXOF\$ OCT, OCTOF\$	Convert a number to its binary, hexadecimal, or octal string representation, and vice versa.
FIX FLOAT	Convert between integer and numeric.
FORMAT\$	Format a number as a string.
NBR	Test a string for valid number.
VAL STR\$	Convert between number and string representation.
STRTIMES\$	Format a date and time as a string.

---

**CONSOLE/PRINTER INPUT & OUTPUT:**

AT\$	Generate cursor positioning string.
CLS\$ CRT\$	Generate string for clear screen/page eject, or change display attribute, or draw lines.
COLOR	Change the colors on the console.
INP ON.KEY.TOKEN	Return the control key value used to terminate the last input or the last key value used to invoke an <a href="#">ON KEY</a> trap.
LINE PAGE	Return the screen, window, or printer page width and depth.
POS TAB	Return the column position or set the column position during print operations.
YESNO\$	Accept a 'Y' or 'N' response from operator.
ERRMSG\$	Display message and accept reply.

**FILE INPUT & OUTPUT:**

OPEN CLOSE UNLOCK	Open or close a data file or device.
LOCKED.BY	Return partition number of user locking file or record
NEXT.FILE	Returns the next available I/O channel number.
IOLIST	Declares a name that can be used to read or write multiple fields.
INPUT LINPUT LINPUT USING	Accept input from the operator or a data file.
PRINT PRINT USING	Display data on console or file.
DELETE	Remove a record from a data file.
GET UNGET PUT	Accept or display characters on console or device.
READ, READNEXT, READPREV WRITE	Read or write records to a data file. (READ also may read from <a href="#">DATA</a> statements.)
MAT INPUT MAT PRINT MAT READ MAT READNEXT MAT READPREV MAT WRITE	Read or write array variables from or to the console or data files.
EOF	Return end-of-file status of an i/o channel.
MOUNT	Allow removable disk to be changed.



---

**PROGRAM LINKAGE:**

CALL	Invoke a subprogram or a 'C' language routine.
CALL.RETURN.VALUE	
DECLARE CALL	Define name and formal arguments to a C language function.
CHAIN	Terminate current program and transfer control to another program.
LINK	
RUN	
CSI	Execute a system command and return.
SYSTEM	
CSI.RETURN.CODE	Return code from CSI or SYSTEM.
INCLUDE	Specifies a source program file that will be included in the current program during execution by the interpreter or compilation.

---

**DATE & TIME:**

DATE\$	Convert between date string and day number.
DAY	
DTE\$	Validate date string.
MSEC	Return time since system boot, in milliseconds.
SECOND	Convert between time string and seconds.
TIME\$	
STRTIME\$	Format date and time.

---

**WINDOW MANAGEMENT:**

WINDOW CHOICE	Defines screen windows; selects window to be active.
WINDOW OPEN	
WINDOW CLOSE	
WINDOW SELECT	
AVAIL.WINDOWS	Returns number of windows available for use or already used.
TOTAL.WINDOWS	
USED.WINDOWS	
WINDOW EDIT	Allows an array of strings to be entered and maintained in a special edit window.
WINDOW REFRESH	Display or remove a window from the screen.
WINDOW REMOVE	
WINDOW CLEAR	Clears the contents of a window to spaces or a specific repeating character.
WINDOW CLIP	Define or change the attributes of an existing screen window.
WINDOW FRAME	
WINDOW INVERT	
WINDOW TITLE	
WINDOW GET TITLE	Report cursor location and attributes of screen windows.
WINDOW LOCATE	
WINDOW STATUS	
WINDOW SAVE	Write or read a screen window on disk.
WINDOW RESTORE	
WINDOW COPY	Copy, get or take text from a window or move a window to a new location on the screen.
WINDOW GET	
WINDOW MOVE	
WINDOW TAKE	

---

**MOUSE CONTROL:**

MOUSE.BUTTON	Functions that return the current mouse button and mouse cursor location.
MOUSE.COL	
MOUSE.FRAME	
MOUSE.ROW	
MOUSE.WINDOW	
ON MOUSE	A collection of statements that allow mouse action trapping to be enabled and disabled, and to specify the specific mouse event(s) to be trapped.
GET COMMON	Transfer data items to another task.
PUT COMMON	

---

**VDI GRAPHICS:**

FILL	Draw and fill various shapes on the device.
FILL BAR	
FILL CIRCLE	
FILL PIE	
PLOT	Draw various shapes on the device.
PLOT ARC	
PLOT BAR	
PLOT CIRCLE	
PLOT PIE	
SET FILL	Define attributes of drawing tools.
SET LINE	
SET MARKER	
SET TEXT	
TEXT	Display text on device.
VDI	General interface to VDI devices.

---

**ERROR PROCESSING:**

ON ERROR	Define routine to handle trappable errors.
ERR	Error number and line.
ERL	
ERRMSG\$	Display message and accept reply.

---

**MULTITASKING:**

SEMAPHORE	Catalog a semaphore name.
ACTIVATE	Start and stop a subtask.
KILL	
EVENT	Return semaphore status; <b>RESET</b> and <b>SET</b> also change the status of a semaphore.
SET	
RESET	
WAIT EVENT	Synchronize processing with an event.

---

**TRIGONOMETRIC FUNCTIONS:**

COS	Compute the functions: cosine, cotangent, cosecant, secant, sine, tangent.
COT	
CSC	
SEC	
SIN	
TAN	
ACOS	Compute the inverse functions: arccosine, arccotangent, arccosecant, arcsecant, arcsine, and arctangent.
ACOT	
ACSC	
ASEC	
ASIN	
ATAN, ATN	
COSH	Compute the hyperbolic functions: hyperbolic cosine, hyperbolic cotangent, hyperbolic cosecant, hyperbolic secant, hyperbolic sine, hyperbolic tangent.
COTH	
CSCH	
SECH	
SINH	
TANH	
ANGLE	Compute the angle in a right triangle.
DEG	Convert between degrees and radians.
RAD	

---

**BIT MANIPULATION:**

LRL, LRR, LSL, LSR	Perform a logical rotate left or right on the bits of a value or perform a logical shift left or right on the bits of a value.
--------------------	--

---

**MISCELLANEOUS:**

ADDR OF	Compute the memory location of a variable.
CMDARG\$	Return a command line argument.
SYS.ENV\$	Return or set a system environment value.

Although statements are entered separately, there are many statements that are closely tied to another statement during execution. As an example, the **NEXT** statement is very dependent upon the **FOR** statement and visa versa. In the statement descriptions that follow, these types of statements are only fully described under the parent or main statement of the group.

For example, the **NEXT** statement is described partially under and fully under the **FOR** statement; the **CASE**, **OTHERWISE**, and **CEND** statements are described partially under their own headings and fully under the **SELECT** statement, *etc.* Specifically:

<b>DATA</b>	describes	<b>RESTORE</b>
<b>DEF FN</b>	describes	<b>FNEND</b>
<b>FOR</b>	describes	<b>NEXT</b>
<b>GET COMMON</b>	describes	<b>PUT COMMON</b>
<b>GOSUB</b>	describes	<b>RETURN</b>
<b>IF</b>	describes	<b>THEN</b> <b>ELSE</b> <b>IFEND</b>
<b>SELECT</b>	describes	<b>CASE</b> <b>OTHERWISE</b> <b>CEND</b>
<b>SUB</b>	describes	<b>END SUB</b>
<b>WHILE</b>	describes	<b>WEND</b>

---

## ABS Function

ABS returns the absolute, or unsigned value of a number.

**ABS**( *numeric-expression* )

**Operation**      The absolute, or unsigned, value of *numeric-expression* is returned as the value of the function.

**Notes**            *numeric-expression* may be an integer or a numeric value. The returned value is always a numeric value.

**See also**        [SGN](#)

**Examples**        1000 INPUT "Enter quantity and amount: ",QTY%,PRICE.EACH  
                      1010  
                      1020 PRINT "Extended amount is ";ABS(QTY%\*PRICE.EACH)

---

## ACOS Function

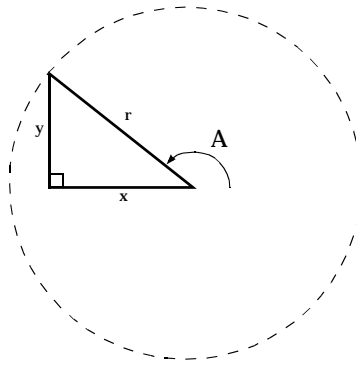
ACOS returns the *arccosine* or *inverse cosine* of a number.

**ACOS**( *numeric-expression* )

**Operation**      The *numeric-expression* is evaluated and interpreted as the cosine of an angle. That angle is computed and returned. The angle returned is expressed in degrees or radians, depending upon the status of the current **OPTION** DEGREE or **OPTION** RADIAN statement.

**Notes**            Arccosines have values that range from 0 to  $\pi$  radians or 0 to 180°.

The *arc*, or inverse trigonometric functions, provide a means of computing an angle when the base and height of a point on the circumference of a circle or arc is known:



In the above figure, the angle is A, the radius of the circle or arc is  $r$ , and the base and height of a point on the circumference is  $x$  and  $y$ . The *arc* trigonometric functions have the following relationships:

$$\begin{aligned} A &= \operatorname{asin}\left(\frac{y}{r}\right) & A &= \operatorname{acos}\left(\frac{x}{r}\right) \\ A &= \operatorname{atan}\left(\frac{y}{x}\right) & A &= \operatorname{acot}\left(\frac{x}{y}\right) \\ A &= \operatorname{asec}\left(\frac{r}{x}\right) & A &= \operatorname{acsc}\left(\frac{r}{y}\right) \end{aligned}$$

**Restrictions**      The value of *numeric-expression* must be in the domain of -1 to +1. Values outside of this domain cause ACOS to return a meaningless value.

**Examples**

*This example  
accepts an angle's  
cosine.*

*The angle is then  
computed and  
translated into  
degrees, minutes,  
and seconds using  
the FP function.*

```

10  OPTION PROMPT " ", DEGREE
20
30  INPUT "Enter cosine of an angle: ",VALUE
40  PRINT
50
60  MY.ANGLE = ACOS(VALUE)
70  MY.ANGLE.MIN = FP(MY.ANGLE)*60 ! minutes of angle
80  MY.ANGLE.SEC = FP(MY.ANGLE.MIN)*60 ! and seconds
90
100 MY.ANGLES$ = STR$(IP(MY.ANGLE))&CHR$(248)&
      STR$(IP(MY.ANGLE.MIN))&"' "&
      STR$(ROUND(MY.ANGLE.SEC,4))&" " " "
110
120 PRINT USING "The arccosine of 'e is 'e",
      STR(VALUE),MY.ANGLES$

```

**Enter cosine of an angle: .71934**

**The arccosine of 0.71934 is 43°59'59.9407"**



---

## ACOT Function

ACOT returns the *arccotangent* or *inverse cotangent* of a number.

**ACOT**( *numeric-expression* )

**Operation**      The *numeric-expression* is evaluated and interpreted as the cotangent of an angle. That angle is computed and returned. The angle returned is expressed in degrees or radians, depending upon the status of the current [OPTION DEGREE](#) or [OPTION RADIAN](#) statement.

**Notes**            Arccotangents have values that range from 0 to  $\pi$  radians or 0 to 180°.  
Refer to the [ACOS](#) function for a description of inverse trig relationships.

**See also**        Other trigonometric functions, [OPTION](#) statement

**Examples**

```
10 OPTION PROMPT "", DEGREE
20
30 INPUT "Enter cotangent of an angle: ",VALUE
40 PRINT
50
60 MY.ANGLE = ACOT(VALUE)
70 MY.ANGLE.MIN = FP(MY.ANGLE)*60 ! Determine minutes
80 MY.ANGLE.SEC = FP(MY.ANGLE.MIN)*60 ! and seconds
90
100 MY.ANGLE$ = STR$(IP(MY.ANGLE))&CHR$(248)&
    STR$(IP(MY.ANGLE.MIN))&"' "&
    STR$(ROUND(MY.ANGLE.SEC,4))&"'"
110
120 PRINT USING "The arccotangent of 'e is 'e",
    STR(VALUE),MY.ANGLE$
```

Enter cotangent of an angle: 1.015862

The arctangent of 1.015862 is 44°32'57.0194"

Statements

---

## ACSC Function

ACSC returns the *arccosecant* or *inverse cosecant* of a number.

**ACSC**( *numeric-expression* )

**Operation**      The *numeric-expression* is evaluated and interpreted as the cosecant of an angle. That angle is computed and returned. The angle returned is expressed in degrees or radians, depending upon the status of the current **OPTION** **DEGREE** or **OPTION** **RADIAN** statement.

**Notes**            Arc cosecants have values that range from 0 to  $\pi/2$  radians or 0° to 90° (when *numeric-expression*  $\geq +1$ ) or -p to  $-\pi/2$  radians or 0° to -90° (when *numeric-expression*  $\leq -1$ ).

Refer to the **ACOS** function for a description of inverse relationships.

**Restrictions**    The value of *numeric-expression* must be greater than or equal to +1, or less than or equal to -1. Values outside of this domain cause **ACOS** to return a meaningless value.

**See also**        Other trigonometric functions, **OPTION** statement

### Examples

```
10  OPTION PROMPT " ", DEGREE
20
30  INPUT "Enter cosecant of an angle: ",VALUE
40  PRINT
50
60  MY.ANGLE = ACSC(VALUE)
70  MY.ANGLE.MIN = FP(MY.ANGLE)*60 ! Determine minutes
80  MY.ANGLE.SEC = FP(MY.ANGLE.MIN)*60 ! and seconds
90
100 MY.ANGLE$ = STR$(IP(MY.ANGLE))&CHR$(248)&
      STR$(IP(MY.ANGLE.MIN))&"' "&
      STR$(ROUND(MY.ANGLE.SEC,4))&" "
120 PRINT USING "The arccosecant of 'e is 'e",STR(VALUE),MY.ANGLE$
```

**Enter cosecant of an angle: 5.98**

**The arccosecant of 5.989 is 9°37'35.2569"**

---

## ACTIVATE Function

ACTIVATE starts a subtask and returns the partition number of the started subtask.

- 1 **ACTIVATE**
  - 2 **ACTIVATE**( *task-name* )
  - 3 **ACTIVATE**( *task-name*, *priority* )
  - 4 **ACTIVATE**( *task-name*, *priority*, *data-segment-length* )

### Operation

**Mode 1**—Starts a copy of the current program as a subtask to the current program. This process is called *forking*. The subtask inherits a copy of the current program's data segment along with all of its opened files (except files opened to spooled printers) and device attachments. The data segment includes all of the variables used by the current program.

The subtask has the same priority as the current task. Execution in the subtask continues with the statement following the ACTIVATE function reference. In the current task the ACTIVATE function returns the partition number of the activated subtask; in the subtask the ACTIVATE function returns a negative one.

The activated program has all events disabled. This includes all [ON KEY](#), [ON EVENT](#), [ON ERROR](#) and [ON MOUSE](#) events.

**Mode 2**—Starts the task, named *task-name*, as a subtask to the current program. If *task-name* is the null string, this mode is identical to [Mode 1](#).

**Mode 3**—Starts the task, named *task-name*, as a subtask to the current program. If *task-name* is the null string, this mode is identical to [Mode 1](#), except that the priority of the subtask is set to *priority*.

**Mode 4**—Identical to [Mode 3](#) except that a *data-segment-length* parameter is allowed for upward compatibility from prior versions of THEOS BASIC. The *data-segment-length* value is ignored as the length of data segments is dynamically allocated with MultiUser BASIC.

When **Mode 1** of the **ACTIVATE** function is used, it is common to use an **IF** statement immediately following, so that the two tasks can begin divergent execution. For example:

```
SUBTASK% = ACTIVATE
IF SUBTASK%           ! In subtask?
  GOTO SUBTASK.ACTIVITIES
IFEND
```

Do not use the **CHAIN**, **LINK** or **RUN** statements in the subtask as those statements will terminate execution of the subtask and all subtasks subordinate to this task.

The partition used for the activated subtask will be the last unused partition number available. Partitions are created with the **SYSGEN** utility command "Maximum number of tasks" parameter. Refer to the *THEOS System Reference Manual* for a description of this command. For example:

THEOS Show Users							
Pid	Username	Programe	Status & Info	Pid	Username	Programe	Status & Info
1	SYSTEM	SHOW	Z	11			X Stopped
2		LOGON	I	12		LOGON	I
3		LOGON	I	13			X Stopped
4		LOGON	I	14			X Stopped
5		LOGON	I	15			X Stopped
6		LOGON	I	16			X Stopped
7		LOGON	I	17			X Stopped
8			X Stopped	18	SPOOLER	DESPOL	I Task of 19
9			X Stopped	19	SPOOLER	DESPOL	I
10			X Stopped	20	CACHE	SYNC	Z

Assuming the above display reflects the current user status, a subtask started with the **ACTIVATE** function would assign the new subtask to partition number 17.

A zero is returned when the subtask cannot be started .

## Restrictions

Priority levels for programs and tasks are integer values in the range of 0 to 7. Specifying a -1 priority indicates that the subtask is to receive the same priority level as the current program's priority.

Refer to the *Programmer's Guide* for a discussion of multitasking programming.

The **ACTIVATE** function may not be used in the interpreter. Only compiled programs may be multitasking. The error message "Multitasking not available in MultiUser BASIC Interpreter." is reported when an attempt is made to use the **ACTIVATE** function in an interpreted program.

**Examples**

*This simple program merely accepts a text field from the operator and then displays that field.*

```
10 OPTION PROMPT "", CASE "M"
20
30 WINDOW OPEN 1,1,2,80,21; SELECT
40
```

*A subtask is activated.*

```
50 SUB.TASK% = ACTIVATE("SUBTASK")
60
70 LOOP:
80
90 PRINT AT$(1,5);"Enter any text: ";
100 LINPUT USING SPACE(50),ANY.TEXT$
110 PRINT AT$(1,10);CRT$("EOL");
120 PRINT ANY.TEXT$
130
140 GOTO LOOP
```

*This simple subtask displays the time of day on the screen, once per minute.*

Program: SUBTASK

```
10 TIMER% = SEMAPHORE("MINUTE")
20
30 TIMER TIMER% SYNC 60 ! 1 minute times
40
50 WAIT.TIMER:
60
70 WAIT EVENT(TIMER%)
80
90 WINDOW STATUS PRIOR.WIN%
100
110 WINDOW SELECT 0, UPDATE OFF
120
130 PRINT AT$(72,1);TIME$(0)
140
150 TIMER TIMER% SYNC 5
160
170 WINDOW SELECT PRIOR.WIN%
180
190 GOTO WAIT.TIMER
200
210 END
```

*This program should use some kind of signaling semaphore to communicate with the main task to make sure that they are not both writing to the screen at the same time. But, because separate windows are used (main task using window 1, subtask using window 0) there should be no conflict.*

---

## ADDROF Function

ADDROF returns the address of a variable, array, file channel or subprogram.

**ADDROF**(*variable-name* )

**ADDROF**( # *file-channel* )

**ADDROF**( SUB *subprogram-name* )

<b>Operation</b>	The memory address of <i>variable-name</i> , <i>file-channel</i> or <i>subprogram</i> is determined and returned.
<b>Notes</b>	<p>The first form of this function is used in the <a href="#">WINDOW CHOICE</a> and <a href="#">WINDOW OPEN</a> statements when you want the window number to be assigned. All forms of the function can be used when calling C language functions from MultiUser BASIC.</p> <p>Since the interface to the C language function is always “call by value,” the C function cannot modify the variables in the calling program. With the ADDROF function, the value passed to the C routine is the address of the variable in memory. This value can then be used by the C routine to modify the variable in the calling program.</p> <p>Passing the ADDROF(#<i>file-channel</i>) allows the C routine to use a file opened by the MultiUser BASIC program. The syntax ADDROF(SUB <i>subprogram</i>) provides the C routine with the address of the start of a subprogram in the MultiUser BASIC program.</p>
<b>Restrictions</b>	Except for the first syntax ( ADDROF( <i>variable-name</i> ) ) the ADDROF function may only be used in the <a href="#">CALL</a> statement.
<b>See also</b>	<a href="#">CALL</a> , <a href="#">WINDOW CHOICE</a> , <a href="#">WINDOW OPEN</a>

## Examples

*A C language function is called giving it the location of a variable used as a return value.*

```
80230 CALL MY.FUNCT(ADDROF(RET.VALUE%), INP.VALUE%)
```

*A window is opened allowing Session Manager to assign the window number to WIN%.*

```
80520 WINDOW OPEN ADDROF(WIN%), 10, 5, 60, 3; SELECT
```

```
90000 CALL BWPROCEDURE(BUTTON.OBJECT%, WEVENT.CLICK%,  
ADDROF(SUB EVENT.HANDLER), 0)
```

---

## ANGLE Function

ANGLE returns the *arctangent* of the ratio between two tangents.

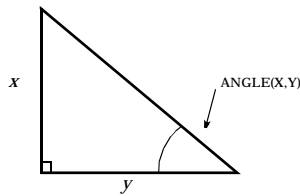
ANGLE( *numeric-expression*<sub>1</sub>, *numeric-expression*<sub>2</sub> )

**Operation**      The numeric expressions are evaluated and interpreted as the length of the two legs of a right triangle. The arctangent of *numeric-expression*<sub>1</sub> divided by *numeric-expression*<sub>2</sub> is returned.

**Notes**            The angle returned is expressed in degrees or radians, depending upon the status of the current [OPTION DEGREE](#) or [OPTION RADIAN](#) statement.

In traditional, trigonometric notation, the ANGLE function is expressed as:

$$\text{ANGLE}(X,Y) = \text{atan}\left(\frac{X}{Y}\right)$$



**Restrictions**      *numeric-expression*<sub>2</sub> must not be zero or a meaningless value is returned.

**See also**            Other trigonometric functions, [OPTION](#) statement

### Examples

```
10 OPTION DEGREE
20
30 PRINT "The two angles of a 3-4-5 right triangle are: ";
40 PRINT USING "##.# and ##.#",ANGLE(3,4),ANGLE(4,3)
```

The two angles of a 3-4-5 right triangle are: 36.9 and 53.1



---

# ASC Function

ASC returns the ASCII code value of a single character.

ASC( *string-expression* )

**Operation**            The ASCII code value of the first character in the *string-expression* is returned.

**Notes**                Refer to Appendix B: “[THEOS Character Set](#),” starting on page [638](#) for a table of the ASCII code values.

**See also**             [CHR\\$](#), [ORD](#)

## Examples

<i>This subroutine is used to accept the operator's menu selection character. A timer is used to force selection within a specified amount of time. The character entered is saved in an integer variable.</i>	98100 GET.SEL: ! Get menu selection response 98110 98120 ON EVENT(MENU.TIMER%) GOTO INPUT.TIME.OUT 98130 TIMER MENU.TIMER% ELAPSED WAITTIME% 98140 98150 SEL% = 0 \ TIME.OUT% = FALSE% 98160 98170 WAIT #0 \ GET SEL% 98180 98190 ON EVENT(MENU.TIMER%) GOTO 0 ! Disable time out 98200 98210 IF TIME.OUT% THEN SEL% = 128 98220
<i>Both the character entered (SEL\$) and the value of the character entered (SEL%) are saved for easy comparisons later in the program.</i>	98230 SEL\$ = UCASE\$(CHR\$(SEL%)) \ SEL% = <b>ASC(SEL\$)</b> 98240 98250 RETURN

Statements

---

## ASEC Function

ASEC returns the *arcsecant* of a number.

**ASEC**( *numeric-expression* )

**Operation**      The *numeric-expression* is evaluated and interpreted as the secant of an angle. That angle is computed and returned. The angle returned is expressed in degrees or radians, depending upon the status of the current [OPTION](#) DEGREE or [OPTION](#) RADIAN statement.

**Notes**            Arcsecants have values that range from 0 to  $\pi/2$  radians (0° to +90°).  
  
Refer to the [ACOS](#) function for a description of inverse trig relationships.

**Restrictions**    The value of *numeric-expression* must be greater than or equal to +1, or less than or equal to -1. Values outside of this domain cause ASEC to return a meaningless value.

**See also**        Other trigonometric functions, [OPTION](#) statement

**Examples**

```
10  OPTION PROMPT "", DEGREE
20
30  INPUT "Enter secant of an angle: ",VALUE
40  PRINT
60  MY.ANGLE = ASEC(VALUE)
70  MY.ANGLE.MIN = FP(MY.ANGLE)*60 ! Determine minutes
80  MY.ANGLE.SEC = FP(MY.ANGLE.MIN)*60 ! and seconds
90
100 MY.ANGLES$ = STR$(IP(MY.ANGLE))&CHR$(248)&
      STR$(IP(MY.ANGLE.MIN))&"' "&
      STR$(ROUND(MY.ANGLE.SEC,4))&"'"
110
120 PRINT USING "The arcsecant of 'e is 'e",
      STR(VALUE),MY.ANGLES$
```

**Enter secant of an angle: 1.06548**

**The arcsecant of 1.06548 is 20°11'29.4646"**

---

## ASIN Function

ASIN returns the *arcsine* of a number.

**ASIN**( *numeric-expression* )

**Operation**      The *numeric-expression* is evaluated and interpreted as the sine of an angle. That angle is computed and returned. The angle returned is expressed in degrees or radians, depending upon the status of the current [OPTION DEGREE](#) or [OPTION RADIAN](#) statement.

**Notes**            Arcsines have values that range from  $-\pi/2$  to  $\pi/2$  radians ( $-90^\circ$  to  $+90^\circ$ ).  
Refer to the [ACOS](#) function for a description of inverse trig relationships.

**Restrictions**    The value of *numeric-expression* must be in the domain of  $-1$  to  $+1$ . Values outside of this domain cause ASIN to return meaningless values.

**See also**        Other trigonometric functions, [OPTION](#) statement

### Examples

```
10 OPTION PROMPT "", DEGREE
20
30 INPUT "Enter sine of an angle: ",VALUE
40 PRINT
50
60 MY.ANGLE = ASIN(VALUE)
70 MY.ANGLE.MIN = FP(MY.ANGLE)*60 ! Determine minutes
80 MY.ANGLE.SEC = FP(MY.ANGLE.MIN)*60 ! and seconds
90
100 MY.ANGLE$ = STR$(IP(MY.ANGLE))&CHR$(248)&
      STR$(IP(MY.ANGLE.MIN))&"' "&
      STR$(ROUND(MY.ANGLE.SEC,4))&"'"
110
120 PRINT USING "The arcsine of 'e is 'e",STR(VALUE),
      MY.ANGLE$
```

Enter sine of an angle: .74314

The arcsine of .74314 is 47°59'58.5125"

## Statements

**AT\$( column, row )**

**Notes** The screen origin is at the upper left corner of the screen or window. Columns and rows are numbered from one through the attached width and depth of the screen or size of the window.

<b>Restrictions</b>	The <i>column</i> and <i>row</i> values must be less than or equal to the width and depth of the current console screen or window. When either of these values is too large, a null string is returned.
---------------------	---

**See also** [CRT\\$](#)

Display two field labels in the middle of the screen.

```
901214 PRINT AT$(15,10); "Operator: "; AT$(39,10);
          "Password: ";
```

Display program	900730	PRINT	AT\$(2,1);FROM.PROG\$;
title information on	900740	PRINT	AT\$(25,1);"SCREEN APPEARANCE MAINTENANCE";
line 1 of the screen.	900750	PRINT	AT\$(71,1);DATE\$(0);

---

## ATAN, ATN Functions

ATAN returns the *arctangent* of a number. ATN is a synonym to ATAN.

1 **ATAN**( *numeric-expression* )

2 **ATN**( *numeric-expression* )

**Operation**      The *numeric-expression* is evaluated and interpreted as the tangent of an angle. That angle is computed and returned. The angle returned is expressed in degrees or radians, depending upon the status of the current **OPTION DEGREE** or **OPTION RADIAN** statement.

**Notes**            Arctangents have values that range from  $-\pi/2$  to  $+\pi/2$  radians ( $-90^\circ$  to  $+90^\circ$ ).

Refer to the **ACOS** function for a description of inverse trig relationships.

**See also**        Other trigonometric functions, **OPTION** statement

### Examples

```
10 OPTION PROMPT "", DEGREE
20
30 INPUT "Enter tangent of an angle: ",VALUE
40 PRINT
50
60 MY.ANGLE = ATAN(VALUE)
70 MY.ANGLE.MIN = FP(MY.ANGLE)*60 ! Determine minutes
80 MY.ANGLE.SEC = FP(MY.ANGLE.MIN)*60 ! and seconds
90
100 MY.ANGLE$ = STR$(IP(MY.ANGLE))&CHR$(248)&
      STR$(IP(MY.ANGLE.MIN))&"' "&
      STR$(ROUND(MY.ANGLE.SEC,4))&"'"
110
120 PRINT USING "The arctangent of 'e is 'e",
      STR(VALUE),MY.ANGLE$
```

Enter tangent of an angle: 1.25

The arctangent of 1.25 is 51°20'24.6903"

---

## AVAIL.WINDOWS Function

AVAIL.WINDOWS returns the number of currently available or unopened windows.

AVAIL.WINDOWS

**Operation**      The number of unopened windows.

**Notes**               $AVAIL.WINDOWS = TOTAL.WINDOWS - USED.WINDOWS$

Since [TOTAL.WINDOWS](#) includes window 0 and window 0 is always open, AVAIL.WINDOWS will never equal [TOTAL.WINDOWS](#).

**See also**          [TOTAL.WINDOWS](#), [USED.WINDOWS](#)

**Examples**

```
60000  HELP.DISPLAY:
60010
60020      IF AVAIL.WINDOWS<1 THEN RETURN ! Window available?
60030
60040      WINDOW OPEN ADDR OF(HELP.WIN%), 10, 5, 60, 12 ! Yes, open it
60050
.60060      ...
60970
60980      RETURN
```

---

## BIN, BINOF\$ Functions

BIN returns the numeric value of a binary character string.

BINOF\$ returns a binary character string of a numeric expression.

**BIN**( *string-expression* )

**BINOF\$**( *integer-expression* )

**Operation**      BIN analyzes the *string-expression* looking for binary digits (0 and 1). When the first non-binary digit is found, analysis stops and the string of digits is evaluated as a numeric value expressed in base 2.

BINOF\$ converts the *integer-expression* to a sixteen-digit binary character string. For instance, BINOF\$(43) returns "0000000000101011".

**Notes**            The BIN and BINOF\$ functions are complements of each other:

$$A = \text{BIN}(\text{BINOF\$}(A))$$

**Restrictions**    Only integer values are returned (−32768 to +32767).

Only the last sixteen binary digits of *string-expression* are used:

$$\text{BIN}("111100001111000011111111") \Leftrightarrow \text{BIN}("1111000011111111")$$

## Examples

The following code might be used to build a bit-mapped field holding a customer's monthly activity status. Refer to the LRL and LRR functions for an example showing how this type of field might be displayed.

Refer to the [HEX](#), [HEXOF\\$](#) function for an example using the BINOF function.

*Build an array of yes/no values for the prior 16 months.*

*Build a string containing the binary representation of the 16 month bits.*

*Use the **BIN** function to convert the binary string to a numeric value. This value can then be saved in a disk file, using only three bytes of storage.*

```
1000 FOR I% = 1 TO 16
1010     INPUT "Customer active in month "&STR$(I%),ACTIVE$(I%)
1020     NEXT
1030
1040 CUST.ACTIVE$ = "" ! Clear active field
1050
1060 FOR I% = 1 TO 16
1070     IF ACTIVE$(I%)="Y"
1080         CUST.ACTIVE$ = CUST.ACTIVE$&"1"
1090     ELSE CUST.ACTIVE$ = CUST.ACTIVE$&"0"
1100     IFEND
1110 NEXT
1120
1130 CUST.ACTIVE% = BIN(CUST.ACTIVE$)
~
```



---

## BREAK Statement

BREAK exits from the current program structure.

### BREAK

**Operation** The current, enclosing [DEF FN-FNEND](#), [FOR-NEXT](#), [SELECT-CEND](#), [SUB-END SUB](#), or [WHILE-WEND](#) program structure is exited. The statement executed is the statement following the program structure termination statement ([FNEND](#), [NEXT](#), [CEND](#), [END SUB](#) or [WEND](#)).

**Notes** The BREAK statement is an unconditional control transfer statement. For example:

```
10 FOR I% = 1 TO 10
20   PRINT I%
30   BREAK
40   ITEM%(I%) = ITEM%(I%)+1
50   NEXT
60 REM Next statement
```

The above loop will only be executed one time and lines 40 and 50 will never be executed. Because BREAK is an unconditional jump, it is normally used in an IF statement:

```
10 FOR I% = 1 TO 10
20   PRINT I%
30   IF ITEM%(I%)>20 THEN BREAK
40   ITEM%(I%) = ITEM%(I%)+1
50   NEXT
60 REM Next statement
```

**Restrictions** Specifying a BREAK statement outside of a [FOR-NEXT](#), [SELECT-CEND](#), [SUB-END SUB](#) or [WHILE-WEND](#) program structure causes an error message “BREAK without DEF, FOR, SELECT, SUB or WHILE at line ...” when the program is attempted to be executed by the interpreter or when it is compiled.

**See also** [CEND](#), [DEF FN](#), [END SUB](#), [FNEND](#), [FOR](#), [NEXT](#), [SELECT](#), [SUB](#), [WEND](#), [WHILE](#)

### Examples

*Upon loop exit, the index (SI%) will be the index of the match or it will be greater than 51.*

```
601100 FOR SI% = 1 TO 51
601110   IF TRIM$(FIELD$)=STATE.CODES$(SI%)
601120     BREAK
601130   IFEND
601140 NEXT
601150
```

---

## CALL Statement

CALL invokes a subprogram or a THEOS C language function.

1 **CALL** *procedure-name*

2 **CALL** *procedure-name*( *argument-list* )

---

*procedure-name*   »   *C-function-name*

                  »   *subprogram-name*

*argument-list*   »   *expression*[, *argument-list*]

                  »   **ADDROF**( *variable-name* )[, *argument-list*]

                  »   **ADDROF**( **#***integer-expression* )[, *argument-list*]

                  »   **ADDROF**( **SUB** *subprogram-name* )[, *argument-list*]

                  »   **DIM** *array-name*( *subscript-variable* )

                              [, *argument-list*]

                  »   **DIM** *array-name*( *subscript-variable*<sub>1</sub>,  
  *subscript-variable*<sub>2</sub> )[, *argument-list*]

### Operation

**Mode 1**—Call a C language routine or a subprogram that requires no arguments.

**Mode 2**—Call a C language routine or a subprogram giving it one or more arguments.

#### ■ Passing Subprogram Names

A C language function can be passed the location of the start of a subprogram. Only an advanced function could make use of this capability. One such function is the BWPROCEDURE function in the NWM toolkit. It needs to know the location of a subprogram so that it may program the system to invoke that subprogram when a specific event occurs.

To pass the location of a subprogram to a C function use the syntax **ADDROF**(**SUB** *subprogram-name*).

---

# CALL.RETURN.VALUE Function

The CALL.RETURN.VALUE function returns the value set by the last C language function called by this program.

CALL.RETURN.VALUE

**Operation**      The value set by the last C language function executed by this program is returned as the value of this function.

**Notes**            If the C language function did not set a return value the value of the CALL.RETURN.VALUE is undefined.

                      Although subprograms are executed with the [CALL](#) statement they do not set the CALL.RETURN.VALUE. Only C language functions can set this value.

**Restrictions**    The C language function must be programmed to return an integer value. If the function doesn't return a value or specifically returns a void the CALL.RETURN.VALUE will be undefined and will have no meaning.

**See also**         [CALL](#)

**Examples**        Refer to the *MultiUser BASIC Programmer's Guide* chapter on "Using C Language Routines" for example usage.

Statements

---

## CASE Statement

The CASE statement is used as part of a [SELECT-CEND](#) programming structure. CASE marks the beginning of a series of statements and specifies the condition that, when true, causes the series of statements to execute. When the condition is false the statements are not executed.

- 1 **CASE** *relational-expression*
  - 2 **CASE** *expression*

### Operation

**Mode 1**—This form is used in conjunction with the mode 1 form of the SELECT statement. Since the SELECT statement did not specify any expression, the CASE statement must specify the complete *relational-expression* needed for the true/false test.

For example:

```
1000 SELECT
1010     CASE DIFF%=1 OR (DIFF%>20 AND MOD(DIFF%,10)=1)
           statements performed when case is true
1020     CASE DIFF%=2 OR (DIFF%>20 AND MOD(DIFF%,10)=2)
           statements performed when case is true
1030     CASE DIFF%=3 OR (DIFF%>21 AND MOD(DIFF%,10)=3)
           statements performed when case is true
1040     OTHERWISE
           statements performed when no case is true
1050     CEND
```

With this mode of the CASE statement, the *relational-expression* for this CASE statement may use any of the relational operators and the left and right portions of the expression may be different from the other CASE statements in this program structure.

**Mode 2**—This form is used in conjunction with the mode 2 form of the [SELECT](#) statement. Since the [SELECT](#) statement specified the left portion of the relational expression the CASE statement need only supply the right portion.

The complete relational expression is formed by using the [SELECT](#) statement's *expression* and the CASE statement's *expression*, using the "is equal to" relational operator.

For example:

```
800150 SELECT FRAME.STYLE%
800160     CASE 0
800170         WINDOW FRAME 5 ! Remove frame
800180     CASE 1
800190         WINDOW FRAME 5, SINGLE ! Single line frame
800200     CASE 2
800210         WINDOW FRAME 5, DOUBLE ! Double line frame
800220     CEND
```

When the complete relational expression is true, the statements following this CASE statement are executed. When another CASE or [OTHERWISE](#) statement is encountered, control is transferred to the closing [CEND](#) statement.

#### Notes

CASE statements are evaluated in the sequence specified in the program structure. When the result is false (0), the statements following the CASE are bypassed until another CASE, [OTHERWISE](#), or [CEND](#) statement is encountered.

When the result of a CASE statement is true, the statements following the CASE statement are executed. When the next CASE or [OTHERWISE](#) statement is encountered, control is transferred to the closing [CEND](#) statement.

When all of the CASE statements have evaluated false and an [OTHERWISE](#) statement is encountered, the statements following that [OTHERWISE](#) are executed. After executing these statements and another CASE or [OTHERWISE](#) statement is encountered, control is transferred to the closing [CEND](#) statement.

#### Restrictions

In mode 2, the *expression* must match the data type (string or numeric) of the *expression* used in the [SELECT](#) statement.

#### See also

[CEND](#), [OTHERWISE](#), [SELECT](#)

## Examples

Determine the starting display line of a group of items. There is no OTH-ERWISE statement because the range of possibilities is only 1-7.

```

~
3530      SELECT ITEM.COUNT%
3540          CASE 1 \ ITEM.LINE% = 8
3550          CASE 2 \ ITEM.LINE% = 7
3560          CASE 3 \ ITEM.LINE% = 7
3570          CASE 4 \ ITEM.LINE% = 6
3580          CASE 5 \ ITEM.LINE% = 5
3590          CASE 6 \ ITEM.LINE% = 5
3600          CASE 7 \ ITEM.LINE% = 4

```

The SELECT structure allows only one of these statements to execute.

```

3610      CEND
~

```

This code is used following the operator's response for a record key request. The INP function is tested to see if special control keys were used to terminate the entry.

```

~
10140     SELECT
10150         CASE INP=10 ! Down arrow = next record
10160             IF KEY$ THEN READ #1,KEY$: X$
10170             GOSUB READ.NEXT.CUSTOMER
10180         CASE INP=11 ! Up arrow = prev record
10190             IF KEY$ THEN READ #1,KEY$: X$
10200             GOSUB READ.PREV.CUSTOMER
10210         CASE (INP=13 OR INP=0) AND KEY$<>" " ! Read rec
10220             REC$(1) = KEY$
10230             GOSUB READ.CUSTOMER
10240         OTHERWISE
10250             PRINT CRT$("BELL");
10260             VALID% = FALSE%
10270     CEND
~

```

---

## CEIL Function

CEIL returns the ceiling of a numeric number.

**CEIL**( *numeric-expression* )

**Operation**      The ceiling of *numeric-expression* is determined and returned.

**Notes**            The ceiling of a number is the smallest whole number that is greater than or equal to the number.

The ceiling of a whole number is the number itself.

The ceiling of positive fractional numbers is equal to

$$\text{IP}(\text{numeric-expression}) + 1$$

The ceiling of negative fractional numbers is equal to

$$\text{IP}(\text{numeric-expression})$$

**See also**        [FLOOR](#)

### Examples

*This routine computes the number of pages required for a print job. CEIL is used because any fraction of a page requires an entire page.*

*Note the use of the FLOAT function.*

*This is necessary because the normal result of an operation on two integers is an integer.*

```
10 INPUT "Number of lines of text",LINE.COUNT%
20 INPUT "Number of lines per page",LINES.PER.PAGE%
30
40 PRINT "Number of pages required: ";
50 PRINT CEIL(FLOAT(LINE.COUNT%)/LINES.PER.PAGE%)
60
```

---

# CEND Statement

The CEND statement is used as part of a [SELECT-CEND](#) programming structure. CEND marks the end of the selection statement groups of the programming structure.

CEND

**Operation**      End a “select one of” program structure. Conditional execution of [CASE](#) or [OTHERWISE](#) statement groups is terminated and normal execution of statements is resumed.

**Notes**            Every [SELECT](#) statement must have a single, closing CEND statement.

**See also**         [CASE](#), [OTHERWISE](#), [SELECT](#)

## Examples

Statements

*After the operator terminates input to a field, check to see if a control key was used.*

```
10500 IF INP
10510     SELECT INP
10520         CASE 11             ! Up arrow
10530             IF I%>1 THEN IJ% = I%-2 ELSE IJ% = 99
10540         CASE 10             ! Down arrow
10550             IJ% = I%
10560         CASE 13             ! Return
10570             IJ% = I%
10580         CASE 25             ! End key
10590             IJ% = 99
10600         OTHERWISE          ! Control key invalid here
10610             PRINT CRT$("BELL");
10620             IJ% = I%-1
10630         CEND
10640     IFEND
~
```

*Each CASE statement tests for a particular control key that is acceptable for this field.*

*The OTHERWISE statement specifies the action when none of the CASE statements have been true.*

*CEND marks the end of SELECT.*



---

## CHAIN Statement

CHAIN closes all files and transfers control to another MultiUser BASIC program.

**CHAIN** *program-name-expression*

<b>Operation</b>	All open files are closed and the program designated by <i>program-name-expression</i> is loaded and execution resumes with the first statement in that program.
<b>Notes</b>	When executed in the interpreter, the chained-to program may have a file-type of BASIC or BAS.
<b>See also</b>	<a href="#">CLEAR</a> , <a href="#">CLOSE</a> , <a href="#">COMMON</a> , <a href="#">CSI</a> , <a href="#">LINK</a> , <a href="#">RUN</a>

---

# CHR\$ Function

CHR\$ returns the character corresponding to the value of an expression.

CHR\$( *numeric-expression* )

Operation	The character whose ASCII value is <i>numeric-expression</i> is returned.
Notes	<p>The CHR\$ function is frequently used for printing control characters or for testing for control character input.</p> <p>Refer to Appendix B: “<a href="#">THEOS Character Set</a>,” starting on page <a href="#">638</a> for a table of ASCII values.</p>
Restrictions	The <i>numeric-expression</i> should be a positive value less than 256. Values outside of this range are modularized to this range. For example, the CHR\$(268) is the same as the CHR\$(12).
See also	<a href="#">ASC</a> , <a href="#">ORD</a>

## Examples

Statements

<i>Here, a character accepted from the operator is compared with certain, key character codes.</i>	791060	CASE IN.CHAR\$=CHR\$(8) ! Backup
	791070	IF IN.POS%>1
	791080	IN.POS% = IN.POS%-1
	~	
	791120	ELSE PRINT CRT\$("BELL");

<i>The CHR\$ function is used to generate the character value of the characters that cannot be easily specified in a string.</i>	791130	CASE IN.CHAR\$=CHR\$(13) OR IN.CHAR\$=CHR\$(9)
	~	

---

# CLEAR Statement

CLEAR erases the contents of variables and arrays.

**CLEAR** *variable-name*[,*variable-name*] ...

**Operation**            The contents of the *variable-names* are set to nulls.

**Notes**                Clearing a numeric variable or array sets the variable or the elements of the array to 0. Clearing a string variable or array sets the variable or the elements of the array to "" (an empty, zero length, string).

                          The amount of storage space is not changed when the *variable-name* is a numeric variable or array. Clearing a string variable or array does free up the memory used for the prior contents of the strings.

**Restrictions**        The CLEAR statement cannot be used to clear specific elements of an array.

**See also**             [ASC](#), [ORD](#)

## Examples

<i>The program using this code used some common data that was defined by another program.</i>	9000	RETURN.TO.CALLER:
	9010	
	9020	GOSUB CLOSE.FILES ! Close files opened by program
	9030	GOSUB CLOSE.WINDOWS ! Close windows
	9040	
<i>Since this program is through with the data, it is cleared to make sure that no other program can use the information by mistake.</i>	9048	REM Clear common variables used just for this program
	9049	
	9050	<b>CLEAR</b> PRT.RESET\$,PRT.ATTR\$,REPORT.TITLE\$
	9060	
	9070	LINK TO.PROG\$ ! Return to calling program

# CLOSE Statement

The CLOSE statement closes an open file or input/output channel.

CLOSE #channel

**Operation** The open I/O file or device on *channel* is closed.

**Notes** The process of closing a file I/O channel includes:

- ▶ Flushing any data remaining in the output buffer
- ▶ Synchronizing the disk cache with the actual disk
- ▶ Unlock the file and any records in the file locked by this program
- ▶ Update the disk directory if the information has changed
- ▶ Release the channel number for reuse by this program

The process of closing files is performed automatically by the following statements: CHAIN, CSI, END, QUIT and RUN. The LINK and SYSTEM statements do not close any files.

**Restrictions** The *channel* must be an integer value in the range 1–999. Attempting to close a channel outside of this range generates a trappable error number 38 “Invalid file channel ... out of range.”

**See also** CHAIN, CSI, END, OPEN, QUIT, RUN, WINDOW CLOSE

**Examples**

```
6470      IDX% = 1

6480  FILENAME$ = SYSTEM$&".SYSTEM.FORMS"
6490  OPEN #1:FILENAME$, INPUT INDEXED
6500
6510  READNEXT #1,K$: FORMS$(IDX%),X,X,FORM.QUEUE.IDX%(IDX%)
6520
6530  WHILE NOT EOF(1) AND IDX%<26
6540      IDX% = IDX%+1
6550      READNEXT #1,K$: FORMS$(IDX%),X,X,FORM.QUEUE.IDX%(IDX%)
6560      WEND
6570

6580  CLOSE #1
```

*A data file is opened and the first record is read.*

*The remaining records in the file are read until the end of the file is reached.*

*Because the program is finished with the file, it is closed.*

180 CLOSE

---

# CLS\$ Function

CLS\$ returns the character that, when displayed on the console, clears the screen or window.

CLS\$

**Operation**            The form-feed character is returned.

**Notes**                The form-feed character is ASCII value 12. When it is output to the console it clears a console screen or window. When output to a printer it ejects a page.

                         This function does not clear the screen itself. Only when the function is printed to the screen is the screen actually cleared.

**See also**             [CRT\\$](#)

<b>Examples</b>	901080	REM Define title line area of Window 0
	901090	
<i>Select the full screen window but do not display.</i>	901100	WINDOW SELECT 0, UPDATE OFF
<i>Set the desired colors and then clear the window to those colors.</i>	901110	COLOR FG%,BG%,-1,-1
	901130	PRINT <b>CLS\$</b> ;
	901140	PRINT AT\$(2,1);FROM.PROG\$;
<i>Display program title line information.</i>	901150	PRINT AT\$((LINE(0)-LEN(CONAME\$))/2,1);CONAME\$;
	901160	~

Statements

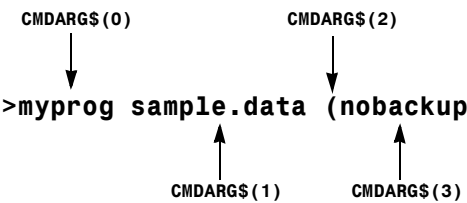
# CMDARG\$ Function

CMDARG\$ returns a command line argument specified when the program was invoked.

CMDARG\$( *argument-number* )

**Operation**      The command line used to invoke the execution of this program is analyzed and the parameter indexed by *argument-number* is returned.

**Notes**            Command line arguments are numbered from 0 with argument number 0 matching the program name parameter.



Interpretive programs can have command line arguments. Refer to the [RUN](#) command for details on specifying them.

The [SYS.ENV\\$](#) function may be used to retrieve and set EXEC command line arguments.

**See also**        [RUN](#) command, [SYS.ENV\\$](#)

**Examples**       The following code section was taken from a program that can access different files, depending upon the presence of a command line argument.

*If the program was invoked with a parameter, then use that parameter for the file name; otherwise, use the default for the file name.*

```
900640 IF CMDARG$(1)=" "  
900650     CALENDAR.NAME$ = OPNAME$  
900660 ELSE CALENDAR.NAME$ = CMDARG$(1)  
900670     IFEND  
~
```

If the program was invoked with:

```
>APPOINTS SPECIAL
```

then the `CALENDAR.NAME$` variable is set to "SPECIAL", otherwise, it is set to the current value of the `OPNAME$` variable.

---

# COLOR Statement

The COLOR statement defines the display colors for normal and reverse video.

COLOR fg[, bg[, rvfg, rvbg]]

**Operation**      The colors for subsequently displayed characters is set to the codes specified.

Color	Code	Color	Code
Black	0	Red	4
Blue	1	Magenta	5
Green	2	Yellow	6
Cyan	3	White	7

Omitting the background color code, *bg*, means the background is displayed with color 0. Omitting the reverse video color codes, *rvfg* and *rvbg*, means that reverse video is the inverse of normal video. For example:

COLOR 4      ⇔      COLOR 4,0,0,4  
COLOR 7,1    ⇔      COLOR 7,1,1,7

When the display is not capable of colors, this statement has no effect.

**Notes**      The color codes do not necessarily correspond to the colors listed in the above table. The table lists the conventional colors for the codes and will be true for most displays. The actual correspondence between color codes and colors is defined by the display's class code. Refer to the CLASSGEN command and its description in the *THEOS System Reference* manual.

Some color terminal displays implement color changes via video attributes. To specify the colors for those types of displays the program will have to display a video attribute change with the CRT\$ function.

## Examples

Change to red on  
black and display  
an error message.

1050 COLOR 4,0  
1060 REPLY\$ = ERRMSG\$("File not found...create")  
1070 COLOR 7 ! Change back to white on black characters

Statements

# COMMON Statement

COMMON declares variables and arrays to be common or shared between several chained, linked, or multitasked programs. Common variables are also public variables.

**COMMON** *variable-list*

*variable-list*       »   *variable-name*[, *variable-list*]  
                          *array*[, *variable-list*]  
  
*array*                »   *array-name*( *subscript* )  
                          *array-name*( *subscript*<sub>1</sub>, *subscript*<sub>2</sub> )

**Operation**       The variables declared in the list are marked as *common variables*; arrays declared in the list are *dimensioned* and marked as *common arrays*.

**Notes**           Variables and arrays that are declared with the COMMON statement are not cleared when the program is invoked with a [CHAIN](#) or [LINK](#) from another program that also declared the variables and arrays. Conversely, all variables and arrays that are not declared with the COMMON statement in both programs are cleared.

The variables and arrays declared as common variables and arrays also have the property of public variables and arrays (see [PUBLIC](#) statement on page 439).

The sequence of the names in the *variable-list* does not matter. The sequence and position of COMMON statements in the program does not matter. All [DIM](#) and COMMON statements are compiled as if they were grouped together at the end of the program. The interpreter simulates this by scanning the entire program for [DIM](#) and COMMON statements prior to executing the first statement.

**Dimensioning:**   Arrays may be one-dimensional or two-dimensional.

One-dimensional array: 

0	1	2	3	4
---	---	---	---	---

Two-dimensional array: 

0,0	0,1	0,2	0,3	0,4
1,0	1,1	1,2	1,3	1,4
2,0	2,1	2,2	2,3	2,4



The number of elements allocated for an array depends upon the **OPTION BASE** statement. When **OPTION BASE 0** is in effect (default), one is added to each of the dimension sizes specified to account for the zero indices. For example:

```
OPTION BASE 0
COMMON A(5)           ! 6 elements allocated
COMMON B(12)          ! 13 elements allocated
COMMON C(2,4)         ! 15 elements allocated

OPTION BASE 1
COMMON A(5)           ! 5 elements allocated
COMMON B(12)          ! 12 elements allocated
COMMON C(2,4)         ! 8 elements allocated
```

The number of elements allocated to a **COMMON** array in one program is retained throughout all of the programs in the linked or chained set of programs. However, each program may dimension the array differently than the prior program. For instance:

First program:

```
OPTION BASE 1
COMMON A$(20),B(2,100)
```

Second program:

```
OPTION BASE 1
COMMON A$(10,2), B(100)
```

The second program redimensioned the arrays. The **B** array is redimensioned smaller than the first program's and will only receive the first 100 elements of the first program's array. The remainder of the array is discarded. If the second program links back to the first program, these discarded elements are not recovered.

## Restrictions

To retain the value of a common variable, every program in a chained or linked set of programs must declare the variable as **COMMON**. Executing a program that does not declare a variable as **COMMON** causes the value for that variable to be discarded.

Each program in a set of linked or chained programs should use the same **OPTION BASE** value. When different **BASE** values are used in a set of programs the references to the elements of **COMMON** arrays changes. For instance, if one program uses **OPTION BASE 1** and sets the fifth element of an array, a second program using **OPTION BASE 0** will have to access that element with an index of 4. The change required for two-dimensional arrays is more complex.

A variable declared as COMMON cannot have been declared with [DIM](#) or [SHARED](#) in the same program. The COMMON declaration is used instead of the [DIM](#) or [SHARED](#) declarations.

**See also** [CHAIN](#), [DIM](#), [GET COMMON](#), [LINK](#), [LOCAL](#), MAT statements, [OPTION](#) BASE, [PUBLIC](#), [PUT COMMON](#), [REDIM](#), [RUN](#), [SHARED](#), [STATIC](#)

**Examples**

```
900060  COMMON  FROM.PROG$           ! Name of invoking program
900070  COMMON  PARAM.FLAG%          ! Variable passed to program
900080  COMMON  FILENAME$            ! Name of file being accessed
900090  COMMON  P$(30)               ! Application personality array
900100  COMMON  BELL$
900110  COMMON  ATTRIB$(26),TERM$(10) ! Terminal abilities
900120  COMMON  ERR.NBR%             ! Error number
```

# CONSTANT Declaration

Define names for commonly used constant values.

**CONSTANT** *constant-name* = *constant-literal*

---

*constant-name*       »   Name of constant

*constant-literal*    »   Value of constant

**Operation**       The value of *constant-literal* is assigned to the named constant *constant-name*. During execution of the program by the interpreter, this value is substituted for every occurrence of *constant-name* in each statement of the program. During compilation of the program, this value is substituted in each statement of the program before the program is compiled. (The substitution is done on the working copy of the statement/program, not the actual source program.)

**Notes**           Like all declarations, the **CONSTANT** declaration is evaluated by the interpreter's preprocessor or by the compiler's preprocessor prior to evaluating any of the statements or functions in the program. Constant declarations may be located anywhere in the source program and the effect is the same as if they were at the start of the program source.

For a description of valid *constant-names* refer to [“Named Constants”](#) on page 31.

**Restrictions**    A named constant cannot be redefined, except with the same value as it was originally defined. This allows included files to define constants that may already be defined, as long as they use the same value.

**See also**        [LET](#)

**Examples**       1000 **CONSTANT** TRUE% = -1  
                  1010 **CONSTANT** FALSE% = 0  
                  1020 **CONSTANT** COMPANYS = "THEOS Software Corporation"  
                  1030 **CONSTANT** PROGRAMS = "SAMPLE1"  
                  1040 **CONSTANT** GROWTH = 0.3

Assign the value 0 to the DONE% variable

                  ...  
                  2000 LET DONE% = FALSE%

Statements

---

## CONTINUE Statement

CONTINUE branches to the current program structure repeat statement.

### CONTINUE

**Operation** The current, enclosing [FOR-NEXT](#) or [WHILE-WEND](#) statement is repeated. The next statement executed is the [NEXT](#) or [WEND](#) statement for the loop.

**Notes** The CONTINUE statement operates exactly like a GOTO *line-ref* where the *line-ref* is the line number or label of the loop-repeat statement. For example:

```
3490   FOR I% = 1 TO 25                      ! Next 25 records
3500       READNEXT #8,KEY$: TYPE$,RECORD$
3510       IF EOF(8) THEN BREAK
3520       IF TYPE$<>"X" THEN CONTINUE
3530       PRINT KEY$;" ": ";TYPE$;" ";RECORD$
~
3900       NEXT
```

executes exactly like:

```
3490   FOR I% = 1 TO 25                      ! Next 25 records
3500       READNEXT #8,KEY$: TYPE$,RECORD$
3510       IF EOF(8) THEN BREAK
3520       IF TYPE$<>"X" THEN GOTO EXIT.LOOP
3530       PRINT KEY$;" ": ";TYPE$;" ";RECORD$
~
3900   EXIT.LOOP:
3910       NEXT
```

The CONTINUE statement not only removes the necessity of coding the line reference desired, it provides a structured method of loop control.

Specifying a CONTINUE statement outside of a [FOR-NEXT](#) or [WHILE-WEND](#) program structure causes an error message "CONTINUE without FOR or WHILE at line ..." when the program is attempted to be executed in the interpreter or compiled.

**See also** [FOR](#), [WHILE](#)

---

## COS Function

COS returns the *cosine* of an angle.

**COS( *numeric-expression* )**

**Operation**      The *numeric-expression* is interpreted as an angle expressed in degrees or radians, depending upon the status of the current [OPTION DEGREE](#) or [OPTION RADIANT](#). The cosine of that angle is computed and returned.

**Notes**            Cosines have values that range from -1 to +1.

For non-scientific programming, the basic trigonometric functions are used to compute x and y coordinates of points on the circumference of a circle or arc.

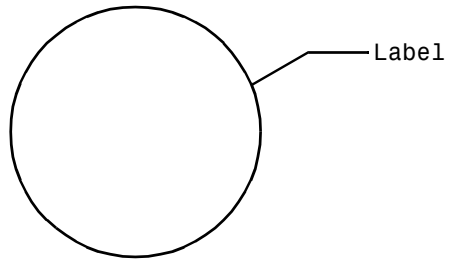
Refer to [ACOS](#) for a description of the trigonometric relationships.

**See also**        Other trigonometric functions, [OPTION](#) statement

**Examples**       The SIN and COS functions can be used to compute the x and y coordinates of a point on the circumference of a circle. For example, to draw a line from the edge of a circle to some text:

```
10 OPTION DEGREE      ! Use degrees for trig functions
20
30 PLOT CIRCLE 16000,16000;8000
40
50 X1 = 8000*COS(45)+16000      ! circumference point, x
60 Y1 = 8000*SIN(45)+16000      ! circumference point, y
70 X2 = 10000*COS(45)+16000     ! end line segment, x
80 Y2 = 10000*SIN(45)+16000     ! end line, segment, y
90
100 PLOT X1,Y1;X2,Y2;X2+5000,Y2 ! draw line
110
120 TEXT X2+5400,Y2-400;"Label" ! draw label
```

Statements



---

## COSH Function

COSH returns the *hyperbolic cosine* of a value.

**COSH**( *numeric-expression* )

**Operation**        The hyperbolic cosine of *numeric-expression* is computed and returned.

**See also**         Other trigonometric functions

**Examples**

```
10      OPTION PROMPT ""
20
30      INPUT "Enter area: ",AREA
40      PRINT
50
60      PRINT "The hyperbolic cosine of";AREA;"is";COSH(AREA)
```

**Enter area: 100**

**The hyperbolic cosine of 100 is 1.344058570907E+43**

---

## COT Function

COT returns the *cotangent* of an angle.

COT( *numeric-expression* )

**Operation**      The *numeric-expression* is interpreted as an angle expressed in degrees or radians, depending upon the status of the current [OPTION](#) DEGREE or [OPTION](#) Radian. The cotangent of that angle is computed and returned.

**Notes**            Refer to the [ACOS](#) function for a description of trigonometric relationships.

**Restrictions**    The value of *numeric-expression* must not be 0 or a multiple of  $\pi$  radians (multiple of 180°) as these values are outside this function's domain and cause COT to return meaningless values.

**See also**        Other trigonometric functions, [OPTION](#) statement

### Examples

```
10 OPTION PROMPT "", DEGREE
20
30 INPUT "Enter angle in degrees: ",MY.ANGLE
40 PRINT
50
60 MY.ANGLE.MIN = FP(MY.ANGLE)*60      ! Determine minutes
70 MY.ANGLE.SEC = FP(MY.ANGLE.MIN)*60  ! and seconds
80
90 MY.ANGLE$ = STR$(IP(MY.ANGLE))&CHR$(248)&
      STR$(IP(MY.ANGLE.MIN))&"' "&
      STR$(ROUND(MY.ANGLE.SEC,0))&" " "
100
110 PRINT USING "The cotangent 'e is #.#####",MY.ANGLE$,
      COT(MY.ANGLE)
```

Enter angle in degrees: 10

The cotangent of 10°0'0" is 5.67128



---

## COTH Function

COTH returns the *hyperbolic cotangent* of a number.

**COTH**( *numeric-expression* )

**Operation**      The hyperbolic cotangent of *numeric-expression* is computed and returned.

**See also**        Other trigonometric functions

**Examples**

```
10      OPTION PROMPT ""
20
30      INPUT "Enter value: ",VALUE
40      PRINT
50
60      PRINT "The hyperbolic cotangent of";VALUE;"is";COTH(VALUE)
```

**Enter value: .5**

**The hyperbolic cotangent of 0.5 is 2.163953413738**

Statements

## CRT\$ Function

CRT\$ returns the string of characters that, when output to the console or a printer, perform the various control functions of the console or printer.

**CRT\$( *string-parameter* )**

**Operation**      The *string-parameter* is examined and the corresponding character string is generated that performs the indicated action when output to the screen or printer. The *string-parameter* may be any one of the following:

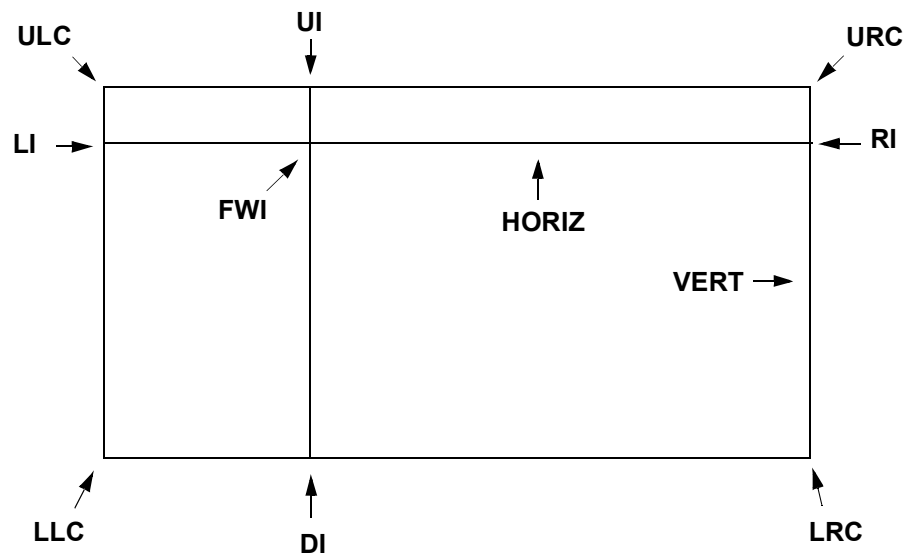
Parameter	Effect when output to	
	Console	Printer
<u>B</u> ELL	Sound the buzzer or bell.	Sound the buzzer or bell.
<u>B</u> OFF	Set blink off.	Set italics off.
<u>B</u> ON	Set blink on.	Set italics on.
<u>C</u> LEAR	Erase the screen or window. Same as the CLS\$ function.	Eject the page.
<u>C</u> R	Perform CR,LF.	Perform CR,LF.
<u>D</u> C	Delete current character. Characters to right of the cursor position are shifted left one character.	
<u>D</u> L	Delete current line. Lines below cursor position are shifted up one line.	
<u>D</u> OWN	Move cursor to next line, same column.	Perform LF.
<u>E</u> OL	Erase characters from current position to the end of the line.	Set double width print on.
<u>E</u> OS	Erase characters from current position to the end of the screen or window.	Set double width print off.
<u>E</u> SC	Escape character (27 <sub>10</sub> , 1B <sub>16</sub> )	Escape character (27 <sub>10</sub> , 1B <sub>16</sub> )
<u>E</u> U	Erase all unprotected characters on screen or window. (Protected characters are shown with reduced intensity.)	

Parameter	Effect when output to	
	Console	Printer
<u>FOFF</u>	Set format mode off.	Set compressed print off.
<u>FON</u>	Set format mode on. Some terminals only protect characters if they have been displayed with PON and when FON is in effect.	Set compress print on.
<u>G OFF</u>	Set graphics mode off.	Set graphics mode off.
<u>G ON</u>	Set graphics mode on. See notes on pages following this table.	Set graphics mode on. See notes.
<u>H OFF</u>	Set half intensity off (same as POFF).	Set secondary color or shading off.
<u>HOME</u>	Position cursor to top left corner of the screen or window.	
<u>HON</u>	Set half intensity on (same as PON).	Set secondary color or shading on.
<u>IC</u>	Insert a blank character. Characters to the right of the cursor location are shifted right one character. The last character of the line is dropped.	
<u>IL</u>	Insert a blank line. Lines below the cursor line are shifted down one line. The last line of the screen is dropped.	
<u>K OFF</u>	Turn the cursor off.	Set double height print on.
<u>K ON</u>	Turn the cursor on.	Set double height print off.
<u>LEFT</u>	Position cursor one character left.	Move the print head one character left.
<u>MOFF</u>	Set monitor mode off.	
<u>MON</u>	Set monitor mode on. (Displays control characters visually, such as LF, CR, etc.)  Caution: some terminals cannot set MOFF via software codes.	
<u>POFF</u>	Set protect mode off for characters that follow.	Set secondary color or shading off.

Parameter	Effect when output to	
	Console	Printer
<u>PON</u>	Set protect mode on. (Protect mode is shown with reduced intensity characters. Protect characters are protected only from the EU parameter.)	Set secondary color or shading on.
<u>RIGHT</u>	Position cursor one character right.	
<u>RVOFF</u>	Set reverse video off.	Set boldface off.
<u>RVON</u>	Set reverse video on.	Set boldface on.
<u>SOFF</u>	End status line display.	
<u>SON</u>	Begin status line display. (Status line is bottom line of screen.)  Caution: not all terminals have status line ability, and of those that do, not all have a full-width status line.	
<u>TAB</u>	Advance cursor to next tab position.	Move print head to next tab position.
<u>UP</u>	Move cursor to previous line, same column.	
<u>UOFF</u>	Set underline off.	Set underline off.
<u>ULON</u>	Set underline on.	Set underline on.
<u>DDI</u>	Double Down Intersect.	Double Down Intersect.
<u>DFWI</u>	Double Four-Way Intersect.	Double Four-Way Intersect.
<u>DHOR</u>	Double Horizontal.	Double Horizontal.
<u>DI</u>	Down Intersect.	Down Intersect.
<u>DLI</u>	Double Left Intersect.	Double Left Intersect.
<u>DLLC</u>	Double Lower Left Corner.	Double Lower Left Corner.
<u>DLRC</u>	Double Lower Right Corner.	Double Lower Right Corner.
<u>DRI</u>	Double Right Intersect.	Double Right Intersect.
<u>DUI</u>	Double Up Intersect.	Double Up Intersect.
<u>DULC</u>	Double Upper Left Corner.	Double Upper Left Corner.
<u>DURC</u>	Double Upper Right Corner.	Double Upper Right Corner.
<u>DVER</u>	Double Vertical.	Double Vertical.
<u>FWI</u>	Four Way Intersect.	Four Way Intersect.
<u>HOR</u>	Horizontal.	Horizontal.
<u>LI</u>	Left Intersect.	Left Intersect.

Parameter	Effect when output to	
	Console	Printer
<u>LLC</u>	Lower Left Corner.	Lower Left Corner.
<u>LRC</u>	Lower Right Corner.	Lower Right Corner.
<u>RI</u>	Right Intersect.	Right Intersect.
<u>RLLC</u>	Rounded Lower Left Corner.	Rounded Lower Left Corner.
<u>RLRC</u>	Rounded Lower Right Corner.	Rounded Lower Right Corner.
<u>RULC</u>	Rounded Upper Left Corner.	Rounded Upper Left Corner.
<u>RURC</u>	Rounded Upper Right Corner.	Rounded Upper Right Corner.
<u>UI</u>	Up Intersect.	Up Intersect.
<u>ULC</u>	Upper Left Corner.	Upper Left Corner.
<u>URC</u>	Upper Right Corner.	Upper Right Corner.
<u>VERT</u>	Vertical.	Vertical.

The figure below shows the relationship of the line drawing names to the components of a line drawn box.



Statements

## Notes

The parameters may be specified with upper, lower or mixed case characters.

Some terminals require that video attributes take a display position. If your programs run on such a terminal you must take that into account when positioning the cursor and setting video attributes.

Similar to the [AT\\$](#) and [CLSS](#) functions, the [CRT\\$](#) function does not perform the function or draw the characters on the screen or printer unless it is output to the screen or printer with a [PRINT](#) or [PUT](#) statement.

The actual performance of the video attribute, video function, or line drawing character is performed by the console or printer device driver via the class code attached to it. If there are problems with the operation of these functions, examine the class code file attached to the device and refer to the operator's manual for the terminal.

**GON & GOFF** Many terminals that can display line graphic characters use an alternate character set for graphics. To display a graphics character on these terminal the computer must transmit a lead-in sequence of characters that indicate the characters following are graphics characters. After transmitting the graphics characters, THEOS instructs the terminal to switch back to its normal character set.

For example, a terminal might use the sequence of characters ESC, H, STX to switch to the graphics character set and the sequence ESC, H, ETX to switch back to the normal character set. On that terminal, the computer transmits seven characters to display one graphics character: three characters to enable graphics, the graphics character, three characters to disable graphics.

The normal mechanism for displaying a single graphics character can degrade display speed excessively when displaying many graphics characters at one time. Use the GON and GOFF codes when displaying several consecutive graphics characters. They instruct THEOS to try to optimize the output to the terminal.

The GON code instructs THEOS to examine the class code file. If THEOS determines that all the graphics characters use the same beginning and ending sequence of characters, it optimizes the output by transmitting the lead-in sequence only. The GOFF code instructs THEOS to transmit the terminating sequence.

Using the previous example terminal, the number of characters transmitted to display ten graphics characters decreases from 70 characters to 16.

When using the GON and GOFF codes on a terminal that does not use a lead-in and terminating sequence for graphics, THEOS performs no optimization and the graphics characters display normally.

Refer to the following example for a typical usage of the [CRT\\$](#) function and the GON and GOFF codes.

**See also** [AT\\$](#), [CHR\\$](#), [CLSS](#), [COLOR](#)

## Examples

This subroutine displays a print status box on the screen. Because there are so many consecutive graphics characters being displayed, the GON and GOFF codes are used to optimize the character transmissions to the terminal display.

The top line of the box is drawn, followed by the two sides of the box, and then the bottom line of the box is drawn.

PRINT.STATUS.BOX:

```
PRINT CRT$("KOFF");

PRINT AT$(5,5);CRT$("HON");CRT$("GON");
PRINT CRT$("ULC");RPT$(25,CRT$("HOR"));CRT$("URC");
  CRT$("GOFF");

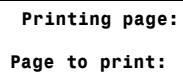
FOR L% = 6 TO 10
  PRINT AT$(5,L%);CRT$("VERT");AT$(31,L%);CRT$("VERT");
NEXT

PRINT AT$(5,11);CRT$("GON");
PRINT CRT$("LLC");RPT$(25,CRT$("HOR"));CRT$("LRC");
  CRT$("GOFF");

PRINT CRT$("HOFF");

PRINT AT$(7,7);" Printing page:";
PRINT AT$(7,9);"Pages to print:";

RETURN
```



```
Printing page:
Page to print:
```

Statements

---

## CSC Function

CSC returns the *cosecant* of an angle.

CSC( *numeric-expression* )

**Operation**      The *numeric-expression* is interpreted as an angle expressed in degrees or radians, depending upon the status of the current [OPTION DEGREE](#) or [OPTION Radian](#). The cosecant of that angle is computed and returned.

**Notes**            Cosecants have values that range from  $-\infty$  to  $-1$  and  $+1$  to  $+\infty$ .  
  
Refer to the [ACOS](#) function for a description of trigonometric relationships.

**Restrictions**    The value of *numeric-expression* must not be either 0 or a multiple of  $\pi$  radians (multiple of  $180^\circ$ ) as these values are outside this function's domain and cause CSC to return meaningless values.

**See also**        Other trigonometric functions, [OPTION](#) statement

**Examples**

```
10      OPTION PROMPT " "
20
30      INPUT "Enter angle in degrees: ",MY.ANGLE
40      PRINT
50
60      MY.ANGLE.MIN = FP(MY.ANGLE)*60      ! Determine minutes
70      MY.ANGLE.SEC = FP(MY.ANGLE.MIN)*60  ! and seconds
80
90      MY.ANGLE$ = STR$(IP(MY.ANGLE))&CHR$(248)&
              STR$(IP(MY.ANGLE.MIN))&"' "&
              STR$(ROUND(MY.ANGLE.SEC,0))&"'' "
100
110     PRINT USING "The cosecant of 'e is #.#####",MY.ANGLE$,
        CSC(MY.ANGLE)
```

Enter angle in degrees: 12.4567

The cosecant of 12°27'24" is 4.63603



---

## CSCH Function

CSCH returns the *hyperbolic cosecant* of a number.

**CSCH**( *numeric-expression* )

**Operation**      The hyperbolic cosecant of *numeric-expression* is computed and returned.

**See also**      Other trigonometric functions

**Example**

```
10      OPTION PROMPT " "  
20  
30      INPUT "Enter area: ",AREA  
40      PRINT  
50  
60      PRINT "The hyperbolic cosecant of";AREA;"is";  
          CSCH(AREA)
```

**Enter area: 23**

**The hyperbolic cosecant of 23 is 2.05237592634E-10**

Statements

---

## CSI Statement

CSI executes a compiled program or EXEC and returns to the calling program.

CSI *command-line*

**Operation** All files and I/O channels are closed; the THEOS Command String Interpreter (CSI) is invoked with the *command-line*. After the command has finished execution, control returns to this calling program with the [CSI.RETURN.CODE](#) function set to the return code value of the *command-line*.

**Notes** This statement is identical to the [SYSTEM](#) statement except that the CSI statement closes all files and [SYSTEM](#) does not.

The *command-line* may specify any executable command (compiled C language, BASIC language, assembly language) or an EXEC program.

The first character of *command-line* may be the angle bracket ( > ) which causes the command line to display on the console. Without this angle bracket the command is invoked “silently.”

Programs and EXEC procedures called by the CSI statement may change directories or accounts. However, the current directory and account are not maintained nor restored when the CSI statement is executed. When accounts or directories are changed they should be changed back prior to returning to this calling program.

Actions	Statements						
	CHAIN	LINK	RUN	CSI	SYSTEM	QUIT	END
Close all files	✓		✓	✓		✓	✓
Clear all non-COMMON data	✓	✓	✓			✓	✓
Clear all COMMON data			✓			✓	✓
Terminate open program structures	✓	✓	✓			✓	✓
Clear the current ON ERROR trap	✓	✓	✓			✓	✓
Clear ON KEY, ON EVENT and ON MOUSE	✓	✓	✓			✓	✓
Begin execution of another program	✓	✓	✓	✓	✓	✓	
Return to calling program				✓	✓		
Reset all OPTIONS						✓	
Close windows							✓
Deactivate subordinate subtasks				✓	✓	✓	✓

**Restrictions**      The CSI statement cannot be used in a subtask program. Attempting to do so causes all subordinate subtasks to terminate, including the current task.

**See also**            [CLOSE](#), [CSI.RETURN.CODE](#), [SYSTEM](#)

### Example

*The SPOOLER list output is redirected to a disk file so that this program can use the information in the list.*

```
5820      CSI "SPOOLER LIST > "&WORKFILES$
5830      FILES.CLOSED% = TRUE%
~
4320      PRINT CLS$;AT$(1,6);
```

*Invoke the SPOOLER command to perform a print alignment pattern for a selected report.*

```
4330      CSI "SPOOLER 1 ALIGN "&STR$(REPORT.NBR%(SELECT.IDX%))
4340
4350      PRINT CLS$;
```

---

## CSI.RETURN.CODE Function

The CSI.RETURN.CODE returns the return code set by the last executed CSI or SYSTEM statement command.

### CSI.RETURN.CODE

**Operation** The return code set by the last CSI or SYSTEM statement executed is returned.

**Notes** The return code is set by all THEOS utility commands and can be set by any program. By convention, a return code of zero indicates program success; non-zero return codes indicate program failure and sometimes the type of failure.

The primary codes that are used by THEOS utilities, and that may be used by user-written programs, are:

Code	Meaning / Message
2	Command not found.
5	Insufficient privilege.
6	Incorrect serial number.
7	Invalid command syntax.
12	Drive code missing or invalid.
13	Disk not attached.
14	Invalid mode opening "fn.ft:fd".
15	Too many files open.
16	Syntax error.
18	File "fn.ft:fd" is protected.
19	File "fn.ft:fd" not found.
20	Device not attached.
21	Disk not mounted.
22	Member "fn.ft.mn:fd" not found.

There are many more "standard" codes and messages. For a complete listing refer to the SYSTEM.TEOS`nnn`.MESSAGE file and the MESSAGE utility.

**See also** CSI, SYSTEM

**Example**

*Execute a command  
and then test the  
possible error  
codes and condi-  
tions.*

```
1000  SYSTEM "CREATE "&NEWFILE$&" (DIR RECL 24, FILE 2000"  
1010  
1020  SELECT CSI.RETURN.CODE  
1030      CASE 42          ! Disk full?  
~  
1100      CASE 43          ! Directory full?  
~  
1200      CASE 44          ! Already exists?  
~  
1300      CEND
```

# DATA Statement

DATA defines string and numeric constants that can be READ into variables by the program.

**DATA** *data-list*

*data-list* » *data*[, *data-list*]  
*data* » *numeric-constant*  
*string-constant*

**Operation** Defines the numeric and string constants in the *data-list*.

**Notes** The sequence of the items in the *data-list* and the sequence of the DATA statements in the program is very important. Refer to the READ statement for a description of how the *data-list* is used.

As a convenience, during program entry and modifications, the MultiUser BASIC editor allows string constants to be entered without the surrounding quote marks. The quotes are added automatically when the statement is accepted. However, you must specify the quotes if the string must contain leading or trailing spaces, embedded comma characters or lowercase characters.

**Restrictions** The DATA statement must be the only statement on a line. It cannot be followed by a [REM](#) statement or comment

**See also** [MAT READ](#), [READ](#), [REM](#), [RESTORE](#)

## Example

An array is dimensioned to hold 100 city names.

```
900510      DIM CITIES$(100)
~
900690 MAT READ CITIES$
~
```

The MAT READ initializes the array by reading the next 100 DATA items.

```
980230 REM Bay area cities
980240
980250 DATA "Alameda"
980260 DATA "Alamo"
980270 DATA "Albany"
980280 DATA "Antioch"
980290 DATA "Aptos"
980300 DATA "Atherton"
```

---

## DATE\$ Function

DATE\$ returns a date-string for today's date or a day value.

DATE\$( *day-number* )

**Operation**      The *day-number* expression is evaluated and interpreted as the number of days since January 1, 1900. A date-string is then generated matching that day number.

A *day-number* of 0 is a special code meaning today's date.

**Notes**            The date-string generated is formatted according to the current DATE-FORM and DATEOUT system environment variables (see the SET command in the *THEOS System Reference Manual*).

**DATEFORM**      The format of the date-string returned by this function depends upon the current setting of the DATEFORM system environment variable. The DATE-FORM specifies the sequence of the components (year, month and day) and the delimiters between the components.

DATEFORM			
<i>day-number</i>	1	2	3
0	10/15/1999	15-10-1999	1999.10.15
1	01/01/1900	01-01-1900	1900.01.01
3,333	02/15/1909	15-02-1909	1909.02.15
31,362	11/12/1985	12-11-1985	1985.11.12
38,456	04/15/2005	15-04-2005	2005.04.15
73,108	02/28/2100	28-02-2100	2100.02.28

## DATEOUT

The current setting of DATEOUT determines how DATE\$ specifies the year in the resulting string returned by the function.

**DATEOUT 0** Years are specified with two-digits if the date is in the twentieth century and four digits when the date is in the twenty-first century.

**DATEOUT 1** The year is always specified with a four-digit year number. This setting should be used to be Y2K compliant.

**DATEOUT 2** The year is always specified with a two-digit year number.

	DATEOUT		
date	0	1	2
"07/04/1900"	"07/04/00"	"07/04/1900"	"07/04/00"
"07/04/1950"	"07/04/50"	"07/04/1950"	"07/04/50"
"07/04/1999"	"07/04/99"	"07/04/1999"	"07/04/99"
"07/04/2030"	"07/04/2030"	"07/04/2030"	"07/04/30"
"07/04/2001"	"07/04/2001"	"07/04/2001"	"07/04/01"

In the above table, DATEFORM 1 is used.

### Restrictions

*day-number* is interpreted as an unsigned value. The *day-number* for dates prior to January 1, 1900 is not allowed. Negative values are interpreted as large positive numbers. For example, DATE\$( - 1 ) generates the date string for June 5, 2079. The *day-number* must be in the range -65536 through +73108. Values outside this range return a null string.

### See also

[DAY](#), [DTE\\$](#), [STRTIME\\$](#)

### Example

*This code computes a month start date and the date of the following month start.*

```
9410 START.DATE = DAY(POST.DATE$[1:2]&"/01/"&POST.DATE$[6:7])
9420 END.DATE$ = DATE$(START.DATE+32)
9430 END.DATE$[4:5] = "01"
9440 END.DATE = DAY(END.DATE$)
```



---

# DAY Function

DAY returns the day number, since January 1, 1900, of a date string.

DAY( *date-string* )

**Operation**      The *date-string* is evaluated as a date representation, using the currently set DATEFORM and DATEIN system environment variables (see the SET command in the *THEOS System Reference Manual*). The number of days between January 1, 1900 and *date-string* is computed and returned.

When *date-string* is invalid for the current DATEFORM, a -1 is returned indicating the error. For example, using DATEFORM 3, a date of “1992/02/30” is invalid (February 30, 1992).

**Notes**            The *date-string* may use any non-numeric character for separators between the day, month, and year. The separators may be different between the three components. For example, “01.02/03” “1,2,3” “1-2.3”. A separator between the components is not necessary, but if there is one separator, then there must be two. For example, “010203” is valid, “01/0203” is not. When there are no separators used, the century is assumed to be 19 unless you specify the date with a four-digit number.

The operation of the DTE\$ function is used to validate the *date-string*.

**DATEIN**            The current setting of DATEIN determines how DAY interprets a date when a two-digit year is specified in *date-string*.

- DATEIN 0

Two-digit year dates are interpreted as a date in the twentieth century (1900's).
- DATEIN 1

Two-digit year dates are interpreted as a date in the same century as the current system date.
- DATEIN 2

Two-digit year dates are interpreted as a date in the nearest century to the current system date. For instance, if the current year is 1999 and a date is specified with a year of '00' then that date is interpreted as a date in the year 2000.

Statements

A date specified with a year of '40' is a date in the year 2040 because 2040 is closer to 1999 then 1940 would be.

		DATEIN		
current date	date-string	0	1	2
03/05/1999	"07/04/00"	"07/04/1900"	"07/04/1900"	"07/04/1900"
	"07/04/30"	"07/04/1930"	"07/04/1930"	"07/04/1930"
	"07/04/50"	"07/04/1950"	"07/04/1950"	"07/04/1950"
	"07/04/70"	"07/04/1970"	"07/04/1970"	"07/04/1970"
	"07/04/99"	"07/04/1999"	"07/04/1999"	"07/04/1999"
03/05/2005	"07/04/00"	"07/04/1900"	"07/04/2000"	"07/04/2000"
	"07/04/30"	"07/04/1930"	"07/04/2030"	"07/04/2030"
	"07/04/50"	"07/04/1950"	"07/04/2050"	"07/04/2050"
	"07/04/70"	"07/04/1970"	"07/04/2070"	"07/04/1970"
	"07/04/99"	"07/04/1999"	"07/04/2099"	"07/04/1999"

**Restrictions**      The smallest day value is 1, corresponding to January 1, 1900. A *date-string* for dates prior to January 1, 1900 are invalid.

Although this function returns whole numbers, it returns them as numeric values. Do not save the returned value in an integer variable if the date might be greater than September 17, 1989 (the largest date that can be represented by an integer).

**See also**      [DATE\\$](#), [DTE\\$](#), [STRTIME\\$](#)

**Example**

Compare a transaction date field (D\$) with the start and ending dates for the report.

```
3350 IF DAY(D$)>=START.DATE AND DAY(D$)<END.DATE
3360 REM Transaction date is within report date range
```

# DECLARE CALL Statement

The DECLARE CALL statement defines the formal arguments expected by a C language routine called with the [CALL](#) statement.

1

**DECLARE CALL** *name*

2

**DECLARE CALL** *name*( *prototype* )

*prototype*

» *variable-name*[, *prototype*]  
**DIM** *array-name*( *integer-var1*[, *integer-var2*] ), *prototype*  
**ADDROF**( *variable-name* ), *prototype*  
**ADDROF**( *#integer-var* ), *prototype*  
**ADDROF**( **SUB** *subprogram* ), *prototype*

Operation	The DECLARE CALL statement is not an executable statement. It is a directive to the interpreter or compiler stating the name of a C language routine accessed by the program and defining the argument types expected by that routine.
Notes	<p>Although MultiUser BASIC allows C language functions to be called without using the DECLARE CALL statement to define the arguments for the C function, doing so causes BASIC to assume that the C function uses the prior version 1.0 style interface and imposes restrictions on what can be passed to that function.</p> <p>A C function that does not use the DECLARE CALL statement can only be passed short strings (string length less than 256 characters) and may only be passed arrays with the <a href="#">ADDROF</a> function, not the DIM capability. This severely restricts the capabilities of the called function.</p> <p>Refer to the <i>Programmer's Guide</i> for additional information.</p>
Restrictions	The DECLARE CALL statement is used only to describe the name and arguments of a new-style C language routine interface.
See also	<a href="#">CALL</a>
Example	Refer to the <i>MultiUser BASIC Programmer's Guide</i> chapter on "Using C Language Routines" for example usage.

---

## DEF FN Statement

DEF FN declares a user-defined function, with or without formal arguments.

```
1  DEF FN $name$  =  $expression$ 
2  DEF FN $name$ (  $argument-list$  ) =  $expression$ 
3  DEF FN $name$ 
4  DEF FN $name$ (  $argument-list$  )
```

---

$argument-list$       »     $variable-name$ [,  $argument-list$ ]

### Operation

[Mode 1](#) and [Mode 2](#) of the DEF FN statement define single-line functions; modes 3 and 4 define multiple-line functions and are used in conjunction with the [FNEND](#) statement to form a DEF FN-FNEND program structure. The series of statements between a [Mode 3](#) or [Mode 4](#) DEF FN statement and the [FNEND](#) statement are referred to as the function definition body.

**Mode 1**—Declares a single-line, user-defined function without arguments. The *expression* is evaluated and returned as the value of the user-defined function. Any variables referenced in *expression* have values defined by the main program code.

**Mode 2**—Declares a single-line, user-defined function with arguments. The *expression* is evaluated and returned as the value of the user-defined function. Variable names referenced in *expression* may refer to variables in the main program code or variable names in *argument-list* (see [Notes](#)).

**Mode 3**—Declares the start of a multiple-line, user-defined function without arguments. The value of the function is determined later by an assignment statement in the function definition body. Any variables referenced in the function definition body have values defined by the main program code.

**Mode 4**—Declares the start of a multiple-line, user-defined function with arguments. The value of the function is determined later by an assignment statement in the function definition body. Variable names referenced in the function definition body may refer to variables in the main program code or variable names in *argument-list* (see [Notes](#)).

All user-defined functions have names that start with FN. Following the FN you must specify one or more characters (letters, digits, or periods) and an optional type specifier (% or \$). For example:

```

FNCUBE
FN.INPUT$
FN.SELECT.FROM.LIST.BOX$
FN.HIGHLIGHT$
FN.VALID%

```

The characters following the FN may form a name that is the same as a variable name used elsewhere in the program. Because of the leading FN, the two names refer to different objects.

In argumentless function definitions ([Mode 1](#) and [Mode 3](#)), any variables referenced in the *expression* or in the function definition body refer to variables used in the main program code. For example:

```

A$ = "MAIN PROGRAM"
PRINT A$
PRINT FN.DEMO$
...
DEF FN.DEMO$
A$ = "FUNCTION DEFINITION"
FN.DEMO$ = A$
FNEND

```

When the above code is executed, the display will be:

```

MAIN PROGRAM
FUNCTION DEFINITION

```

and the value of A\$ will be "FUNCTION DEFINITION" because it was changed in the function definition body.

The arguments in function definitions with arguments ([Mode 2](#) and [Mode 4](#)) are *local variables*. That is, the variable-names used in the *argument-list* are local to the function definition. They exist only during execution of the function and are separate from any variables of the same name used in the main program code or in other function definitions. For example:

```

A$ = "MAIN PROGRAM"
PRINT A$
PRINT FN.DEMO$( "TEST" )
...
DEF FN.DEMO$(A$)
A$ = A$&" of FUNCTION DEFINITION"
FN.DEMO$ = A$
FNEND

```

When the previous code is executed the display will be:

The value of A\$ will still be "MAIN PROGRAM" even though the function definition body has a statement changing the value of A\$. In this example, the A\$ is a formal argument to the function definition. All references to A\$ in the function body refer to the local argument. Local arguments are initialized to the values passed by the statement calling the function (in this case "TEST").

As indicated in the example above, the values of formal arguments are set by the statement using the function (in the example, the PRINT statement). When the *argument-list* contains more than one argument, the values are assigned in a one-to-one correspondence to the list of values used in the statement using the function. For example:

```
POST.DATE$ = FN.GET.DATE$(24,2,"MY",POST.DATE$,"POSTMNTH")
...
DEF FN.POST.DATE$(X%,Y%,STYLE$,DATE.FLD$,HELP.FILE$)
...
FNEND
```

At the time the function is called, the values of the formal arguments are assigned:

```
X% = 24
Y% = 2
STYLE$ = "MY"
DATE.FLD$ = POST.DATE$
HELP.FILE$ = "POSTMNTH"
```

Multiple-line functions do not have to have a statement assigning a value to the function or there may be more than one statement assigning the value. When no assignment statement exists, or one exists but is not executed, the value of the function is 0 or a null string, as appropriate.

The **BREAK** statement may be used in a multiple-line function definition. It will cause control to be transferred to the function definition's **FNEND** statement.

A function definition may be anywhere in the program.

The **TRACE** and TRACE VARS commands operate during user-defined function execution.

It is also possible to perform a **BREAK** on a line or a variable in a user-defined function or a **Break**, **C** during the execution of a multiple-line user-defined function. When the **BREAK** occurs the proper line number is displayed and you may **CONTINUE**, **STEP** or **SSTEP** to the next statement in the function.

Any local variables (formal arguments) used in a function definition are not normally available to the [VARS](#) command and display as if the variable were undefined. Use the [VIEW CURRENT](#) command setting to enable VARS display of currently defined variables, including formal arguments to the currently executing function.

<b>Restrictions</b>	<p>The DEF FN statement may not be specified on a multiple-statement line.</p> <p>Single-line functions must not be <i>recursive</i>. A program stack overflow will result. (It is okay for multiple-line functions to be recursive, but only if programmed properly.) An example of a recursive single-line function is:</p> <pre>DEF FN.TEST(A%) = A%+3*FN.TEST(A%)</pre> <p>The type (string or numeric) of <i>expression</i> must agree with the type of <i>FNname</i>.</p> <p>There must be one and only one <a href="#">FNEND</a> statement for every multiple-line DEF FN statement.</p> <p>Function definitions may not be nested. However, function calls may be nested.</p> <p>User-defined functions may not be redefined in the same program. That is, only one definition for a specific <i>FNname</i> is used in a program. The interpreter and compiler will both detect this error with the message “Duplicate user-defined function “FNaaaa” definition at line “nnnn”.”.</p> <p>References to user-defined functions must provide the same number and type of parameters as specified in the function definition or the error “DEF does not agree with FN call at line ...”.</p>
<b>Special note:</b>	<p>A multiple-line function definition is a program structure that must only be entered at the top (DEF FN) and exited at the bottom (<a href="#">FNEND</a>). All line numbers and line labels defined between a DEF FN statement and its <a href="#">FNEND</a> statement can only be referenced within that function definition.</p>
<b>See also</b>	<a href="#">FNEND</a>

## Example

*FN.HIGHLIGHT\$* 2020 PRINT FN.HIGHLIGHT\$(IDX%,DSP.IDX%,WIDTH%,1);  
is called in the ~  
PRINT statement. 20210 FIELD\$ = FN.DIGIT.STRIPS(FIELD\$)  
Since the function ~  
definition does not  
assign a value to  
the function, the  
PRINT statement at  
line 2020 prints  
nothing.

However, the func- 703470 DEF FN.HIGHLIGHT\$(I%,D%,W%,H%)  
tion definition 703480  
prints text on the 703490 PRINT AT\$(1,D%);  
screen, either in 703500 IF H% THEN PRINT CRT\$("RVON");  
reverse video, or 703510 PRINT " ";RPAD\$(ITEMS\$(I%),W%+1);  
normal video, 703520 IF H% THEN PRINT CRT\$("RVOFF");  
depending upon the 703530  
value of the fourth 703540 FNEND  
argument (H%). ~

This second func- 704120 DEF FN.DIGIT.STRIPS(F\$)  
tion is given a 704130  
string and it returns 704140 X\$ = ""  
a string with all 704150  
non-digit charac- 704160 FOR DI% = 1 TO LEN(F\$)  
ters removed. 704170 DI\$ = F\$[DI%:DI%]  
704180 IF DI\$>="0" AND DI\$<="9" THEN X\$ = X\$&DI\$  
704190 NEXT  
704200  
Note: because X\$, 704210 FN.DIGIT.STRIP\$ = X\$  
DI%, and DI\$ are 704220  
not formal argu- 704230 FNEND  
ments, this function  
might be modifying  
variables used by  
the program.



# DEG Function

DEG returns the number of *degrees* represented by a value in *radians*.

DEG( *radian-value* )

**Operation**      *radian-value* is interpreted as a number of radians. The degree equivalent of those radians is returned.

**Notes**            There are 360 degrees or  $2\pi$  radians in a circle.

                      The sign of *radian-value* is retained as the sign of the function.

**See also**        [OPTION](#), [RAD](#)

## Example

*This simple program computes the sine of an angle specified in radians.*

*Because the current option is DEGREEs, the angle must be converted to degrees before being given to the SIN function.*

```
10      OPTION DEGREE, PROMPT " "
20
30      INPUT "Enter angle in radians: ",MY.ANGLE
40      PRINT
50
60      MY.ANGLE.DEG = DEG(MY.ANGLE) ! Convert to degrees
70      MY.ANGLE.MIN = FP(MY.ANGLE.DEG)*60 ! Find minutes
80      MY.ANGLE.SEC = FP(MY.ANGLE.MIN)*60 ! and seconds
90
100     MY.ANGLE$ = STR$(IP(MY.ANGLE.DEG))&CHR$(248)&
          STR$(IP(MY.ANGLE.MIN))&"' "&
          STR$(ROUND(MY.ANGLE.SEC,0))&"'"
110
120     PRINT USING "The sine of 'e (##.#### rad) is #.####",
          MY.ANGLE$,MY.ANGLE,SIN(MY.ANGLE.DEG)
```

Enter angle in radians: 2.5

The sine of 143°14'22" ( 2.5000rad) is 0.59847

Statements

# DEL\$ Function

DEL\$ returns a string with a field or subfield deleted from it.

DEL\$( *string-expression*, *field-number*, *subfield-number*)

**Operation**      The subfield of *string-expression* is removed from *string-expression*, along with its delimiter. The resulting string is returned.

A *subfield-number* of 0 causes *field-number* substring to be deleted.

**Notes**            String fields can be formatted with fields and subfields with the [INS\\$](#) function. That function builds strings such that the string contains two-level subfields delimited by special characters. The [EXT\\$](#) function extracts these subfields for usage as individual strings. The [REP\\$](#) and DEL\$ functions modify the subfields.

A string with subfields is useful when a related group of items are manipulated as a group and rarely as individual components. For example, an employee name and address record might have prior-year earnings fields for total earned, vacation pay, sick pay, federal withholding, state withholding, *etc.* These fields might only be reported in a special report. If the fields are grouped into one string using subfields the normal maintenance and reporting programs would read and write the record using only one variable name for the entire set of fields.

The delimiters used are special characters with value 221 and 222.

Deleting a subfield with this function changes the relative numbers of all subfields following the deleted subfield. To remove the contents of a subfield without removing its delimiter, use the [REP\\$](#) function with a null string.

Deleting a subfield with only empty subfields following it causes all of the trailing delimiters to be removed.

**See also**          [EXT\\$](#), [INS\\$](#), [REP\\$](#)

<b>Example</b>	1010	FLD\$ = DEL\$(FLD\$,3,2)	! Remove a subfield
	2000	FLD\$ = DEL\$(FLD\$,2,0)	! Remove an entire field

# DELETE Statement

DELETE removes a record from a data file.

DELETE #channel, key

Operation	Deletes the record, indicated by <i>key</i> , from the direct, keyed, or indexed access disk file.
Notes	<p>Deleting a record always sets the end-of-file indicator (<a href="#">EOF</a> function) off and releases all record locks on the file channel.</p> <p>No error is detected when the <i>key</i> does not exist in the file.</p>
Restrictions	<p>The channel must refer to a file opened with OUTPUT or UPDATE. Refer to the <a href="#">OPEN</a> statement for details.</p> <p>When writing to a direct access file, the <i>key</i> must be a numeric expression. Writing to a keyed or indexed access file requires that the <i>key</i> be a string expression.</p>

## Example

<i>The program is about to write a record whose key has been changed. Since writing the new record does not replace the old record because of the key change, it must be deleted.</i>	201000	MOD.FILE:
	201010	
	201020	EVENT.FLAG% = TRUE%
	201030	
	201040	IF NOT NEW.ITEM% AND KEY\$<>REC\$(1) ! Key changed?
	201050	DELETE #1,KEY\$ ! Remove old record
	201060	IFEND
	201070	
	201080	WRITE #1,REC\$(1): REC\$(2),REC\$(2),REC\$(4),REC(5)
	201090	

# DIM Statement

The DIM (dimension) statement defines array names and sizes, and allocates memory for the array contents.

**DIM** *array-list*

*array-list* » *array*[, *array-list*]  
*array* » *array-name*( *subscript* )  
*array* » *array-name*( *subscript1*, *subscript2* )

**Operation** The *array-name* is declared to be a dimensioned array and memory is allocated for each of the elements of the array.

**Notes** Arrays dimensioned with the DIM statement may be shared access arrays or local access arrays, depending upon the position of the DIM statement in the program. DIM statements positioned within a subprogram (between a SUB statement and its END SUB statement) operate as if it were a LOCAL statement and allocates local access arrays; DIM statements appearing anywhere else in a program define shared access arrays.

The position of a DIM statement outside of a subprogram does not matter as all of these DIM statements and COMMON statements are compiled as if they were grouped together at the end of the program. The interpreter simulates this by scanning the entire program for DIM and COMMON statements prior to executing the first statement in the program.

An array with subscripts less than 10 does not have to be dimensioned. Larger arrays must be dimensioned with the DIM, COMMON, LOCAL, SHARED or STATIC statements.

**Dimensioning** Arrays may be one-dimensional or two-dimensional.

One-dimensional array: 

0	1	2	3	4
---	---	---	---	---

Two-dimensional array: 

0,0	0,1	0,2	0,3	0,4
1,0	1,1	1,2	1,3	1,4
2,0	2,1	2,2	2,3	2,4

The number of elements allocated for an array depends upon the [OPTION BASE](#) statement. When [OPTION BASE 0](#) is in effect, one is added to each of the dimensions specified to account for the zero indices.

For example:

```
OPTION BASE 0
DIM A(5)           ! 6 elements allocated
DIM B(12)          ! 13 elements allocated
DIM C(2,4)         ! 15 elements allocated

OPTION BASE 1
DIM A(5)           ! 5 elements allocated
DIM B(12)          ! 12 elements allocated
DIM C(2,4)         ! 8 elements allocated
```

**Restrictions**      The subscript must be positive integer constants in the range of 1–32767.

The *array-name* may only be used as an array in the program and not as a simple variable. Attempts to do so result in the error “Scalar variable used as an array at line ...” or “Array variable used as a scalar at line ....”

Arrays may not be redimensioned with this declaration. Attempts to do so result in an error message. Use the [REDIM](#) statement to redimension an array.

**See also**            [COMMON](#), [LOCAL](#), [MAT](#), [OPTION BASE](#), [PUBLIC](#), [REDIM](#), [SHARED](#), [STATIC](#)

**Example**

```
900500      DIM MAIN$(10,6)
900510      DIM CITIES$(100)
900520      DIM STATE.CODES$(51),STATES$(51)
900530      DIM ITEMS$(200)
900540
~
900690      MAT READ CITIES$
900700
900710      FOR I% = 1 TO 51
900720          READ STATE.CODES$(I%),STATES$(I%)
900730      NEXT
```

---

## DTE\$ Function

DTE\$ returns the standardized format for a date string.

DTE\$( *date-string* )

**Operation**      The *date-string* value is analyzed to see if it is a representation of a valid date. If it is, it is reformatted according to the current DATEFORM (see SET command in *THEOS System Reference Manual*). When the date is invalid (e.g. "02/30/87") a null string is returned.

**Notes**            The *date-string* may use any non-numeric character for separators between the day, month and year. The separators may be different between the month and day and between the day and year. For example, "01.02/03" "1,2,3" "1-2.3". A separator between the components is not necessary, but if there is one separator then there must be two. For example, "010203" is valid, "01/0203" is not.

**DATEFORM**        The current setting of the system environment variable DATEFORM is used by DTE\$ to control how it interpretes the components of *date-string* and how it formats the resulting string returned by the function.

Statements

DATEFORM			
<i>date-string</i>	1 = mm/dd/yy	2 = dd-mm-yy	3 = yy.mm.dd
"8.15.90"	"08/15/90"		
"150209"		"15-02-09"	"15.02.09"
"85 11 12"			"85.11.12"
"04,12,2005"	"04/12/2005"	"04-12-2005"	
"2070 07 04"			"2070.07.04"

The blank cells in the above table indicate that the DTE\$ function returns a null string because the *date-string* is invalid in that DATEFORM..

Notice that the various dateforms use different characters for delimiters. The difference in these characters can be used to determine the dateform.

This function also uses the settings for the system environment variables DATEIN and DATEOUT.

DATEIN

The current setting of DATEIN determines how DTE\$ interprets a date when a two-digit year is specified in *date-string*. It has no effect on the format of dates output by DTE\$.

- DATEIN 0

Two-digit year dates are interpreted as a date in the twentieth century (1900's).
- DATEIN 1

Two-digit year dates are interpreted as a date in the same century as the current system date.
- DATEIN 2

Two-digit year dates are interpreted as a date in the nearest century to the current system date. For instance, if the current year is 1999 and a date is specified with a year of '00' then that date is interpreted as a date in the year 2000. A date specified with a year of '40' is a date in the year 2040 because 2040 is closer to 1999 than 1940 would be.

		DATEIN		
current date	date-string	0	1	2
03/05/1999	"07/04/00"	"07/04/1900"	"07/04/1900"	"07/04/1900"
	"07/04/30"	"07/04/1930"	"07/04/1930"	"07/04/1930"
	"07/04/50"	"07/04/1950"	"07/04/1950"	"07/04/1950"
	"07/04/70"	"07/04/1970"	"07/04/1970"	"07/04/1970"
	"07/04/99"	"07/04/1999"	"07/04/1999"	"07/04/1999"
03/05/2005	"07/04/00"	"07/04/1900"	"07/04/2000"	"07/04/2000"
	"07/04/30"	"07/04/1930"	"07/04/2030"	"07/04/2030"
	"07/04/50"	"07/04/1950"	"07/04/2050"	"07/04/2050"
	"07/04/70"	"07/04/1970"	"07/04/2070"	"07/04/1970"
	"07/04/99"	"07/04/1999"	"07/04/2099"	"07/04/1999"

Statements

## DATEOUT

The current setting of DATEOUT determines how DTE\$ specifies the year in the resulting string returned by the function.

**DATEOUT 0** Years are specified with two-digits if the date is in the twentieth century and four digits when the date is in the twenty-first century.

**DATEOUT 1** The year is always specified with a four-digit year number. This setting should be used to be Y2K compliant.

**DATEOUT 2** The year is always specified with a two-digit year number.

	DATEOUT		
date	0	1	2
"07/04/1900"	"07/04/00"	"07/04/1900"	"07/04/00"
"07/04/1950"	"07/04/50"	"07/04/1950"	"07/04/50"
"07/04/1999"	"07/04/99"	"07/04/1999"	"07/04/99"
"07/04/2030"	"07/04/2030"	"07/04/2030"	"07/04/30"
"07/04/2001"	"07/04/2001"	"07/04/2001"	"07/04/01"

In the above table, DATEFORM 1 is used.

**Restrictions** The range of dates validated by this function is January 1, 1900 through December 31, 2100.

**See also** [DATE\\$](#), [DAY](#), [STRTIME\\$](#)

### Examples

```
~
2960 IF DTE$(TRAN.DATE$)=" "
2970     VALID% = FALSE%
2980     PRINT CRT$( "BELL" );
2990 ELSE TRAN.DATE$ = DTE$(TRAN.DATE$)
3000 IFEND
```



# ELSE Statement

ELSE defines the start of a series of statements that are executed when a multiple-line IF statement expression is false.

1

ELSE

2

ELSE *statement*

**Operation** The ELSE statement marks the beginning of lines that are conditionally executed as part of a multiple-line IF statement. When the *relational-expression* of the IF statement is true, the ELSE statement marks the beginning of statements that are skipped; when false, the ELSE statement marks the beginning of statements that are executed.

**Mode 1**—Merely marks the beginning of lines that are conditionally executed as part of the multiple-line IF statement’s false condition.

**Mode 2**—Marks the beginning and gives the first statement that is conditionally executed as part of the multiple-line IF statement’s false condition.

**Notes** During program entry and edit, the syntax analyzer allows you to enter statements like:

```
ELSE 1000
```

This will be accepted and maintained as if you had entered:

```
ELSE GOTO 1000
```

**Restrictions** An ELSE statement is only valid as part of an IF-IFEND program structure.

**See also** IF, IFEND, THEN

Examples

3630

X = SECOND(TIME\$(0))

3640

IF X<28800.0 OR (X>=43200.0 AND X<=46800.0)

OR X>61200.0

3650

OFFTIME% = TRUE%

3660

ELSE OFFTIME% = FALSE%

3670

IFEND

~

7140

IF EOF(1)

7150

PROGRAM.MSG\$ = "'MAIN' ENTRY MISSING IN MENU DATA"

7160

LET ERR = 44

7170

ELSE MAIN.SELECT\$ = ""

7180

IFEND

---

## END Statement

END marks the logical end of the program.

END

### Operation

The END statement always terminates program execution. Terminating program execution with this statement closes all open I/O channels, closes windows, cleans the program stack and deactivates all subtask programs of this program.

If the program is executing in its compiled form, the program exits, similar to mode 1 of the [QUIT](#) statement, returning control to the calling environment (see [QUIT](#) statement). When the program is executing in the interpreter, the command mode of the interpreter is invoked.

### Notes

Although a program may have several END statements, it is good programming practice to have only one END statement and to have that one END statement be at the physical end of the program. To terminate program execution prior to the physical end of the program, use the [QUIT](#) statement.

The ANSI BASIC standards require that the END statement be the last statement in a program.

In a multiple-task environment, the END statement only closes windows opened by this task, and, since subtasks of this task are deactivated, the windows opened by those subtasks are closed. Windows opened by tasks not subordinate to this task are not closed.

Actions	Statements						
	CHAIN	LINK	RUN	CSI	SYSTEM	QUIT	END
Close all files	✓		✓	✓		✓	✓
Clear all non-COMMON data	✓	✓	✓			✓	✓
Clear all COMMON data			✓			✓	✓
Terminate open program structures	✓	✓	✓			✓	✓
Clear the current ON ERROR trap	✓	✓	✓			✓	✓
Clear ON KEY, ON EVENT and ON MOUSE	✓	✓	✓			✓	✓
Begin execution of another program	✓	✓	✓	✓	✓	✓	
Return to calling program				✓	✓		
Reset all OPTIONS						✓	
Close windows							✓
Deactivate subordinate subtasks				✓	✓	✓	✓

When a compiled program executes the END statement, the program is exited with a return code of zero.

#### See also

[QUIT, STOP](#)

#### Examples

```

2000 END.RUN:
2010

2020 WINDOW SELECT 0, UPDATE OFF
2030 PRINT CRT$("KON");CLSS;
2040 WINDOW SELECT 0
2050
2060 QUIT "LOGOFF"
~
990100 DATA "Sunday","Monday","Tuesday","Wednesday"
990110 DATA "Thursday","Friday","Saturday"
990120
999999 END

```

*The program is exited with the QUIT statement.*

*Here, the END statement is used only to mark the end of the source program.*

---

# END SUB Statement

The END SUB statement defines the physical end of a callable subprogram started with the [SUB](#) statement. It terminates execution of the subprogram and returns control to the statement following the [CALL](#) to the subprogram.

END SUB

**Operation**      The subprogram environment is exited with control returning to the statement following the [CALL](#) to the subprogram.

**Notes**            All variables declared as [LOCAL](#) to the subprogram are discarded.

**Restrictions**    A subprogram can have only one END SUB statement and it must be the last line of the subprogram.

**See also**         [CALL](#), [SUB](#)

**Examples**

```
100000 SUB GET.DECKS! Subprogram to load multiple playing card decks
100010
100020     SHARED CARDS%(416)           ! Array of unshuffled card
           decks
100030     SHARED DECKS%                ! Number of decks in "shoe"
100040
100050     LOCAL I%                    ! Work index variable
100060     LOCAL J%                    ! Work index
100070     LOCAL JJ%                   ! Work index
100080
100090     FOR I% = 1 TO DECKS%        ! Repeat for each deck in shoe
100100
100110         JJ% = (I%-1)*52+1
100120         RESTORE CARDS           ! Point to fresh deck of cards
100130
100140         FOR J% = JJ% TO JJ%+51  ! For each card in deck
100150             READ CARDS%(J%)      ! Read it
100160             NEXT
100170
100180         NEXT
100190
100200     END SUB                     ! Return to caller
```

# EOF Function

EOF returns the end-of-file status of a I/O channel.

EOF( *channel-number* )

**Operation**            The current end-of-file status for file channel *channel-number* is returned.

**Notes**                The end-of-file status is either true (non-zero) or false (zero).

A file opened for OUTPUT is always at end-of-file, regardless of the file organization.

Files opened for INPUT or UPDATE are at end-of-file when:

**SEQUENTIAL files:** when file pointer is after the last record in file.

**DIRECT, INDEXED, and KEYED files:** when the requested record was not found or, after a READNEXT or READPREV, when an attempt has been made to read beyond the last or first record in the file.

**Restrictions**        The file channel must be open.

**See also**            [INPUT](#), [LINPUT](#), [MAT READ](#), [MAT READNEXT](#), [MAT READPREV](#), [OPEN](#), [READ](#), [READNEXT](#), [READPREV](#)

## Examples

<i>The help text file has already been opened. Read the first record in the file.</i>	800150	LINPUT #38: HELP.TEXT\$
	800160	
	800170	L% = 0 ! Number of lines displayed in window
	800180	
<i>Loop until the end of the file has been read.</i>	800190	WHILE NOT <b>EOF(38)</b> ! Until end of help text
	800200	IF L%>=DEPTH% THEN GOSUB RELEASE.HELP.PAGE
	800210	L% = L%+1 ! Count lines displayed in window
	800220	PRINT AT\$(2,L%);HELP.TEXT\$;
<i>Display the text in the help display window.</i>	800230	LINPUT #38: HELP.TEXT\$ ! Get next help text
	800240	WEND
<i>Read another text record and repeat if not at end-of-file.</i>	800250	
	800260	GOSUB RELEASE.HELP.PAGE ! Wait for operator
	~	

Statements

---

## EPS Function

EPS returns the smallest, positive fraction that the system can maintain (epsilon).

**EPS**

**Operation**      The constant for the smallest positive value maintained by the system is returned.

**Notes**            The actual value returned depends upon the current status of the [OPTION IEEE](#) or [OPTION BCD](#) statement:

OPTION	
BCD	IEEE
$1^{-126}$	$2.225073858507^{-308}$

This is a very small value. When this value is added to another value that is larger by more than 12 orders of magnitude, no change is made to the value because there are only 13 significant digits.

EPS is the smallest positive value maintained by the system. The value zero and negative values will compare less than EPS.

**See also**        [INF](#), [OPTION](#)

**Examples**        1500      PRINT "The smallest logarithm is ";LOG(**EPS**)

The smallest logarithm is -292.4283068102

Statements

---

## ERL Function

ERL returns the *error line number* of the program line causing the last error.

### ERL

**Operation** When an execution-time error occurs, the line number or location of the statement causing the error is saved. The ERL function returns this saved line number.

**Notes** When running a program with the interpreter the actual line number is returned by this function. Normally, the ERL function is compared to a line number or line label in an IF statement. The [RENUMBER](#) command renumbers these line references. The ERL function used in a [CASE](#) statement is also [RENUMBER](#)ed correctly.

When running a compiled program the value returned is the memory address of the line containing the statement that caused the error. The compiler converts the line number or line label reference when it is compared to the value of ERL.

For example:

```
IF ERL = 1010 THEN ...
```

will work properly in both the interpreted program and the compiled program. A problem could occur if you save the value of ERL in another numeric variable and then compare that variable's value to a line number.

The ERL function may be compared to a specific line number or you may use the relational operators `>`, `<`, `=>`, `<=`, and `<>` for line number range comparisons. This range comparison would be necessary when multiple statement lines are used.

Label references may also be used when comparing to the ERL function.

**See also** [ERR](#), [ON ERROR](#)

## Examples

The following is an ON ERROR trap routine.

*When an error occurs and it is a file not found error, check to see if the error location is line 88050. If so, then change the file name to the default name and retry the operation.*

```
890000  ERROR.TRAP:
890010
890020      IF ERR=30          ! File not found error?
890030          IF ERL=88050
890040              FILENAME$ = DEFAULT$
890050              RESUME      ! Try again
890060          IFEND
890070      IFEND
890080
890090      RESUME 0          ! Let BASIC handle it
```



# ERR Function

ERR returns the *error number* of the last program error.

ERR

**Operation** When an execution-time error occurs, the error number of the error is saved. The ERR function returns this saved error code.

**Notes** Refer to Appendix D: “[Error Codes and Messages](#),” starting on page [640](#) for a list of these error codes.

**See also** [ERL](#), [ON ERROR](#)

**Examples**

890000 ERROR.TRAP:

*First, check to see if the error is a BREAK,C event. If so, then ask if the operator is sure.*

*When the error is not a BREAK,C then transfer control to the standard error trap reporting program.*

```
890010
890020      IF ERR=1
890030          PRINT BELL$;
890040          PRINT "DO YOU WISH TO EXIT ";
890050          ANS$ = YESNO$
890060          IF ANS$="Y" THEN RESUME END.RUN
890070          RESUME
890080      ELSE ERR.NBR% = ERR \ ERR.LIN = ERL
890090          LINK "ERRORTRP"
890100          IFEND
```

Statements

---

## ERRMSG\$ Function

ERRMSG\$ displays a message on the bottom line of the screen or window, waits for operator response, and returns the single character response.

**ERRMSG\$( *string-expression* )**

**Operation**      The cursor is positioned to the start of the last display line on the screen or window and the line is erased. The *string-expression* is then displayed at the start of that line followed by a single space character. Processing is then suspended until the operator types a single character response.

After the operator responds, the last line of the screen or window is cleared and the operator's response is returned as the value of the function. The character accepted from the operator is not echoed to screen.

**Notes**              This is one of only two functions that perform output on the screen and return a value. The other is the [YESNO\\$](#) function.

The operator's response can be any key that generates a character, including control keys and function keys. Alphabetic character responses are returned in uppercase.



When an [ON TIMEOUT](#) statement is in effect and the time-out period elapses without a response from the operator, this function is assigned a null string value and the time-out event handler is invoked.

**I/O Redirection**   This statement actually displays the message on the current standard output device (stdout) and gets the input from the current standard input device (stdin). Normally this is the console display and keyboard. However, stdout and/or stdin may have been redirected when the program was invoked:

```
>myprog > text.output < text.input
```

When stdout has been redirected the *string-expression* is displayed on that device or written to that file or pipe. When stdin has been redirected the character accepted by this statement comes from that file, device or pipe.

A program can determine if the stdin or the stdout device has been redirected to a file or device other than the console keyboard and display by using the [SYS.ENV\\$](#) function:

```
IF SYS.ENV$(34,"STDIN") = "Y"
```

The above test is true when stdin has been redirected.

**Restrictions** To avoid scrolling, do not use a *string-expression* that is longer than the screen width or current window width.

**See also** [YESNO\\$](#)

## Examples

*An error has occurred. If it is a file not found error inform the operator and allow operator to abort.*

```
500 ERROR.TRAP:
510
520     IF ERR=30                ! File not found?
530         REPLY$ = ERRMSG$(CRT$("RVON")&"FILE MISSING")
540         IF REPLY$=CHR$(18) ! ctrl+Q entered?
```

*Entry of **Quit** means abort.*

```
550             QUIT 253        ! Exit program
560             ELSE RESUME NEXT.FILE
570             IFEND
580 ELSE RESUME 0
590             IFEND
```

---

# EVENT Function

EVENT returns the status of an event *semaphore*.

EVENT( *semaphore* )

Operation	The current status of semaphore is tested and returned.
Notes	<p>Semaphores have states of true (-1) or false (0) corresponding to the <a href="#">SET</a> and <a href="#">RESET</a> functions. Although the <a href="#">SET</a> and <a href="#">RESET</a> functions return the current status of a semaphore they also change the status of the event. The EVENT function does not change the status of the semaphore, it merely returns the current status.</p> <p>Semaphores may be set from the same program or another task. The main task and subtasks all share the same pool of 64 semaphores.</p> <p>Refer to the <i>MultiUser BASIC Programmer's Guide</i> for a description of multitasking, semaphores, and events.</p>
Restrictions	Semaphores have values ranging from 0–63.
See also	<a href="#">ON EVENT</a> , <a href="#">ON TIMEOUT</a> , <a href="#">RESET</a> , <a href="#">SEMAPHORE</a> , <a href="#">SET</a> , <a href="#">TIMER</a> , <a href="#">WAIT EVENT</a>

Examples	<pre>1230      WHILE NOT EVENT(FLAG%)      ! Event not set yet? 1240          GOSUB DO.SOMETHING.ELSE  ! No--occupy yourself 1250          WEND                    ! Check it again 1260 1270      ! Event has been set by another task, handle it 1280</pre>
----------	---

Statements

---

## EXP Function

EXP returns the value  $e$  raised to the power of a numeric expression.

EXP( *numeric-expression* )

<b>Operation</b>	The value of $e^{\textit{numeric-expression}}$ is returned.
<b>Notes</b>	<p>The value of the constant <math>e</math> is 2.718281828459.</p> <p>The EXP function is the complement of the LOG function.</p>
<b>See also</b>	LOG
<b>Examples</b>	<pre>2000      PRINT LOG(EXP(5))</pre>

# EXT\$ Function

EXT\$ returns a field or subfield of a string.

EXT\$( *string-expression*, *field-number*, *subfield-number* )

**Operation**      The subfield of *string-expression* is extracted from *string-expression*, without its delimiter. The resulting subfield is returned. A *subfield-number* of 0 causes *field-number* substring to be extracted.

**Notes**            String fields can be formatted with fields and subfields with the [INS\\$](#) function. That function builds strings such that the string contains two-level subfields delimited by special characters. The EXT\$ function extracts these subfields for usage as individual strings. The [REP\\$](#) and [DEL\\$](#) functions modify the subfields.

A string with subfields is useful when a related group of items are manipulated as a group and rarely as individual components. For example, an employee name and address record might have prior-year earnings fields for total earned, vacation pay, sick pay, federal withholding, state withholding, etc.. These fields might only be reported in a special report. If the fields are grouped into one string using subfields the normal maintenance and reporting programs would read and write the record using only one variable name for the entire set of fields.

The delimiters used are the special characters with value 221 and 222.

**See also**        [DEL\\$](#), [INS\\$](#), [REP\\$](#)

**Examples**       Refer to the [INS\\$](#) function for another example of the EXT\$ function.

This routine  
extracts the sub-  
fields from the  
record field, saving  
them in individual  
array elements for  
easy access and  
maintenance.

600000

EXPLODE: ! Convert file fields to maintenance fields

600010

600020

FLDS\$(1) = REC\$(1)

600030

FLDS\$(2) = EXT\$(REC\$(2),1,0)

600040

FLDS\$(3) = EXT\$(REC\$(2),2,0)

600050

FLDS\$(4) = EXT\$(REC\$(2),3,0)

600060

FLDS\$(5) = EXT\$(REC\$(2),4,0)

600070

FLDS\$(6) = EXT\$(REC\$(3),1,0)

~

# FILL Statement

FILL is a VDI statement used to fill an area with a color and pattern.

1

FILL [#channel]

2

FILL [#channel:] [color;]  $x_1, y_1; x_2, y_2; x_3, y_3; \dots; x_n, y_n$

Operation	<p><b>Mode 1</b>—The graphics display is cleared by performing a page eject.</p> <p><b>Mode 2</b>—A polygon is drawn on the graphics device with the corners of the polygon at <math>x_1, y_1, x_2, y_2, x_3, y_3, \dots, x_n, y_n</math>. The current line width and color are used; the line style is always 1, solid. The enclosed area is filled with the current fill style, and the current fill color is used unless the <i>color</i> is specified with this statement.</p>
Notes	<p>Refer to the <a href="#">SET LINE</a> and <a href="#">SET FILL</a> statements for a description of the various colors, sizes, and styles of lines and fill patterns available.</p> <p>Refer to the <i>MultiUser BASIC Programmer's Guide</i> for a description of VDI devices, statements, and a complete program example.</p> <p>Mode 2 of the FILL statement operates similar to mode 2 of the <a href="#">PLOT</a> statement, except that the last vertex (<math>x_n, y_n</math>) is connected to the first vertex (<math>x_1, y_1</math>) to close the figure.</p>
Restrictions	<p>Invalid parameters do not cause errors. Invalid coordinates are plotted as the maximum value (32767); invalid colors are set to the maximum color code.</p> <p>Most VDI device drivers do not support polygon fill. For those devices, this statement may operate as a polygon draw without fill.</p> <p>The <i>channel</i> number must refer to an open channel. That channel, or the default VDI1 device, must be graphics capable or the trappable error message number 37 “Graphics not available.” is reported.</p>
See also	<a href="#">PLOT</a> , <a href="#">SET FILL</a>

**Examples**

```

10      OPTION BASE 1, DEGREE
20
30      GOSUB COMPUTE.YADJUST
40
Clear the graphics
display. 50      FILL                ! Start with a clean display page
60
Draw a circle. 70      PLOT CIRCLE 16000,16000;10000
80
85                        ! Compute sides of right triangle
87
Draw a right trian- 87
gle inside the circle. 90      BASE% = 16000+10000*COS(35)
The YADJUST is 90      HEIGHT% = 16000+10000*SIN(35)*YADJUST
used to adjust a 100
possible difference 110
in the device's x and 115                        ! Draw triangle
y scales. 117
120      FILL 16000,16000;BASE%,16000;BASE%,HEIGHT%
130
Draw the right tri- 140      PLOT BAR BASE%-1000,16000;BASE%,16000+1000*YADJUST
angle symbol. 150
~
Refer to the FILL 1000 COMPUTE.YADJUST:
PIE statement for a 1010
discussion of using 1020      DIM VC%(6),VI%(10),VPI%(1),VO%(45),VPO%(6,2)
the VDI statement 1030
to acquire informa- 1040      VC%(1) = 1                ! Command is open
tion about scaling 1050      VC%(4) = 0                ! No inputs
factors. 1060
1070      VDI VC%,VI%,VPI%,VO%,VPO%
1080
1090      YADJUST = ((FLOAT(VO%(1))+1)/(VO%(2)+1))*VO%(4)/VO%(5)
1100
1110      RETURN

```



---

## FILL BAR Statement

FILL BAR is a VDI statement used to draw and fill a rectangle.

**FILL BAR** [*#channel*:] [*color*;] *x<sub>1</sub>*, *y<sub>1</sub>*; *x<sub>2</sub>*, *y<sub>2</sub>*

**Operation**      A rectangular area is drawn and filled. The position and size of the rectangle are defined by the two opposite corners  $x_1, y_1$  and  $x_2, y_2$ . The current line width and color are used to outline the rectangle. The line style is always 1, solid. The current fill style is used to fill the interior of the rectangle. The current fill color is used unless the *color* is specified with this statement.

**Notes**            Refer to the [SET LINE](#) and [SET FILL](#) for a description of the various colors, sizes, and styles of lines and fill patterns available.

Refer to the *MultiUser BASIC Programmer's Guide* for a description of VDI devices, statements, and a complete program example.

The rectangle is drawn with the corners precisely at the coordinates specified. An attempt to draw a square rectangle might be thwarted by a non-uniform horizontal and vertical scale, as used on most consoles and printers. For example:

```
PLOT BAR 5000,5000;10000,10000
```

would be expected to draw a square of size 5000. Because of the mapping of the VDI world coordinate system to the actual device coordinates, the rectangle will not be square unless the device's horizontal and vertical scales are equal.

A program can take into account a difference in the device's horizontal and vertical scaling by using the [VDI](#) statement. Refer to the [FILL PIE](#) statement for an example of this adjustment to device coordinates.

**Restrictions**    Invalid parameters do not cause errors. Invalid coordinates are plotted as the maximum value (32767); invalid colors are set to the maximum color code.

The *channel* number must refer to an open channel. That channel, or the default VDI1 device, must be graphics capable or the trappable error message number 37 "Graphics not available." is reported.

**See also**        [FILL](#), [PLOT BAR](#), [SET FILL](#), [SET LINE](#)

Examples

Get the data items  
to graph.

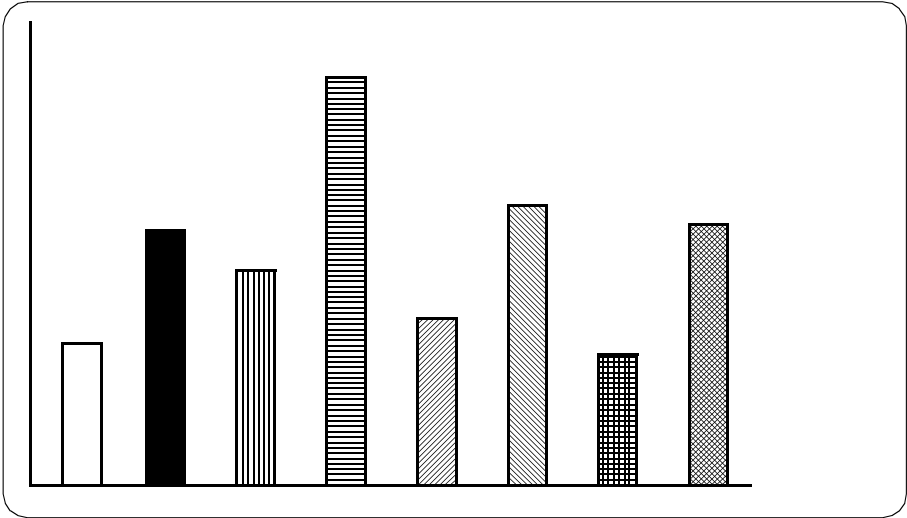
Find largest data  
item to use as a  
scaling factor for  
other data items.  
Plot the graph axes.

Set bar width.

Draw and fill the  
data bars. Each bar  
is filled with a dif-  
ferent pattern.

```
10      OPTION BASE 1
20
30      DIM A(8)
40
50      MAT READ A
60
70      FOR I% = 1 TO 8      ! Determine largest value
80          MAX.VALUE = MAX(MAX.VALUE,A(I%))
90      NEXT
100
110     PLOT 5000,30000;5000,5000;30000,5000 ! Draw axes
120
130     BAR.WIDTH% = 25000/16
140
150     BAR.START% = 5000+BAR.WIDTH%/2
160
170     FOR I% = 1 TO 8
180         SET FILL STYLE I%-1
180         BAR.HEIGHT% = 5000+25000*A(I%)/MAX.VALUE
190         BAR.END% = BAR.START%+BAR.WIDTH%
200         FILL BAR BAR.START%,5000;BAR.END%,BAR.HEIGHT%
210         BAR.START% = BAR.END%+BAR.WIDTH%
220     NEXT
~
9000    DATA 25,55,40,95,30,55,15,45
```

Statements



---

# FILL CIRCLE Statement

FILL CIRCLE is a VDI statement used to draw and fill a circular area.

FILL CIRCLE [#channel:] [color;] x, y; radius

**Operation**      A circle is drawn and filled. The circle is centered at location *x,y* and has a radius of *radius*. The current line width and color are used; the line style is always 1, solid. The circle is filled with the current fill pattern and the current fill color is used unless the *color* is specified with this statement.

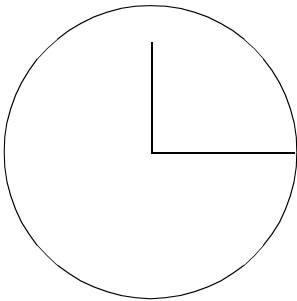
**Notes**            Refer to the [SET FILL](#) and [SET LINE](#) statements for a description of the various colors, sizes, and styles of lines and fills available.

Refer to the *MultiUser BASIC Programmer's Guide* for a description of VDI devices, statements, and a complete program example.

Circles are always drawn round, even when the horizontal and vertical scales of the device are not equal. When the scales are not equal, the points along the circumference of a circle are adjusted to make the circle round.

The only circumference points that are *radius* distance from the center point *x,y* are the points along the horizontal radius. For example:

```
10 FILL CIRCLE 16000,16000;10000
20 PLOT 16000,16000;26000,16000      ! Horizontal radius
30 PLOT 16000,16000;16000,26000      ! Vertical radius
```



Notice that the vertical radius does not reach the circumference. This is due to a difference in scale in the horizontal and vertical directions, typically found on terminals and printers.

Refer to the [FILL PIE](#) statement for a method that a program can utilize to accommodate this nonuniform scaling. Also, refer to the *MultiUser BASIC Programmer's Guide* for a full description of the y axis adjustment.

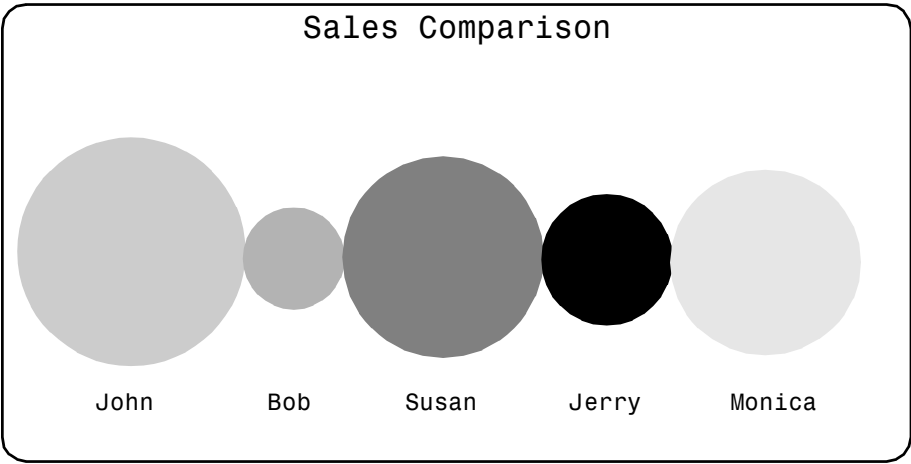
Statements

**Restrictions** Invalid parameters do not cause errors. Invalid coordinates or lengths (*radius*) are plotted as the maximum value (32767); invalid colors are set to the maximum color code.

The *channel* number must refer to an open channel. That channel, or the default VDI1 device, must be graphics capable or the trappable error message number 37 “Graphics not available.” is reported.

**See also** [FILL](#), [PLOT CIRCLE](#), [SET FILL](#)

<b>Examples</b>	10	OPTION BASE 1
	20	
	30	DIM VALUES(5),LABELS\$(5)
	40	
<i>Get the data for the displays.</i>	50	READ TITLE\$
	60	MAT READ LABELS\$ \ MAT READ VALUES
	70	
	80	FOR I% = 1 TO 5
<i>Compute the total amount of all values.</i>	90	TOT.VALUE = TOT.VALUE+VALUES(I%)
	100	NEXT
	110	
	115	! Compute scaling factor
	117	
<i>Calculate the scaling factors for the circles.</i>	120	FACTOR = 30000/(TOT.VALUE*2)
	130	
	140	INITIAL.X% = 0 ! X coordinate of left side
	150	
	160	SET TEXT SIZE 4 ! Size for title
	170	TEXT.X% = 16000-LEN(TITLE\$)*700/2
<i>Display the graph title.</i>	180	TEXT TEXT.X%,28000;TITLE\$
	190	
	200	SET TEXT SIZE 1 ! Size for labels
	210	FOR I% = 1 TO 5
<i>Display circles in sizes comparable to the data values.</i>	220	SET FILL STYLE I%+1
	230	RADIUS% = VALUES(I%)*FACTOR
<i>Each circle is filled with a different pattern. Place the data labels underneath the center of each circle.</i>	240	X% = INITIAL.X%+RADIUS% ! Center coordinate
	250	<b>FILL CIRCLE X%,16000;RADIUS%</b>
	260	TEXT.X% = X%-LABELS\$(I%)*350/2 ! start of text
	270	TEXT TEXT.X%,5000;LABELS\$(I%)
	280	INITIAL.X% = INITIAL.X%+2*RADIUS%
	290	NEXT
	300	
	~	
	9000	DATA "Sales Comparison"
	9010	
	9020	DATA "John","Bob","Susan","Jerry","Monica"
	9030	DATA 60,20,40,30,35
	9040	END



Statements

---

## FILL PIE Statement

FILL PIE is a VDI statement used to draw and fill a “pie-shaped” area.

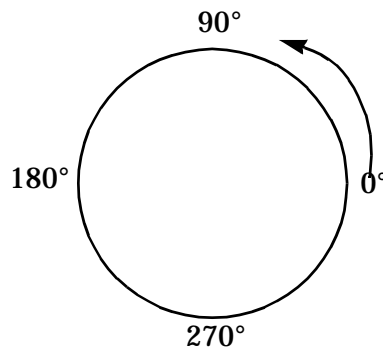
**FILL PIE** [#*channel*:] [*color*;] *x*, *y*; *radius*, *angle*<sub>1</sub>, *angle*<sub>2</sub>

**Operation**      A “pie-slice” is drawn and filled. A pie-slice is a wedge of a circle with the circle centered at location *x,y* and the circle radius is *radius*. The pie-slice arc is drawn counterclockwise from *angle*<sub>1</sub> to *angle*<sub>2</sub>. The current line width and color are used and the line style is always 1, solid. The current fill pattern is used and the current fill color is used unless the *color* is specified with this statement.

**Notes**            Refer to the [SET FILL](#) and [SET LINE](#) statements for a description of the various colors, sizes, and styles of lines available.

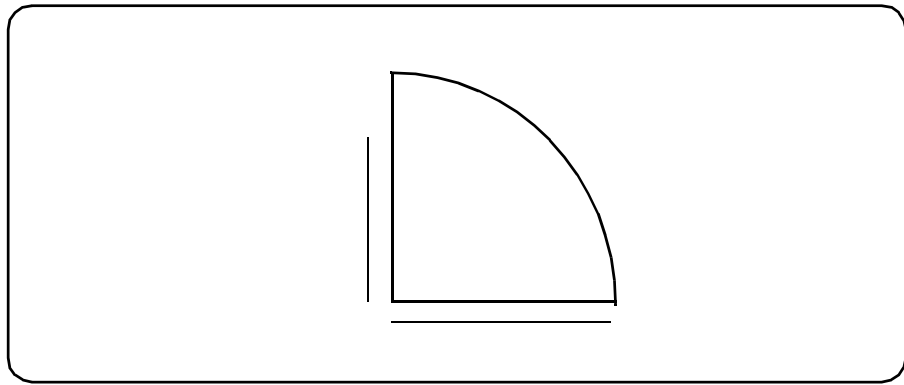
Refer to the *MultiUser BASIC Programmer's Guide* for a description of VDI devices, statements, and a complete program example.

The angle specification is in radians or degrees, depending upon the current setting of the [OPTION RADIAN](#) or [OPTION DEGREE](#). The angle specifications are counterclockwise with the 0 angle at the 3 o'clock position:



Pie slices are always drawn as part of a round circle, even when the horizontal and vertical scale of the device are not equal. When the scales are unequal, the points along the circumference are adjusted to make the arc round. The only circumference points that are *radius* distance from the center point *x,y* are the points on the horizontal radius.

```
10 SET FILL STYLE 4
20 FILL PIE 16000,16000;10000,0,90
30 PLOT 16000,15000;26000,15000 ! Horizontal radius
40 PLOT 15000,16000;15000,26000 ! Vertical radius
```



The two radii drawn are both 10,000 units long. But, because the vertical size of the display area is smaller than the horizontal size, the vertical radius appears shorter.

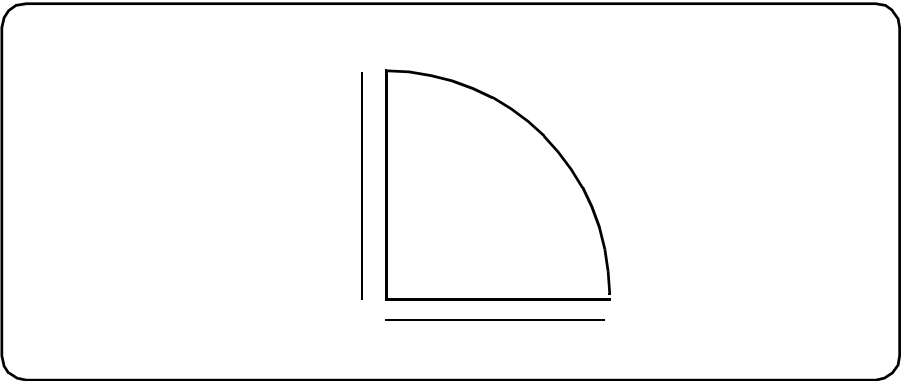
When necessary, the program can determine the difference in scale and make accommodations for it. Use the [VDI](#) statement, open statement, to access the device specifications and capabilities. Some of the output array values returned include the horizontal and vertical scales. Use this information to adjust the vertical coordinate of a point or line drawn on the radius of a circle or arc.

```

10  OPTION BASE 1
20  GOSUB  COMPUTE.YADJUST
30  SET FILL STYLE 4
40  FILL PIE 16000,16000;10000,0,90
50  PLOT 16000,15000;26000,15000           ! Horizontal radius
60  PLOT 15000,16000;15000,16000+10000*YADJUST ! Vertical
~
1000COMPUTE.YADJUST:
1010
1020  DIM VDI.C%(6),VDI.I%(10),VDI.PI%(1)
1025  DIM VDI.O%(45),VDI.PO%(6,2)
1030
1040  VDI.C%(1) = 1           ! Command is open
1050  VDI.C%(4) = 0           ! No input
1060
1070  VDI VDI.C%,VDI.I%,VDI.PI%,VDI.O%,VDI.PO% ! Open
1080
1090  YADJUST = ((FLOAT(VDI.O%(1))+1)/(VDI.O%(2)+1))*
          VDI.O%(4)/VDI.O%(5)
1100
1110  RETURN

```

Statements



Refer to the *MultiUser BASIC Programmer's Guide* for a description of the y coordinate adjustment.

**Restrictions** Invalid parameters do not cause errors. Invalid coordinates or lengths (radius) are plotted as the maximum value (32767); invalid colors are set to the maximum color code; invalid angles are modularized to the range 0–360 degrees or 0– $2\pi$  radians.

The *channel* number must refer to an open channel. That channel, or the default VDI1 device, must be graphics capable or the trappable error message number 37 “Graphics not available.” is reported.

**See also** [FILL](#), [PLOT PIE](#), [SET FILL](#), [SET LINE](#)

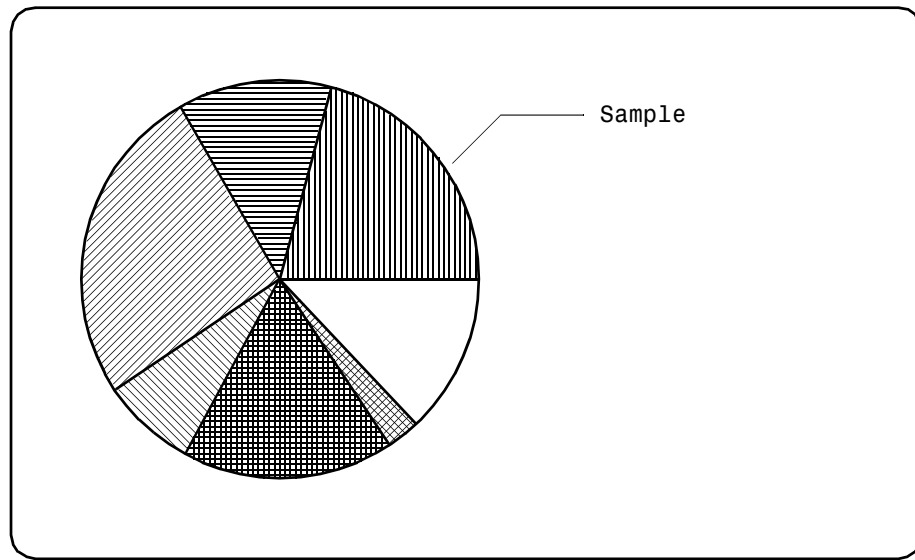
Statements

**Examples**

```
OPTION          10      OPTION BASE 1, DEGREE
DEGREE is used  20
so that angles  30      DIM A(9)
can be specified 40
in degrees instead of
radians.
Get y-scaling factor. 50      GOSUB COMPUTE.YADJUST
60
70      MAT READ A
Draw the pie.     80      OX% = 10000 \ OY% = 16000 ! Circle center
90
```



<i>Highlight the second slice with a pattern and a line drawn from the slice to a text label. COS and SIN are used to compute the point on the circumference that is in the middle of the selected slice.</i>	100	FOR I% = 1 TO 8
	110	SET FILL STYLE I%
	120	FILL PIE 0X%,0Y%;8000,A(I%),A(I%+1)
	130	NEXT
	140	
	150	ANG = A(2)+(A(3)-A(2))/2 ! Compute mid-angle of pie slice
	160	PX1% = 0X%+8000*COS(ANG) ! tangent point at mid-angle
	170	PY1% = 0Y%+8000*SIN(ANG)*YADJUST
	180	PX2% = 0X%+1200*COS(ANG)
	190	PY2% = 0Y%+1200*SIN(ANG)*YADJUST
	200	
 <i>Draw the text, centered on the line from the slice.</i>	210	PLOT PX1%,PY1%;PX2%,PY2%;PX2%+5000,PY2%
	220	
	230	SET TEXT SIZE 4
	240	TEXT PX2%+5300,PY2%-400;"Sample"
	~	
	1000	COMPUTE.YADJUST:
	1010	
<i>Use the VDI statement to access the horizontal and vertical scales.</i>	1020	DIM VC%(6),VI%(1),VPI%(1),V0%(45),VP0%(6,2
	1030	
	1040	VC%(1) = 1 ! Command is open
	1050	VC%(4) = 0 ! No inputs
	1060	
	1070	VDI VC%,VI%,VPI%,V0%,VP0%
	1080	
<i>The FLOAT function is used to ensure that the result is not integerized.</i>	1090	YADJUST = (FLOAT(V0%(1)+1)/(V0%(2)+1))*V0%(4)/V0%(5)
	1100	
	1110	RETURN
	1120	
	9000	DATA 0,25,80,120,215,245,300,315,360
	9010	
	9020	END



---

# FIX Function

FIX returns the integer portion of a numeric value.

FIX( *numeric-expression* )

**Operation**      The fractional portion of *numeric-expression* is truncating and the integer portion is returned.

**Notes**            This function is a synonym to the [IP](#) function.

The FIX (and [IP](#)) function differs from the [INT](#) function only when the *numeric-expression* is negative. FIX always returns the value with the fractional portion truncated. [INT](#) might return a smaller value. For example:

FIX(-12345.6789)   ⇒ -12345  
INT(-12345.6789)   ⇒ -12346

The FIX function returns a numeric value, not an integer. That is, it's value is not limited to the range of an integer.

**See also**        [FLOAT](#), [IP](#)

**Examples**

```
5000 MAKE.CHANGE:
5010
5020     DOLLAR.AMOUNT = IP(AMOUNT)           ! Get whole dollar amt
5030     CHANGE.AMOUNT = 100*FP(AMOUNT)       ! Get change amount
5040
5050     FIFTY.CENTS% = FIX(CHANGE.AMOUNT/50) ! 50¢ pieces
5060     CHANGE.AMOUNT = CHANGE.AMOUNT-50*FIFTY.CENTS%
5070     QUARTERS% = FIX(CHANGE.AMOUNT/25)    ! 25¢ pieces
5080     CHANGE.AMOUNT = CHANGE.AMOUNT-25*QUARTERS%
5090     DIMES% = FIX(CHANGE.AMOUNT/10)       ! 10¢ pieces
5100     CHANGE.AMOUNT = CHANGE.AMOUNT-10*DIMES%
5110     NICKLES% = FIX(CHANGE.AMOUNT/5)      ! 5¢ pieces
5120     PENNIES% = CHANGE.AMOUNT-5*NICKLES%
~
```

Statements

---

## FLOAT Function

FLOAT returns the numeric value of an integer expression value.

FLOAT( *numeric-expression* )

**Operation**      The value of *numeric-expression* is determined and returned as a numeric value (the fractional portion will be .0).

**Notes**            This function is the complement of [FIX](#).

The FLOAT function is performed automatically by all functions and statements that require numeric arguments or parameters, but are given integer values.

**See also**        [FIX](#)

### Examples

*The FLOAT function is used here to force an expression to be numeric. In this syntax, without the FLOAT function, the division of two integers produces an integer.*

```
10 INPUT "Number of lines of text",LINE.COUNT%
20 INPUT "Number of lines per page",LINES.PER.PAGE%
30
40 PRINT "Number of pages required: ";
50 PRINT CEIL (FLOAT (LINE.COUNT%) / LINES.PER.PAGE%)
60
```

*The FLOAT function could have been applied to either variable.*

Statements

---

## FLOOR Function

FLOOR returns the floor of a number. The floor of a number is the largest whole number that is less than or equal to the number.

**FLOOR**( *numeric-expression* )

**Operation**      The floor of *numeric-expression* is determined and returned:

**Notes**            The floor of a whole number is the number itself.

                      The floor of positive fractional numbers is equal to

$\text{IP}(\textit{numeric-expression})$

                      The floor of negative fractional numbers is equal to

$\text{IP}(\textit{numeric-expression}) - 1$

**See also**         [CEIL](#)

### Examples

```
10 INPUT "Number of lines of text",LINE.COUNT%
20 INPUT "Number of lines per page",LINES.PER.PAGE%
30
40 PRINT "Number of complete pages to print: ";
50 PRINT FLOOR(FLOAT(LINE.COUNT%) / LINES.PER.PAGE%)
60
```

---

## FNEND Statement

FNEND marks the end of a multiple-line, user-defined function definition.

### FNEND

**Operation** During execution of a multiple-line function definition, the FNEND statement exits the function and transfers control back to the statement calling the function.

A [BREAK](#) statement, executed as part of a function definition, transfers control to the FNEND statement.

Refer to the [DEF FN](#) statement description for information about the FNEND statement, restrictions, and examples.

**See also** [BREAK](#), [DEF FN](#)

## FOR Statement

FOR marks the beginning of, and defines the parameters for a program loop.

```
1  FOR numeric-var = start TO to STEP increment
2  FOR numeric-var = start TO to
3  FOR var = expression-list
```

---

*expression-list*      »    *expression*[, *expression-list* ]

### Operation

The FOR statement is part of a program structure that starts with the FOR statement and ends with the **NEXT** statement. These two statements may be separated by other statements that are executed with every iteration of the loop.

**Mode 1**—The *numeric-var* is set to *start* and tested against the *to* value. The test that is performed depends upon the sign of *increment*.

When the sign of *increment* is positive the *numeric-var* is tested to see if it is less than or equal to *to*. When the sign of *increment* is negative the *numeric-var* is tested to see if it is greater than or equal to *to*.

When the result of the test is true the statements of the loop are executed, the *numeric-var* is adjusted by *increment* and tested against the *to* value again. This process is repeated until the result of the test is false, at which point control is transferred to the statement following the **NEXT** statement.

It is the **NEXT** statement that performs the increment of *numeric-var* and performs the test, and conditionally repeats the loop.

**Mode 2**—Identical to mode 1 in operation with the *increment* set to the default of +1.

**Mode 3**—The *var* is set to the first value in the *expression-list*. The statements of the loop are executed and the *var* is set to the next value in the *expression-list*. This process is repeated until all values in the *expression-list* are used, at which point control is transferred to the statement following the **NEXT** statement.

It is the **NEXT** statement that assigns the next value from the *expression-list* to *var* and conditionally repeats the loop.

Another way to describe a FOR-NEXT loop is to compare it to straight-line code that produces the same result:

```
FOR INDEX = START.VAL TO END.VAL STEP STEP.VAL
  statements...
NEXT INDEX
```

is equivalent to:

```
      INDEX = START.VAL
      GOTO LOOP.TEST
LOOP.NEXT:
      INDEX = INDEX + STEP.VAL
LOOP.TEST:
      IF SGN(STEP.VAL) < 0
        IF INDEX < END.VAL GOTO LOOP.EXIT
      ELSE IF INDEX > END.VAL GOTO LOOP.EXIT
      IFEND
LOOP:
  statements...
  GOTO LOOP.NEXT
LOOP.EXIT:
```

As is apparent, the FOR-NEXT structure is more simple and direct than the equivalent straight-line code.

The expressions *start*, *to*, and *increment* are evaluated once, at the time the FOR statement is encountered. These original values are used throughout the execution of the FOR-NEXT loop, even if the statements of the loop change the values used in those expressions. For example:

```
END.VAL% = 10 \ STEP.VAL% = 1
FOR I% = 1 TO END.VAL% STEP STEP.VAL%
  PRINT I%
  END.VAL% = 3
NEXT
```

The above loop will execute 10 times even though END.VAL% was changed inside the loop.

To exit a FOR-NEXT loop early, the recommended method is to use the BREAK statement or to set the *numeric-var* to the *to* value and then branch to the NEXT statement. It is not a good programming practice to branch out of a FOR-NEXT loop with the GOTO statement (see Caution).

The only way to exit from a mode three FOR-NEXT loop is with the BREAK statement.

The CONTINUE statement can be used to transfer control to the FOR-NEXT structure's NEXT statement, causing the loop to be repeated with the next value.



## Restrictions

The type of expressions used in *expression-list* must all be the same and they must match the type of *var*. That is, they must all be numeric or all strings.

Every FOR statement must have only one **NEXT** statement. Every **NEXT** statement must have only one FOR statement.

The *numeric-var* or *var* should not be the same variable as is used in another open (or nested) FOR-**NEXT** structure. No error is detected or reported but the program will probably not execute properly. For example:

```
FOR I% = 1 TO 10
  FOR I% = 2 TO 5
    statement...
  NEXT
NEXT
```

The first **NEXT** statement will close the inner FOR-**NEXT** loop when I% reaches 6. The second **NEXT** statement executes and, since I%≤10, the outer loop is repeated which starts the inner loop, which terminates and repeats the outer loop, ad infinitum.

This statement defines the beginning of a program structure. This type of program structure should only be entered at the top (FOR statement) and exited at the bottom (**NEXT** statement). If control is transferred out of this control block via a **RETURN** *line-ref*, a **RESUME** *line-ref*, or a **GOTO** *line-ref*, the loop will be left in an open status.

Branching into the middle of a FOR-**NEXT** program structure is not good programming practice.

## See also

**BREAK**, **CONTINUE**, **NEXT**, **WHILE**

## Examples

```
140  DISPLAY.DATA:
150
160      FOR FLD% = 1 TO LAST.FIELD%
170          PRINT AT$(INX%(FLD%),INX%(FLD%));
180          GOSUB DISPLAY.FIELD
190      NEXT
200
210  RETURN
```

*Here, the alternate form of the FOR statement is used to repeat an operation that cannot be controlled by a numeric variable.*

```
10000  FOUND% = FALSE%
10010  FOR KEY$ = INKEY$,UCASE$(INKEY$)
10020      READ #2,KEY$:RECORD$
10030      FOUND% = EOF(2)
10040      IF FOUND% THEN BREAK
10050  NEXT
```

*The control variable is a record key that is first set to one value, and then to another.*

*Since the intent was to read one of these records, the FOUND% variable is used to determine if the loop should be repeated or if the BREAK statement terminates execution of the FOR-NEXT loop.*

# FORMAT\$ Function

FORMAT\$ returns a formatted number, similar to the way that [PRINT USING](#) prints numbers.

FORMAT\$( *numeric-expression*, *mask-string* )

**Operation**      The value of *numeric-expression* is formatted according to the specifications in *mask-string*.

The *mask-string* may contain the following special symbols:

Characters	Meaning	Example	Output
9	Format number with leading zeros. Can be used to the left or right of decimal point, but leading zeros only apply on left.	999.99	001.23
#	Format number with leading zeros suppressed (except for the 1's position)	###.##	0.23
ZZ	Format number with leading zeros suppressed (except for the 1's position) Identical to # specification except when the value is zero. Z returns spaces only. At least two consecutive Z characters must be used.	ZZZ.ZZ	0.23
+	Specified at the end of number: both positive and negative numbers have a trailing sign. <sup>1</sup>	999.99+	1.23+
-	Only negative numbers are signed. <sup>1</sup>	999.99-	1.23-
,	Use commas to separate thousands, millions. <sup>2,4</sup>	9,999.99	1,234.56
**	Format number with leading zeros replaced with asterisks. <sup>3</sup>	***,***.##	*****1.23

Statements

Characters	Meaning	Example	Output
\$\$	Format number with leading zeros suppressed except for last which is replaced with \$. <sup>3</sup>	\$\$#,###.99	\$1.23
DB	Negative values signed with trailing DB <sup>1</sup>	999.99DB	001.23DB
CR	Negative values signed with trailing CR <sup>1</sup>	999.99CR	01.23CR
>	Negative values surrounded with angle brackets. <sup>1</sup> This notation cannot be used with the '9' specification. It can only be used with 'Z' or '#'. <sup>3</sup>	###.##>	<1.23>
^^	Use scientific notation (E) with 1 digit unsigned exponent. <sup>1</sup>	#####^^	1.2345E1
^^^	Use scientific notation (E) with 1 digit signed exponent. <sup>1</sup>	#####^^^	1.2345E+1
^^^^	Use scientific notation (E) with 2 digit signed exponent. <sup>1</sup>	####.####^^^^	123.456E-30
^^^^^	Use scientific notation (E) with 3 digit signed exponent. <sup>1</sup>	#####^^^^^	1.2345E+127

The following notes refer to the table above.

1. This specification is used only at the end of the number specification.
2. Only one comma is needed in the specification; more are allowed. For example, “#,#####” and “###,###,###” are equivalent.
3. This specification is used only at the beginning of the number specification.
4. Whether commas or periods are used to separate thousands is dependent upon the status of the [OPTION COMMA](#) statement.

#### Notes

The *mask-string* specifies the complete format for the number, including the length of the resulting string. If the number cannot be formatted to fit in that length without losing significant digits (digits to the left of the decimal point) the resulting string will start with a percent character (%) followed by the number formatted normally (as if it were formatted by the [STR\\$](#) function). This string might be longer than the length of *mask-string*.

See also [STR\\$, PRINT USING](#)

## Examples

*Format dollar amounts for subsequent printing. In this case the dollar amounts are maintained as positive values. The call to the **FORMAT** function varies according to the sign that must be applied to the amount.*

```
3230 SELECT TYPE$
3240 CASE "V"
3250     DESC$ = "V O I D" \ AMT$ = ""
3260 CASE "C"
3270     AMT$ = FORMAT$(-AMT, "###,###.##+ ")
3280     BAL = BAL - AMT
3290 CASE "D"
3300     AMT$ = FORMAT$(AMT, "###,###.##+ ")
3310     BAL = BAL + AMT
3320 OTHERWISE
```

*The format includes commas and a trailing sign.*

```
3330     AMT$ = FORMAT$(AMT, "###,###.##+ ")
3340     BAL = BAL + AMT
3350 CEND
```

*The beginning balance format will include a floating dollar sign, commas, and a trailing sign when negative.*

```
3940 PLINE$ = SPACE$(16)&LEFT$("Beginning balance"&
    RPT$(80, ". "), 50)& " &
    FORMAT$(OPENING.BALANCE, "$$###,###.##- ")
```

*Note the usage of the **RPT\$** function in this last line. By using it and the **LEFT\$** function, the leaders are generated and truncated at the end of the description area.*

# FP Function

FP returns the fractional portion of a numeric value.

**FP( *numeric-expression* )**

**Operation**      The value of *numeric-expression* is determined and the integer portion is discarded. The sign of *numeric-expression* is retained as the sign of the returned fractional portion.

**See also**      [FIX](#), [INT](#), [IP](#)

**Examples**

*Here, the FP function is used to get the fractional portion of an angle. This is then multiplied by 60 to compute the minutes of angle represented by the fraction.*

*The same is done for the seconds of angle.*

*The IP function is then used to format the display, using only the whole number portion of the angle and the minutes of angle.*

```
10      OPTION PROMPT "", DEGREE
20
30      INPUT "Enter tangent of an angle: ",VALUE
40      PRINT
50
60      MY.ANGLE = ATAN(VALUE)
60      MY.ANGLE.MIN = FP(MY.ANGLE)*60 ! Determine minutes
70      MY.ANGLE.SEC = FP(MY.ANGLE.MIN)*60 ! and seconds
80
90      MY.ANGLE$ = STR$(IP(MY.ANGLE))&CHR$(248)&
              STR$(IP(MY.ANGLE.MIN))&"' "&
              STR$(ROUND(MY.ANGLE.SEC,4))&"'"
100
110     PRINT USING "The arctangent of 'e is 'e",STR(VALUE),
              MY.ANGLE$
```

**Enter tangent of an angle: 1.25**

**The arctangent of 1.25 is 51°20'24.6903"**

Statements

---

## GCD Function

GCD returns the *Greatest Common Divisor* of two values.

**GCD**( *numeric-expression*<sub>1</sub>, *numeric-expression*<sub>2</sub> )

**Operation**      The largest value that can be divided into each of the values with no remainder is returned.

**Notes**            The sequence of *numeric-expression*<sub>1</sub> and *numeric-expression*<sub>2</sub> is irrelevant. That is, the GCD(A,B)  $\Leftrightarrow$  GCD(B,A).

The GCD of a series of values can be determined by nesting calls to the GCD function: GCD(A,GCD(B,GCD(C,D))) computes the GCD of A,B,C,D.

**Examples**            1000      PRINT **GCD**(45,**GCD**(15,**GCD**(30,90)))

---

## GET Statement

GET retrieves the next byte or character from an input device.

1 GET *variable-list*

2 GET #*channel*: *variable-list*

---

*variable-list* » *variable-name*[, *variable-list*]

**Operation**      **Mode 1**—Gets a character, or characters, from the console keyboard or EXEC stack. The characters are not displayed and no case-mode translation is performed. If stdin is the console, class code translations are performed.

**Mode 2**—Gets a character, or characters, from the open file channel. When the file is the console, operates the same as [Mode 1](#).

**Notes**            Only one character is accepted for each variable name in *variable-list*. Numeric variables receive the 8-bit value of the character from the file; string variables receive a character from the console or file.

No testing is done to determine if a character is actually available on the console or file. When no character is available, a null value is assigned to the variables in the *variable-list*.

**I/O Redirection**   [Mode 1](#) of this statement and [Mode 2](#) when *channel* is 0 actually get the character from the current standard input device (stdin). Normally this is the console keyboard. However, stdin may have been redirected when the program was invoked:

```
>myprog < text.file
```

When stdin has been redirected the text accepted by this statement comes from that file, device or pipe.

A program can determine if the stdin device has been redirected to a file or device other than the console by using the [SYS.ENV\\$](#) function:

```
IF SYS.ENV$(34,"STDIN") = "Y"
```

The above test is true when stdin has been redirected.



**Restrictions**      The *channel* must refer to an open file channel opened with INPUT SEQUENTIAL or UPDATE SEQUENTIAL. (Channel number 0 is always open as the standard input device.)

**See also**            [OPEN](#), [PUT](#), [UNGET](#), [WAIT](#)

**Examples**            809000   RELEASE.HELP.PAGE:

<i>The cursor is positioned to the bottom right corner of the window.</i>	809010	
	809020	PRINT AT\$(WIDTH%,DEPTH%);
	809030	
	809040	REPLY% = 0
	809050	WHILE REPLY%=0
<i>Even though the WAIT is performed, because of event trapping, it is possible for the GET to receive no characters. Thus, the WHILE loop.</i>	809060	WAIT #0 \ <b>GET</b> REPLY%
	809070	WEND
	809080	
	809090	L% = 0
	809100	PRINT CLS\$;
	809110	
	809120	RETURN

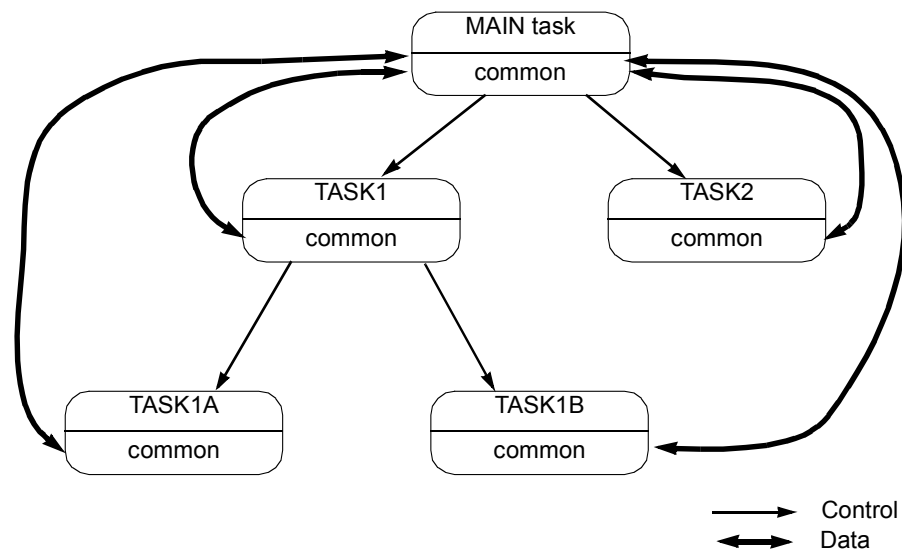
## GET COMMON Statement

GET COMMON copies common data variable values from a parent task to the current child subtask.

**GET COMMON** *variable-list*

**Operation** The values of the variables in *variable-list* are copied from the parent task's **COMMON** data pool.

**Notes** A common means of communicating data between two tasks is via **COMMON** data variables and arrays. The two tasks transfer the data by putting or getting the information from the main task's **COMMON** data pool.



The figure illustrates a main task that has spawned two subtasks, one subtask has spawned two more subtasks. The bold lines show the paths that GET COMMON and PUT COMMON can use to transfer data.

Any of the tasks, except the main task, can perform GET COMMON and PUT COMMON statements. These statements communicate between the current subtask and the main task. Using the above illustration, TASK1A can communicate data to TASK2 only by having TASK1A perform a PUT COMMON and TASK2 performs a GET COMMON, and vice versa.

**Restrictions**      The variables in *variable-list* must be declared as **COMMON** in both the current subtask and in the main or parent task. The trappable error message 68 “COMMON variable "aaaaa" not found in main task.” is reported. Dimensioned arrays should be declared with the same number of elements in both task’s **COMMON** statements or data beyond the smaller array will not be moved with the GET COMMON or **PUT COMMON** statements.

**See also**            **COMMON, PUT COMMON**

**Examples**

	MAIN:	
<i>The main task defines common data and the communication semaphores. Subtasks started.</i>	10 20 30 40 50 60 70	COMMON INFO\$  COMMON.DATA% = SEMAPHORE( "COMMON-DATA" ) COMMON.DATA.CHANGED% = SEMAPHORE( "DATA-CHANGED" )  SUB.TASK1% = ACTIVATE( "TASK1", -1,0) SUB.TASK2% = ACTIVATE( "TASK2", -1,0)
<i>The semaphore is set to signal that it is okay to use the common data path.</i>	80 90 100	 X% = SET(COMMON.DATA%) ! Okay to PUT/GET COMMON
<i>Wait for a subtask to change the common data.</i>	110 120 130 ~ 230	 WAIT EVENT(COMMON.DATA.CHANGED%) ! Wait on subtask  ! Common data has changed, use ...  
<i>Use it, then signal that the path is open again.</i>	240 250 260	GOTO 90 ! Wait for more changes  END
	TASK2:	
<i>The subtask defines common data and gets the communication semaphore numbers.</i>	10 20 30 40 50	COMMON INFO\$  COMMON.DATA% = SEMAPHORE( "COMMON-DATA" ) COMMON.DATA.CHANGED% = SEMAPHORE( "DATA-CHANGED" )
<i>Wait for the common data path to be available.</i>	60 70	WAIT EVENT(COMMON.DATA%) ! Wait for data available
<i>Get INFO\$ from the parent task.</i>	80 90 100 ~	<b>GET COMMON INFO\$</b> ! Get data  ! Make changes to INFO\$
<i>Make changes to INFO\$ and send it back to parent.</i>	220 230	<b>PUT COMMON INFO\$</b> ! Send to parent
<i>Signal parent that common data has changed.</i>	240 250 260 270 280	X% = SET(COMMON.DATA.CHANGED%) ! Let parent know  ! Wait for event requiring INFO\$ change  GOTO 60

# GOSUB Statement

GOSUB calls a subroutine in the current program.

**GOSUB** *line-reference*

**Operation**      The current program location is saved on the program stack and control is transferred to *line-reference*.

**Notes**            The statements at *line-reference* should be a subroutine, terminated with the **RETURN** statement. When the subroutine's **RETURN** statement is executed the location saved on the stack with this GOSUB statement is retrieved and program execution continues with the statement following this GOSUB statement.

Subroutine calls may be nested to any level. That is, a GOSUB may be made to subroutine A; subroutine A may perform a GOSUB to subroutine B; subroutine B may perform a GOSUB to subroutine C; *etc.* When subroutine C performs a **RETURN**, execution continues in subroutine B; when subroutine B performs a **RETURN**, execution continues in subroutine A; when subroutine A performs a **RETURN** execution continues in the mainline.

Subroutines may be called recursively. That is, a GOSUB may be made to subroutine AA; subroutine AA may perform a GOSUB to itself. However, MultiUser BASIC does not provide local variables to subroutines. Therefore, it is the programmer's responsibility to write code that will perform properly when called recursively.

**Restrictions**    *line-reference* must be defined in the program.

**See also**        [CALL](#), [ON GOSUB](#), [RETURN](#), [SUB](#)

## Examples

Perform <b>SETUP</b>	1000	<b>GOSUB SETUP</b> ! Set up variables, display menu mask
and <b>DIS-</b>	1010	<b>GOSUB DISPLAY.MENU</b>
<b>PLAY.MENU.</b>	1020	
	1030	IF SEL\$=" "
Perform the rou-	1040	<b>GOSUB GET.SELECTION</b> ! If no selection, get it
tine <b>GET.SELEC-</b>	1050	IFEND
<b>TION</b> if a selection	1060	
has not already	1070	IF TO.PROG\$ THEN GOTO DISPATCH
been made.	~	

---

## GOTO Statement

GOTO transfers control to another location in the current program.

**GOTO** *line-reference*

<b>Operation</b>	GOTO transfers control to <i>line-reference</i> , unconditionally.
<b>Notes</b>	The line at <i>line-reference</i> may be a <code>do-nothing</code> line such as a remark statement, a blank line, or a line with only a line label on it. Execution will continue with the first executable statement at or following <i>line-reference</i> .
<b>Restrictions</b>	<p><i>line-reference</i> must be defined in the program.</p> <p>A program cannot branch into a <a href="#">FOR-NEXT</a> structure with the GOTO statement unless that structure was exited early and left in the “open” status. When this is attempted the error message “NEXT with no active FOR at line ...” displays.</p> <p>Although it is legal to GOTO any other location, in general the GOTO statement should not be used to transfer control into a program structure such as a subroutines, function definitions, <a href="#">IF-IFEND</a>, <a href="#">FOR-NEXT</a>, <a href="#">SELECT-CEND</a>, <a href="#">WHILE-WEND</a>, or event handling routines (a routine terminated with a <a href="#">RESUME</a>). These types of program structures should only be entered at the top and exited at the bottom. If control is transferred into these program structures with the GOTO statement, the results may be undefined.</p> <p>For the same reason, the above program structures, except for <a href="#">IF-IFEND</a> and <a href="#">SELECT-CEND</a>, and <a href="#">WHILE-WEND</a>, should not be branched out of either.</p>
<b>See also</b>	<a href="#">GOSUB</a> , <a href="#">ON GOTO</a> , <a href="#">ON EVENT</a> , <a href="#">ON ERROR</a> , <a href="#">ON KEY</a>

Examples

*If the record was not found on last read then position to end of file and read the previous record.*

```
13000 READ.PREV.CUSTOMER:
13010
13020     READPREV #1,REC$(1): REC$(2),REC$(3),REC$(4)
13030
```

*The READ.LOOP% variable is used to prevent an infinite loop when the previous record read fails on an empty file.*

```
13040     IF EOF(1) AND NOT READ.LOOP% ! Not found?
13050         READ.LOOP% = TRUE% ! Prevent infinite loop on
            empty file
13060         READ #1,RPT$(INL%(1),"~"): REC$(2) ! Position to end
13070         GOTO READ.PREV.CUSTOMER ! Read record prior to end
13080         IFEND
13090
13100     RETURN
```

---

## HEX, HEXOF\$ Functions

HEX returns the numeric value of a hexadecimal character string.

HEXOF\$ returns a hexadecimal character string of a numeric expression.

**HEX**( *string-expression* )

**HEXOF\$**( *integer-expression* )

**Operation**      HEX analyzes the *string-expression* looking for hexadecimal digits (0–9 and A–F). When the first non-hexadecimal digit is found, analysis stops and the string of digits is evaluated as a numeric value expressed in base 16.

HEXOF\$ converts the *integer-expression* to a four-digit hexadecimal character string. For instance, HEXOF\$(43) returns "002B".

**Notes**            The HEX and HEXOF\$ functions are complements of each other:

A = HEX(HEXOF\$(A))

Only the last four hexadecimal digits of *string-expression* are used. For example:

HEX("123456") ⇔ HEX("3456")

**Restrictions**    HEX returns integer values (–32768 to +32767).

### Examples

```
10 INPUT "Enter hexadecimal value",HEX.VALUE$
20 PRINT
30 PRINT HEX.VALUE$;"H = ";HEX(HEX.VALUE$)
40 PRINT
50 INPUT "Enter numeric value: ",VALUE
60 PRINT
70 PRINT USING "##,### = 'LLLH = 'LLLLLO = 'LLLLLLLLLLLLLLLLB",
    VALUE,HEXOF$(VALUE), OCTOF$(VALUE), BINOF$(VALUE)
```

Enter hexadecimal value? 789A

789AH = 30874

Enter numeric value: 30874

30,874 = 789AH = 0742320 = 0111100010011010B

---

## IF Statement

The IF statement performs conditional execution of a statement or series of statements.

- 1 **IF** *relational-expression* **THEN** *statement...*
- 2 **IF** *relational-expression* **THEN** *statement...* **ELSE** *statement...*
- 3 **IF** *relational-expression*

**Operation**      The *relational-expression* is evaluated and tested for false (0) or true (non-zero).

**Mode 1**—When the *relational-expression* is true the statement or statements following the keyword **THEN** are executed. When the *relational-expression* is false, the statement or statements following the keyword **THEN** are skipped and control continues with the next line of the program.

**Mode 2**—When the *relational-expression* is true, the statement or statements following the keyword **THEN**, up to the keyword **ELSE**, are executed. When the *relational-expression* is false the statement or statements following the keyword **THEN** are skipped and the statement or statements following the keyword **ELSE** are executed.

**Mode 3**—Marks the beginning of a multiple-line IF-**IFEND** program structure. The general form of this structure is:

```
IF relational-expression
   then-statement...
ELSE else-statement...
IFEND
```

The execution is similar to modes one and two except that the conditional execution does not stop at the end of the IF statement line. The *then-statement* is optional and the *else-statement*, along with the **ELSE** statement, is optional.

The following are all proper forms of the IF statement:

```
IF relational-expression
   statement...
IFEND
IF relational-expression
ELSE statement...
IFEND
IF relational-expression
   statement...
```



```

ELSE
    statement...
IFEND

```

When *relational-expression* is true, all of the statements following the IF statement, up to the matching ELSE or IFEND statement, are executed and the statements following the ELSE statement are skipped. A false *relational-expression* causes the statements following the IF statement to be skipped until the ELSE statement; the statements following ELSE are executed.

## Notes

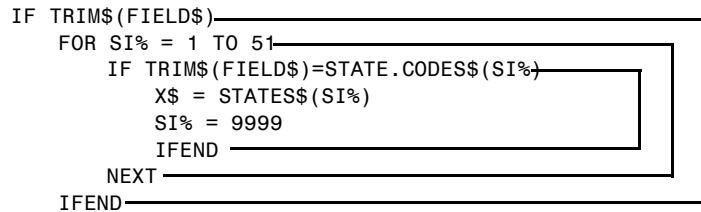
The *relational-expression* may be any valid relational-expression described in the Fundamentals chapter. Additionally, it may be a string-expression. When a string-expression is used the meaning is *string-expression* <> "".

Since IF statements may be nested, it is important to have matching sets of IF statements and IFEND statements.

```

IF TRIM$(FIELD$)
  FOR SI% = 1 TO 51
    IF TRIM$(FIELD$)=STATE.CODES$(SI%)
      X$ = STATES$(SI%)
      SI% = 9999
    IFEND
  NEXT
IFEND

```



The **INDENT** command helps find mismatched IF-IFEND, FOR-NEXT, WHILE-WEND statements.

During program entry and edit, the syntax analyzer allows you to enter statements like:

```
IF A=B THEN 1000 ELSE 3000
```

This will be accepted and maintained as if you had entered:

```
IF A=B THEN GOTO 1000 ELSE GOTO 3000
```

**Restrictions** Multiple-line IF statements must be terminated with the **IFEND** statement.

Do not use the **NEXT**, **WEND**, **FNEND** or **END SUB** statements as the object of an **ELSE** or **THEN** statement.

The object of a single-line IF statement (either the THEN or ELSE clause) cannot be an incomplete programming structure. For instance, the following is invalid:

```
IF A THEN FOR I%=1 TO A
```

Trying to execute this statement will be detected with the error message "Incomplete FOR ... NEXT structure in single line IF at line."

When a **FOR**, **IF**, **SELECT** or **WHILE** statement is an object of a single-line IF statement then the terminating **NEXT**, **IFEND**, **CEND** or **WEND** statement must also be part of the object. For instance:

```
IF A THEN FOR I%=1 TO A \ PRINT I% \ NEXT
```

**See also** [ELSE](#), [IFEND](#), [THEN](#)

**Examples**

```
Display the mes-      1300      IF MSGS%
sage when MSG%       1310      PRINT "YOU HAVE MAIL WAITING";
is not zero.         1320      IFEND
                    ~
                    ~
Read a record.       20000     READ #1,OPNAME$: OPINIT$,PRIVLEV%,HELPSS$
                    20010
                    20020
If the record was    20020     IF EOF(1)
not found set an     20030      ERROR.MSG$ = "ACCOUNT NAME INVALID"
error message and    20040      GOTO GET.ACCOUNT
branch to input       20050     ELSE
routine. Otherwise,  20060      IF HELPSS$="Y" OR HELPSS$="1"
set the PROMPT%      20070      PROMPT% = TRUE%
variable according   20080      ELSE PROMPT% = FALSE%
to the HELPSS$       20090      IFEND
argument.           20100     IFEND
                    ~
```

---

# IFEND Statement

IFEND marks the end of a multiple-line IF statement structure.

IFEND

Operation	Terminate the conditional execution of a multiple-line IF statement program structure.
Notes	The IFEND statement is only valid as the terminating statement of a multiple-line IF statement program structure.
Restrictions	There must be one, and only one, IFEND statement for every multiple-line IF statement.
See also	ELSE, IF, THEN

Examples

```
~
1040      IF SEL$= " "
1050          GOSUB GET.SELECTION
1060          IFEND
1070
1080      IF TO.PROG$
1090          GOTO DISPATCH
1100          IFEND
~
2980      IF PARAM.FLAG%=99
2990          WINDOW SELECT 10, UPDATE OFF
3000          PRINT CLS$;
3010          MENU.FLAG% = FALSE%
3020          IFEND
~
```

---

## INCLUDE Statement

The INCLUDE statement incorporates another MultiUser BASIC source program into the current program.

INCLUDE *program-name*

**Operation**      The indicated source program is incorporated into the current program at the current position.

**Notes**            The line numbers of the included program are separate from the current program. The line numbers in the included program cannot be referenced from outside of the included program section.

All global line labels in the included program may be referenced from the current program.

*Local line labels* may be defined in the included program section. These local line labels may not be referenced from outside of the included program section. Local line labels are defined with a terminating double colon. For instance:

```
1000 LOCAL.LABEL::
1010
1020 !The preceding label can only be referenced within the
1020 !current included program.
```

Local line labels are treated as local line labels only when they are defined inside of an included program, in a subprogram or in a function definition.

Included programs may themselves include other programs.

A program may include the same include program several times.

**Advantages**      The advantage of the INCLUDE statement, particularly over the [MERGE](#) command, is that the included program is not copied to the current program, only a “read-only” reference is made to the included source program.

Changes or updates to the include source program are automatically incorporated in all programs that use the include program the next time they are executed (interpreter) or the next time they are compiled.

Include programs provide source code management for:

- ▶ User-defined functions
- ▶ Subprograms
- ▶ [IOLIST](#) declarations
- ▶ Variable declarations and initializations
- ▶ Other frequently used routines or code

**Special Note** This statement operates as a compiler or interpreter directive. The included program is not merged with the current program and the included program cannot be modified with the interpreter unless it is loaded as a separate program.

When a program is compiled or run in the interpreter the current version of the included program is incorporated as part of the main program. Making a change to the included program will not affect the compiled program unless it is recompiled nor will the changes affect the interpreted program until a [RUN](#) command is used.

**MAKE Note** When you use the MAKE command to maintain your programs, be sure to update the makefile any time that you add a new INCLUDE statement to your program.

**See also** [MERGE](#) command

#### Examples

```

1000 INITIALIZE:
1010
1020     INCLUDE "std_defs"           ! Standard variable defini-
        tions
1030
1040     INCLUDE "keydefs"           ! Keyboard key definitions
1050
1060     INCLUDE "colors"           ! Define color names
1070
1080     ON KEY (F9.KEY%) GOTO END.OF.GAME    ! Trap F9 key
...

```

# INF Function

INF returns the largest value maintained by the system.

INF

**Operation** The largest value maintained by the system is returned.

**Notes** The actual value returned depends upon the current status of the [OPTION BCD](#) or [OPTION IEEE](#) statement:

OPTION	
BCD	IEEE
1 <sup>-126</sup>	2.225073858507 <sup>-308</sup>

This is a very large value. Subtracting small values from this large value has no effect as the number of places for significant digits is much less than the value of INF.

Numeric fields in files are always maintained with BCD. Writing a large IEEE value to a file record will cause the BCD INF value to be written.

**See also** [EPS](#), [OPTION BCD](#), [OPTION IEEE](#)

**Examples** Here, the INF function is used to get a large value that will cause the loop to exit when the NEXT statement is encountered.

~

4530

4540

4550

4560

~

5000

5010

FOR I% = 1 TO MAX.ITEMS%

IF ARRAY\$(I%)=""

I% = FIX(**INF**)-1

ELSE

IFEND

NEXT



! Done with loop

*Note that the FIX function is used before I is subtracted.*

# INP Function

INP returns the control character used to terminate the last input from the keyboard.

INP

Operation	The last control character value used to terminate entry to an <a href="#">INPUT</a> , <a href="#">LINPUT</a> or <a href="#">LINPUT USING</a> statement is returned.
Notes	<p>The  key sets the INP value only when it is entered at the beginning of the input string. The INP value is set to zero when the is used to terminate input at a position other than the beginning of the input string.</p> <p>The values corresponding to the various function and control keys is determined by the class-code definition of the console.</p> <p>When input is terminated due to event trapping (<a href="#">ON EVENT</a>, <a href="#">ON KEY</a> or <a href="#">ON MOUSE</a>) the INP value is set as if the  key were used to terminate input at the position of the cursor when the event occurred. When the input is terminated due to timed-input event trapping (<a href="#">ON TIMEOUT</a>) the INP value is always set to zero.</p>
Restrictions	When the standard input device has been redirected away from the console keyboard, this function always returns a zero.
See also	<a href="#">ON.KEY.TOKEN</a>

## Examples

	10070 GET.KEY.IN:
	10080
	10090 PRINT AT\$(INX%(FLD%),INY%(FLD%));
	10100 LINPUT USING RPAD\$(REC\$(FLD%),INL%(FLD%)),KEY\$
	10110
	10120 SELECT
If the operator responds with an up or down arrow then the key is used to look for the prior or next record in the file.	10130 CASE <b>INP</b> =10 ! Down arrow = next record
	10140 IF KEY\$ THEN READ #1,KEY\$: X\$
	10150 GOSUB READ.NEXT.CUSTOMER
	10160 CASE <b>INP</b> =11 ! Up arrow = prev record
	10170 IF KEY\$ THEN READ #1,KEY\$:X\$
	10180 GOSUB READ.PREV.CUSTOMER
	10190 CASE <b>INP</b> =13 AND KEY\$<>" " ! Return = read rec
	10200 REC\$(1) = KEY\$
	10210 GOSUB READ.CUSTOMER
	10220 CASE <b>INP</b> =0 AND KEY\$<>" " ! Key entered with CR
	10230 REC\$(1) = KEY\$
	10240 GOSUB READ.CUSTOMER
	10250 CEND

Statements

# INPUT Statement

The INPUT statement accepts data from the console or an open, input file or device.

1 INPUT *input-list*

2 INPUT *prompt-text*, *input-list*

3 INPUT #*channel*: *input-list*

4 INPUT #*channel*, *key*: *input-list*

---

*input-list* » IOLIST *listname*  
*variable-list*

*variable-list* » *variable-name*[, *variable-list*]  
*array-reference*[, *variable-list*]

*array-reference* » *array-name*( *subscript*<sub>1</sub> [, *subscript*<sub>2</sub>] )

Statements

**Operation**

**Mode 1**—Accept a line of data from the console, extracting one or more fields. The prompt for the input is the default “?” unless a different prompt string is specified by an **OPTION PROMPT** statement. Characters accepted are echoed to the console.

**Mode 2**—Accept a line of data from the console, extracting one or more fields. The prompt for the input is the *prompt-text* followed by the default “?” or, *prompt-text* followed by the prompt string specified by an **OPTION PROMPT** statement. Characters accepted are echoed to the console.

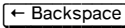



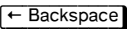

**Mode 3**—Accept one record of data from the sequential access file *channel*, extracting one or more fields. No prompt is output to the file and the accepted characters are not echoed to the file unless it is the console. The *channel* refers to a disk file, tape file, or a device such as “CON” or “COM.”

**Mode 4**—Accept one record of data from the direct, keyed or indexed access file *channel*, extracting one or more fields. No prompt is output to the file and the accepted characters are not echoed to the file or the console.



For **Mode 1** and **Mode 2**, and for **Mode 3** when the file is “CON,” the input may come from EXEC language &STACK statements. If current BASIC program was invoked by an EXEC program that placed data on the EXEC stack, that data will be read by this statement. After all of the data on the stack is read, data will come from the console.

The general operation of this statement is:

- 1. Display the *prompt-text* (**Mode 2** only).
- 2. Display the prompt string (**Mode 1** and **Mode 2** only).
- 3. Locate and read the specified record (**Mode 4** only).
- 4. Accept one line or record of ASCII text, saving the data in an internal buffer. For modes other than **Mode 4**, a record or line is a string of data terminated by the carriage return character.
- 5. If the input is from the console the current status of the **OPTION** CASE statement controls the case mode of the characters input and echoed to the console.
- 6. If the input is from the console, the editing key  is allowed to make corrections to the input prior to pressing . A + may be entered to erase all of the current input and to move the cursor back to the first input character position. A blank line may be terminated with any control character other than . The INP function is set to the value of the control character used.
- 7. For console input, termination of field entry (either by pressing  or using a control character in the first position) causes a CR,LF to be echoed to the screen.
- 8. After the end of the line or record is received, the buffer is scanned for field separators.

Different characters are used to separate fields in the input line, depending upon the status of **OPTION** COMMA or **OPTION** NOCOMMA.

OPTION	
NOCOMMA	COMMA
,	;

- 9. The individual fields identified are then edited according to the variable type (string or numeric) of the corresponding variable in the *variable-list*, and assigned to the variables.

**Numeric fields:** Any nonnumeric character (any character other than a digit, period, sign, or exponent specifier) terminates the number specification, causing the number to the left of the nonnumeric character to be interpreted as the value of the number.

**String fields:** Leading and trailing spaces are removed and the remainder is assigned to the variable. If the field starts with a quotation mark, either single or double, the characters after the quotation mark, up to the matching close quote or end of field, are assigned to the variable.

For example:

```
INPUT A,B,C,D
```

Data input:	<u>1234</u> ,	<u>567.89</u> ,	<u>-654.32</u> ,	<u>987</u>
	↓	↓	↓	↓
	A	B	C	D

```
INPUT NAME$, ADDR$
```

Data input:	<u>ABC Manufacturing</u> ,	<u>"1234 South Main, Suite 321"</u>
	↓	↓
	NAME\$	ADDR\$

When there are more variables in *variable-list* than there are fields in the input line or record, the extra variables are set to nulls. When there are more fields in the input line or record than there are variables in *variable-list*, the extra fields are ignored.

When no record exists matching the *key* (Mode 4 only), the EOF indicator is set and the variables in the *variable-list* are set to nulls.

If an **ON EVENT**, **ON KEY**, **ON MOUSE** or **ON TIMEOUT** is in effect, an INPUT statement may be terminated without entering a carriage return.



When the input is from the console and an **ON TIMEOUT** statement is in effect and the operator does not enter any characters or editing keys before the time-out period elapses, the INPUT statement is terminated with the data already entered by the operator assigned to the *variable-list*. The time-out event handler is then invoked. As each character or editing key is entered the time-out timer is reset.

The *input-list* may be **IOLIST** *listname*. The *listname* must be a previously defined **IOLIST** list name. It specifies the variables to be input and the sequence that those variables are input. Refer to the **IOLIST** statement.

Use the [MAT INPUT](#) statement to input complete arrays.

Reading a record from a file opened for UPDATE access causes that record to be locked so that other users are denied access until this record lock is released. Unless the file channel is opened with the [MULTILOCK](#) option, any attempt to input a record from the file channel releases all prior locks placed on records in this file channel. When the [MULTILOCK](#) option is used then other record locks are not released until a record is written to the file channel, the file channel is closed or the [UNLOCK](#) statement is used to release all of the locks on the file channel.

**I/O Redirection** [Mode 1](#) and [Mode 2](#) of this statement and [Mode 3](#) and [Mode 4](#) when *channel* is 0 actually get the input from the current standard input device (stdin). Normally this is the console keyboard. However, stdin may have been redirected when the program was invoked:

```
>myprog < text.file
```

When stdin has been redirected the text accepted by this statement comes from that file, device or pipe. The prompt character or *prompt-text* is output to the stdout device which may have been redirected. When stdout is redirected, the text accepted is not echoed to the standard output device.

A program can determine if the stdin or stdout device has been redirected to a file or device other than the console by using the [SYS.ENV\\$](#) function:

```
IF SYS.ENV$(34,"STDIN") = "Y"
```

The above test is true when stdin has been redirected.

**Restriction** The *prompt-text* may only be specified with a *string literal expression*, that is, a string expression that starts with a string literal. For example:

```
INPUT "Label",A$
INPUT "Item "&STR$(I%),A$
INPUT ""&PROMPT.STRING$,A$
```

The *channel* must refer to a file opened with [INPUT](#) or [UPDATE](#). Refer to the [OPEN](#) statement for details.

When inputting from a direct access file, the *key* must be a numeric expression; inputting from a keyed or indexed access file, the *key* must be a string expression.

Data read from a disk file should not be data that was written with the [WRITE](#) or [MAT WRITE](#) statements. Inputting from a formatted record reads all of bytes of the record, up to the first binary zero, and assigns that string of characters to the first variable in *variable-list*. The remaining variables

are assigned the null value. Use the READ, READNEXT, READPREV, [MAT READ](#), [MAT READNEXT](#) or [MAT READPREV](#) statements to retrieve records of this type.

**See also** [GET](#), [LINPUT](#), [LINPUT USING](#), [MAT INPUT](#), [ON TIMEOUT](#), [OPEN](#), [OPTION](#), [SYS.ENV\\$](#)

## Examples

*Open a customer name and address data file and a work file.*

```
10 OPEN #1:"DATA.FILE" INPUT KEYED
20 WORKNAME$ = SYS.ENV$(20,"WORK.FILE###@")
30 OPEN #2: WORKNAME$, OUTPUT SEQUENTIAL, QUOTE
40 IOLIST NAME.RECORD = ADDR$, CITY$, STATES$, ZIP$
50 IOLIST WORK.RECORD = CITY$, ZIP$, ADDR$, NAME$, STATES$
60
```

*Select the customers from Oregon and print the selected records to the work file.*

```
70 WHILE NOT EOF(1)
80     IF STATE$="OR"
90         PRINT #2: IOLIST WORK.RECORD
100        IFEND
110        READNEXT #1,NAME$: IOLIST NAME.RECORD
120        WEND
130
```

*Sort the workfile by city, zip, address, and name.*

```
140 CLOSE #1 \ CLOSE #2
150
160 SORTNAME$ = SYS.ENV$(20,"SORT.FILE###@")
170 SYSTEM "SORT -o "&SORTNAME$&" "&WORKNAME$
180
```

*Open the sorted work file and INPUT the records from that file. The original work file was created with option QUOTE so the fields will be separated with commas.*

```
190 OPEN #1: SORTNAME$, INPUT SEQUENTIAL
200
210 INPUT #1: IOLIST WORK.RECORD
220
230 PRINT "Name";TAB(32);"Address";TAB(63);"City";TAB(79);"Zip"
```

*Display the sorted work file records on the screen.*

```
240 WHILE NOT EOF(1)
250     PRINT NAME$;TAB(32);ADDR$;TAB(63);CITY$;TAB(79);ZIP$
260     INPUT #1: IOLIST WORK.RECORD
270     WEND
280
290 CLOSE #1
```

---

# INS\$ Function

INS\$ returns a string with a field or subfield inserted into another string.

INS\$( *string-exp*, *field-number*, *subfield-number*, *substring* )

**Operation**      The *substring* is inserted into *string-exp* following the subfield specified by *field-number*, *subfield-number*. The resulting string is returned.

**Notes**              String fields can be formatted with fields and subfields with the INS\$ function. That function builds strings such that the string contains two-level subfields delimited by special characters. The EXT\$ function extracts these subfields for usage as individual strings. The REP\$ and DEL\$ functions modify the subfields.

A string with subfields is useful when a related group of items are manipulated as a group and rarely as individual components. For example, an employee name and address record might have prior-year earnings fields for total earned, vacation pay, sick pay, federal withholding, state withholding, *etc.* These fields might only be reported in a special report. If the fields are grouped into one string using subfields the normal maintenance and reporting programs would read and write the record using only one variable name for the entire set of fields.

The delimiters used are special characters with value 221 and 222.

**Advantages**      The principal advantages to using the subfield functions DEL\$, EXT\$, INS\$ and REP\$ are that the number of fields, the sequence of fields, and the meanings of the fields in a record do not have to be defined at the time the programs are written. In fact, with proper design, the layout of the record can be easily changed without making any changes to the program logic. (The layout of the record would have to be defined by data values set outside of the program.)

**Disadvantages**    The principal disadvantage to using these functions is a slight decrease in performance and additional memory requirements. In addition, the program references to the subfields are not as self-documenting as referring to individual fields with descriptive names.

**See also**            DEL\$, EXT\$, REP\$

Statements

Examples

*This code builds a string with 11 sub-fields. The first field is the name, the next five fields is a billing address, and the last five fields is the same address.*

```
2000 ADDR$ = IN$$ (NAME$,1,0,ADDR1$)
2010 ADDR$ = IN$$ (ADDR$,2,0,ADDR2$)
2020 ADDR$ = IN$$ (ADDR$,3,0,CITY$)
2030 ADDR$ = IN$$ (ADDR$,4,0,STATES$)
2040 ADDR$ = IN$$ (ADDR$,5,0,ZIP$)
2050 REC$ = IN$$ (CONAMES$,1,0,ADDR$) ! Save as billing address
2060 REC$ = IN$$ (REC$,2,0,ADDR$) ! And as ship to address
~
```

Statements

# INT Function

INT returns the largest whole number that is less than or equal to a numeric expression value.

INT( *numeric-expression* )

**Operation**            The largest whole number that is less than or equal to *numeric-expression* is returned. This is synonymous with the [FLOOR](#) function operation.

**Notes**                The INT function differs from the [IP](#) function only when the *numeric-expression* is negative. [IP](#) will always return the value with the fractional portion truncated. INT might return a smaller value. For example:

IP ( -12345.6789 )        ⇒ -12345  
INT ( -12345.6789 ) ⇒ -12346

The INT function does not return an integer value; it returns a numeric value that is a whole number. That is, the value of the INT function is not limited to the range of an integer variable.

**See also**            [FIX](#), [FLOOR](#), [IP](#)

## Examples

*Without the INT function the result might have been rounded up when it is displayed. Also, because we want the result to be displayed to the nearest penny, the program should use that value for its subsequent calculations.*

*In this code the INT function is used to truncate the result to the nearest penny.*

```
1250        INTEREST = INT(100*PRINCIPAL*RATE)/100 ! Truncate
1260        PRINCIPAL = PRINCIPAL - (PAYMENT - INTEREST)
```

Statements

# IOLIST Statement

The IOLIST statement defines a record structure or list of variables that may be used in input and output statements.

**IOLIST** *listname* = *variable-list*

---

*variable-list*       »   *variable-name* [, *variable-list*]  
                          *array-reference* [, *variable-list*]

*array-reference*   »   *array-name*( *subscript* )  
                          *array-name*( *subscript*<sub>1</sub>, *subscript*<sub>2</sub> )

**Operation**       The list of variable names and array elements is saved for subsequent usage by input and output statements.

**Notes**           The subscripts for array references are interpreted with the value for the subscript at the time the IOLIST statement is executed, not when the *list-name* is used in an input or output statement. For instance:

```
N% = 15
IOLIST DATA.RECORD = NAME$, REC$(N%), FIELD$(4,N%)

N% = 10
READ #14,KEY$: IOLIST DATA.RECORD
```

When the READ statement is executed the record is read into NAME\$, REC\$(15) and FIELD\$(4,15) because the value of N% was 15 at the time the IOLIST statement executed.

A *listname* declared in the main program area can be used by all function definitions and subprograms (without declaring it as [SHARED](#)).

An IOLIST *listname* may only be used in [INPUT](#), [PRINT](#), [PRINT USING](#), [READ](#), [READNEXT](#), [READPREV](#) and [WRITE](#) statements.

**Restrictions**    An IOLIST defined in a function definition or subprogram is *local* to that function definition or subprogram.

The *listname* is not a variable name and is therefore not a string, integer or float. It is not terminated with a \$ or % character.

Statements



The *listname* may not be passed as an argument to a function, subprogram or C language routine and it may not be the object of a [COMMON](#), [LOCAL](#), [SHARED](#) or [STATIC](#) statement.

**See also** [INPUT](#), [PRINT](#), [PRINT USING](#), [READ](#), [READNEXT](#), [READPREV](#), [WRITE](#)

**Examples**

```
1000      IOLIST CUST.RECORD = NAME$, ADDRESS$, CITY$, STATES$,  
          ZIP.CODE$, BALANCE, LAST.PAYDATES$, LAST.INVDATE$  
~  
12350 READ.CUSTOMER:  
12360  
12370      READNEXT #80, CUSTOMER$: IOLIST CUST.RECORD  
12380  
12390      RETURN
```

See also the example for the [INPUT](#) statement.

# IP Function

IP returns the integer portion of a numeric value.

**IP**( *numeric-expression* )

**Operation**      The value of *numeric-expression* is determined and the fractional portion is discarded with the result returned.

**Notes**            The IP function differs from the [INT](#) function only when the *numeric-expression* is negative. IP always returns the value with the fractional portion truncated. [INT](#) might return a smaller value. For example:

IP(-12345.6789)    ⇒ -12345  
INT(-12345.6789)⇒ -12346

The IP function returns a numeric value, not an integer. That is, it's value is not limited to the range of an integer.

**See also**        [CEIL](#), [FIX](#), [FLOOR](#), [FP](#), [INT](#)

## Examples

Statements

*This example accepts an angle specified in degrees.*

*The angle is then translated into degrees, minutes, and seconds using the FP function.*

*The IP function is used to discard the fractional portions of the values.*

```
10      OPTION PROMPT " "
20
30      INPUT "Enter angle in degrees: ",MY.ANGLE
40      PRINT
50
60      MY.ANGLE.RAD = RAD(MY.ANGLE) ! Convert to radians
70      MY.ANGLE.MIN = FP(MY.ANGLE)*60 ! Determine minutes
80      MY.ANGLE.SEC = FP(MY.ANGLE.MIN)*60 ! and seconds
90
100     MY.ANGLE$ = STR$(IP(MY.ANGLE))&CHR$(248)&
        STR$(IP(MY.ANGLE.MIN))&"' "&
        STR$(ROUND(MY.ANGLE.SEC,0))&" " " "
110
120     PRINT USING "The sine of 'e (##.#### rad ) is
        #.####",MY.ANGLE$,MY.ANGLE.RAD,SIN(MY.ANGLE)
```

**Enter angle in degrees: 42.624**

**The sine of 42°37'26" ( 0.7439 rad ) is 0.67718**

# KILL Statement

KILL terminates the execution of a subtask.

KILL task-id

Operation	The subtask number <i>task-id</i> , and all tasks subordinate to that subtask, are terminated. Memory used by these subtasks is freed.
Notes	<p>An implicit KILL of all subtasks is performed when a <a href="#">CSI</a>, <a href="#">END</a>, <a href="#">QUIT</a> or <a href="#">SYSTEM</a> statement is performed.</p> <p>Any windows opened by the subtask are closed, and, since deactivating a task deactivates its subtasks, windows opened by subordinate tasks are also closed.</p>
Restriction	The <i>task-id</i> must be the value returned from an <a href="#">ACTIVATE</a> function call.
See also	<a href="#">ACTIVATE</a>

## Examples

<i>This simple program opens a window and accepts a text string. Until the first character of the string is typed, a time of day clock is displayed on the screen.</i>	10	OPTION PROMPT "", CASE "M"
	20	
	30	PRINT CLS\$;
	40	WINDOW OPEN 1,1,2,80,23; COLOR 7,1; SELECT
	50	
	60	ONE.ONLY% = SEMAPHORE("ONE-ONLY")
	70	SUB.TASK% = ACTIVATE("TIME\$DAY") ! Start subtask
	80	
<i>The time of day clock is maintained by a subtask.</i>	90	PRINT AT\$(1,5);"Enter any text: ";
	100	WAIT #0 ! Both tasks operate until char typed
	110	WHILE SET(ONE.ONLY%) \ SLEEP 0.1 \ WEND
<i>When the input is accepted the subtask is terminated with the KILL statement.</i>	120	INPUT ANY.TEXT\$
	130	
	140	KILL SUB.TASK%
	~	

Statements

*The ONE.ONLY semaphore is used to prevent both tasks from attempting to display data on the screen at the same time.*

TIME\$DAY program:

```
10     TIMER% = SEMAPHORE("MINUTE") ! Catalogue names
20     ONE.ONLY% = SEMAPHORE("ONE-ONLY")
30     TIMER TIMER% SYNC 1      ! Set timer every second
40
50 WAIT.TIMER:
60
70     WAIT EVENT(TIMER%) ! Wait for second timer
80     WHILE SET(ONE.ONLY%) ! Display is one only
81         SLEEP 0.1
82     WEND
90     WINDOW STATUS PRIOR.WIN% ! Get current window
100    WINDOW SELECT 0,UPDATE OFF ! Change to main window
110    PRINT AT$(72,1);TIME$(0);
120    WINDOW SELECT PRIOR.WIN% ! Reselect prior window
130    X = RESET(ONE.ONLY%) ! Finished with screen
140    TIMER TIMER% SYNC 1 ! Set a new timer
150    GOTO WAIT.TIMER
160    END
```

# LBOUND, UBOUND Functions

The LBOUND and UBOUND functions return the lower and upper boundaries of indexes for an array.

**LBOUND** ( *array-name*, *dimension* )

**UBOUND** ( *array-name*, *dimension* )

*array-name*

» Array name

*dimension*

» Number of array dimension (1 or 2)

Operation	<b>LBOUND</b>	The lower limit for the array dimension is returned. This will be a zero or one depending upon the OPTION BASE that is in effect for this program.
	<b>UBOUND</b>	The upper limit for the array dimension is returned. This value is the maximum valid index that can be used for that dimension of <i>array-name</i> .

**Notes** Although the LBOUND for an array will not change during the execution of a program, it is possible that the UBOUND of an array may change if the REDIM statement is used.

**See also** COMMON, DIM, LOCAL, OPTION BASE, PUBLIC, REDIM, SHARED, STATIC

**Examples**

```
1000 OPTION BASE 1
1010 DIM A$(10,20)
...
1800 REDIM A$(ROWS%,COLS%)
...
2000 FOR I% = LBOUND(A$,1) TO UBOUND(A$,1)
2010     FOR J% = LBOUND(A$,2) TO UBOUND(A$,2)
2020         PRINT I%;J%;" = ";A$(I%,J%)
2030     NEXT
2040 NEXT
```

Statements

---

# LCASE\$ Function

LCASE\$ returns a string with all alphabetic characters in lowercase.

LCASE\$( *string-expression* )

**Operation**      *string-expression* is evaluated and all alphabetic characters are replaced with their lowercase equivalent. Non-alphabetic characters are not changed. The resulting string is returned.

**See also**      [UCASE\\$](#)

<b>Examples</b>	10	INPUT "Please enter your first name: ",FIRST.NAME\$
	20	
	30	FIRST.NAME\$=UCASE\$(FIRST.NAME\$[1:1])& LCASE\$(RIGHT\$(FIRST.NAME\$,2))
	40	
	50	PRINT "Hello ";FIRST.NAME\$

*In this example, the UCASE\$ and the LCASE\$ functions are used to capitalize the operator's first name.*

# LEFT\$ Function

LEFT\$ returns the leftmost portion of a string.

LEFT\$( *string-expression*, *count* )

**Operation** Starting with the first character, *count* number of characters of *string-expression* are returned.

**Notes** The substring operator [1:*count*] can produce the same results. Refer to the section on “[String Expressions](#)” on page 42 in Chapter 2 “[Fundamentals](#)”.

**See also** [MID\\$](#), [RIGHT\\$](#)

## Examples

In this code section, the LEFT\$ and RIGHT\$ functions are used to replace a section of a string.

900300 LOC% = SCH(1,ERROR.MSGS\$(ERR.NBR%),"%f")

900310 IF LOC%

900320 IF ERR.NBR%=40

900330 ERROR.MSGS\$(ERR.NBR%) = LEFT\$(ERROR.MSGS\$(ERR.NBR%),LOC%-1)&TO.PROG\$&RIGHT\$(ERROR.MSGS\$(ERR.NBR%),LOC%+2)

900340 ELSE ERROR.MSGS\$(ERR.NBR%) = LEFT\$(ERROR.MSGS\$(ERR.NBR%),LOC%-1)&FILENAME\$&RIGHT\$(ERROR.MSGS\$(ERR.NBR%),LOC%+2)

900350 IFEND

900360 IFEND

~

Statements

# LEN Function

LEN returns the number of characters in a string expression.

LEN( *string-expression* )

- Operation**
- The total number of characters in *string-expression* is computed and returned.
- Notes**
- All characters in *string-expression* are counted, including nulls, spaces, control characters, *etc.*

## Examples

*This is a simple ZIP code number verification and formatting routine.*

*A user-defined function is used to strip all of the non-digit characters that may be in the field.*

*Then, based upon the length of the number, the ZIP code number is formatted or rejected.*

*This routine is similar to the above routine, except that it is validating a telephone number based upon the number of digits in the number.*

```
21000  VALIDATE.ZIP:
21010
21020      FIELD$ = FN.DIGIT.STRIP$(FIELD$) ! Remove nondigit
21030
21040      SELECT LEN(FIELD$)
21050          CASE 0              ! Null string
21060          CASE 5              ! 5-digit zip
21070          CASE 9              ! ZIP+4 zip
21080              FIELD$[0:5] = "-" ! insert dash
21090          OTHERWISE
21100              VALID% = FALSE% ! Error!
21110              P2MSG$ = "ZIP CODES HAVE 5 OR 9 DIGITS"
21120          CEND
21130
21140      RETURN
~
22000  VALIDATE.PHONE:
22010
22020      FIELD$ = FN.DIGIT.STRIP$(FIELD$) ! Remove nondigit
22030
22040      SELECT
22050          CASE LEN(FIELD$)=0 ! Null string
22060          CASE LEN(FIELD$)=7 ! Normal phone number
22070              FIELD$[0:3] = "-"
22080          CASE LEN(FIELD$)=10 ! Area code + phone number
22090              FIELD$ = "("&FIELD$[1:3]&") "&FIELD$[4:6]&
                  "-"&FIELD$[7:10]
```



<i>Only standard num-</i>	22100	OTHERWISE
<i>bers are accepted</i>	22110	VALID% = FALSE%
<i>by this routine. It</i>	22120	P2MSG\$ = "PHONE NUMBERS HAVE 7 OR 10 DIGITS"
<i>would not be diffi-</i>	22130	CEND
<i>cult to enhance it to</i>	22140	
<i>allow phone num-</i>	22150	RETURN
<i>bers with exten-</i>		
<i>sions.</i>		

---

## LET Statement

LET assigns a value to a variable, performs substring modification, assigns the value to a multiple-line, user-defined function, or invokes error-trapping in a program.

```
[LET] variable = expression

[LET] string-variable[ from : through ] = string-expression

LET ERR = integer-constant

[LET] FNname = expression
```

**Operation** Except for [Mode 3](#), the keyword LET is not required, and is not normally used..

**Mode 1**—The standard assignment statement. The value of *expression* is assigned to *variable*. The *variable* may be a string, integer, or numeric variable name. It may also be a simple variable or an array element reference. For example:

```
A$ = "INITIAL VALUE"
INDEX% = 25
PAY.RATE = 15.50
X$ = SYS.ENV$(17, "SYSTEM=" & SYSNAME$)
FALSE% = 0 \ TRUE% = NOT FALSE%
CUR.TIME = SECOND(TIME$(0))
ATTRIB$(15) = EXT$(TERM$(5), 2, 0)
MAIN$(8, 2) = "*"
```

**Mode 2**—A special instance of substring reference that allows the *string-variable* to be modified by insertion, deletion or replacement of a portion of the string. See "[Substring Assignment](#):" on page [299](#).

**Mode 3**—A special instance of assignment. This form assigns a value to the [ERR](#) function, causing a jump to the error handling routine defined by an [ON ERROR](#) statement. Note that the LET keyword is required in this mode. Generally, this would be used to test a program's error handling routine. Only trappable error code values may be used for the *integer-constant*. It may also be used for user-defined error conditions. See example, line 7160.

**Mode 4**—This mode of the LET statement assigns a value to a multiple line function definition. Refer to the [DEF FN](#) statement for a discussion of its usage.

## Notes

In general, the value of the expression to the right of the equal sign is assigned to the variable identified on the left of the equal sign. Any prior value of the variable is completely replaced with this new value.

The current value of the variable is not replaced until after the expression is evaluated. Therefore, the expression may contain references to the variable. For example:

```
A$ = "This is"  
A$ = A$&" just a test"
```

Here, A\$ is first assigned an initial value. Then, a string literal is concatenated onto the current value of A\$. The final value of A\$ will be: "This is just a test".

## Substring Assignment:

Substring operators are discussed in a prior chapter on [“Expressions”](#). Normally, the substring operator identifies a portion of a string that is used in an expression. In [Mode 2](#) of the LET statement, the substring operator is used with the variable name to the left of the equal sign operator. In this special instance, the substring operator identifies the portion of the current value of the variable that is modified by the assignment.

There are three forms of modification:

**Substring overlay:** When *from* is less than or equal to *through*, substring overlay is performed. For example:

```
A$ = "ABCDEFGH IJKLMNOPQRSTUVWXYZ"  
A$[4:8] = "1234567890"
```

In this form, the character position range identified with [*from* : *through*] is overlaid with the value of *string-expression*. The *string-expression* value is either padded on the right or truncated to the same length as [*from* : *through*]. In the above example, A\$ is changed to:

```
"ABC12345 IJKLMNOPQRSTUVWXYZ"
```

The substring range is five characters in length; the overlay string value is truncated to five characters and overlays character positions 4 through 8.

```
A$ = "ABCDEFGH IJKLMNOPQRSTUVWXYZ"  
A$[6:10] = " "
```

In this example, the substring range is five characters in length also; the overlay string is padded to five characters and overlays character positions 6 through 10, producing:

```
"ABCDE          KLMNOPQRSTUVWXYZ"
```

**Substring replacement:** When *from* is greater than *through*, substring replacement is performed. For example:

```
A$ = "ABCDEFGH IJKLMNOPQRSTUVWXYZ"
A$[8:4] = "1234567890"
```

In this form, the character position range identified with [*through*+1 : *from*] is replaced with the entire value of *string-expression*. In the above example, *a\$* is changed to:

```
"ABCD1234567890 IJKLMNOPQRSTUVWXYZ"
```

The five characters of the substring range are replaced with the ten characters of the replacement string.

```
A$ = "ABCDEFGH IJKLMNOPQRSTUVWXYZ"
A$[10:6] = ""
```

In this example, the substring range is five characters in length also; the replacement string is a null string causing the five characters to be replaced with nothing, or deleted, producing:

```
"ABCDEFKLMNOPQRSTUVWXYZ"
```

**Substring insertion:** When *from* is 0, substring insertion is performed after the *through* position. For example:

```
A$ = "ABCDEFGH IJKLMNOPQRSTUVWXYZ"
A$[0:8] = "1234567890"
```

Here, the *string-expression* is inserted after position 8, producing:

```
"ABCDEFGH1234567890 IJKLMNOPQRSTUVWXYZ"
```

No deletion or replacement is performed on the variable contents; no padding or truncation is performed on the *string-expression* value.

**Restrictions**      The type of *variable* (string or numeric) must match the type of *expression*.

**See also**            [CONSTANT](#), [DEF FN](#), [IOLIST](#), [MAT](#), [ON ERROR](#)

## Examples

<i>A function definition that accepts a date from the operator in one of two styles.</i>	790000	DEF FN.GET.DATES\$(X%,Y%,STYLE\$,DATE.FLD\$,ATTRIBUTE\$,HELP.FILES)
	790010	
	790020	! Accept a date from the operator
	790030	
<i>If style is month and year, then insert day 01 into string.</i>	790040	! Styles accepts:
	790050	
	790060	! MDY = MM/DD/YYYY
	790070	! MY = MM/YYYY
	790080	
<i>Validate and normalize the date.</i>	790090	IF UCASE\$(STYLE\$)="MY" ! form is mm/yyyy
	790100	DATE.FLD\$[0:3] = "01/" ! Change to mm/01/yyyy
	790110	IFEND
	790120	DATE.FLD\$ = DT\$(DATE.FLD\$) ! Normalize the date
<i>Add century to year by inserting "19" before year number.</i>	790130	IF LEN(DATE.FLD\$)=8 ! Form is mm/dd/yy
	790140	DATE.FLD\$[9:6] = "19" ! Change to mm/dd/19yy
	790150	IFEND
<i>Pick out the various components of date.</i>	790160	IN.DATE%(1) = VAL(DATE.FLD\$[1:2]) ! Get month value
	790170	IN.DATE%(2) = VAL(DATE.FLD\$[4:5]) ! Get day value
	790180	IN.DATE%(3) = VAL(DATE.FLD\$[7:10]) ! Get year value
	790190	
<i>Using another function, reformat validated, complete date into the style desired.</i>	790200	INDATES\$ = FN.FORMAT.INPUT.DATES\$(STYLE\$,DATE.FLD\$)
	~	
	791350	
<i>At end of function operation, assign a value to the function and exit.</i>	791360	FN.GET.DATES\$ = DATE.FLD\$
	791370	
	791380	FNEND
	7100	GET.MAIN:
	7110	
<i>Read a record from a file. The record is a required record. If it is not found it is a fatal error...the program cannot continue operation.</i>	7120	MAT READ #1,"MAIN":ITEMS\$
	7130	
	7140	IF EOF(1)
	7150	PROGRAM.MSG\$ = "'MAIN' ENTRY MISSING IN MENU FILE"
<i>Set the error code to a special value used by the error trapping routines to indicate that a program defined error occurred.</i>	7160	LET ERR = 44 ! Invoke standard error trapping
	7170	ELSE MAIN.SELECT\$ = ""
	7180	IFEND
	~	

<i>The <b>LET ERR</b></i>	980000	ERROR.TRAP:
<i>statement causes</i>	980010	
<i>the <b>ERROR.TRAP</b></i>	980020	IF ERR=1 THEN RESUME
<i>routine to be</i>	980030	
<i>invoked, which, in</i>	980040	! Assign error number and location to common variables
<i>turn, causes the</i>	980050	
<i><b>ERRORTRP</b> pro-</i>	980060	<b>ERR.NBR% = ERR \ ERR.LINE = ERL</b>
<i>gram to be invoked.</i>	980070	
	980080	! Goto standard error logging and reporting program
	980090	
	980100	LINK "ERRORTRP"
	980110	

---

## LINE Function

LINE returns a value one less than the attached line length of an I/O channel.

LINE( *channel* )

**Operation** The attached line length minus one, of the device opened on channel *channel* is returned.

When *channel* is for the console and a window is open, the line length returned depends upon the status of the CLIP attribute of the currently active window. When CLIP ON is in effect, the line length returned is one less than the line length of the full screen; when CLIP OFF is in effect, the line length returned is one less than the line length of the window.

**Notes** A *channel* of zero always refers to the standard output device (stdout), which is normally the console..

This function is only usable for consoles and printers. If the *channel* refers to a disk, tape file, or a communications device, zero is always returned.

**I/O Redirection** When *channel* is 0 this statement actually refers to the current standard output device (stdout). Normally this is the console display. However, stdout may have been redirected when the program was invoked:

```
>myprog > text.file
```

When stdout has been redirected and *channel* is 0, the LINE function returns the attached line length of the stdout device if it is a non-spoiled printer. When stdout is not the console display or a printer, it returns a zero.

A program can determine if the stdout device has been redirected to a file or device other than the console by using the [SYS.ENV\\$](#) function:

```
IF SYS.ENV$(34,"STDOUT") = "Y"
```

The above test is true when stdout has been redirected.

**Restrictions** The *channel* must be 0 or refer to an open file channel.

**See also** [PAGE](#)

Examples

*The text "HEAD-  
ING" is displayed  
on the top line of  
the screen, cen-  
tered. The USING  
mask is 'ccccc with  
the 'c' repeated for  
the full screen  
width.*

1010	PRINT AT\$(1,1);
1020	PRINT USING " "&RPT\$(LINE(0),"c"),"HEADING";

*The WINDOW  
OPEN sets the  
CLIP attribute to  
OFF. The text  
"HEADING" is dis-  
played on the top  
line of the window,  
centered.*

100010	WINDOW OPEN 5,10,4,40,3;SELECT; CLIP OFF
~	
100230	PRINT AT\$(1,1);
100240	PRINT USING " "&RPT\$(LINE(0),"c");"HEADING";
~	

Statements



---

## LINK Statement

LINK loads another program and continues execution in that program without closing any files.

**LINK** *program-name-expression*

**Operation**      The program designated by *program-name-expression* is loaded and execution resumes with the first statement in that program.

**Notes**            When executed in the interpreter, the linked-to program may have a file-type of BASIC or BAS.

**See also**         [CHAIN](#), [COMMON](#), [CSI](#), [RUN](#)

---

## LINPUT Statement

LINPUT, or line input, accepts one string value from the console or an input file.

- 1 **LINPUT** *string-variable*
  - 2 **LINPUT** *prompt-text, string-variable*
  - 3 **LINPUT** *#channel: string-variable*
  - 4 **LINPUT** *#channel, key: string-variable*

### Operation

**Mode 1**—Accept a string from the console using the current prompt string. The prompt for the input is the default “?” unless a different prompt string is specified by an **OPTION PROMPT** statement. Characters accepted are echoed to the console.

**Mode 2**—Accept a line of data from the console. The prompt for the input is the *prompt-text* followed by the default “?” or, *prompt-text* followed by the prompt string specified by an **OPTION PROMPT** statement. Characters accepted are echoed to the console.

**Mode 3**—Accept one line of data from the sequential access file *channel*. No prompt is output to the file. The *channel* may refer to a disk file, tape file, a device such as “CON” or “COM”, or it may be 0, indicating the console. Characters are not echoed to the file unless it is the console.







**Mode 4**—Accept one record of data from the direct, keyed, or indexed access file *channel*. There is no prompt output to the file and the characters accepted are not echoed to the file or the console.

### Notes

For **Mode 1** and **Mode 2**, and for **Mode 3** when the file is “CON,” the input may come from EXEC language &STACK statements. If this program is invoked by an EXEC program that placed data on the EXEC stack, that data will be read by this statement. After all of the data on the stack is read, data will come from the console.

The primary difference between the LINPUT statement and the **INPUT** statement is that LINPUT only accepts one string field and all characters read are assigned to the input variable. That is, any leading or trailing spaces are not removed, no quotation marks are removed, and embedded commas do not cause field termination.

The general operation of this statement is:

1. Display the *prompt-text* (Mode 2 only).
2. Display the prompt string (Mode 1 and Mode 2 only).
3. Locate and read the specified record (Mode 4 only).
4. Accept one line or record of ASCII text, saving the data in *string-variable*. For modes other than Mode 4, a record or line is a string of data terminated by the carriage return character.
5. If the input is from the console the current status of the OPTION CASE statement controls the case mode of the characters input and echoed to the console.
6. If the input is from the console, the editing key  is allowed to make corrections to the input prior to pressing . A + may be entered to erase all of the current input and to move the cursor back to the first input character position. A blank line may be terminated with any control character other than . The INP function is set to the value of the control character used.
7. For console input, termination of field entry (either by pressing  or using a control character in the first position) causes a CR,LF to be echoed to the screen.

Attempting to input a file record (Mode 3 and Mode 4) when there are no more records in the file, returns a null string and the EOF indicator is set.

When no record exists matching the *key* (Mode 4 only), the EOF indicator is set and the variables in the *variable-list* are set to nulls.

If an ON EVENT, ON KEY, ON MOUSE or ON TIMEOUT is in effect, a LINPUT statement may be terminated without entering a carriage return.



When the input is from the console and an ON TIMEOUT statement is in effect and the operator does not enter any characters or editing keys before the time-out period elapses, the LINPUT statement is terminated with the data already entered by the operator assigned to the *string-variable*. The time-out event handler is then invoked. As each character or editing key is entered the time-out timer is reset.

**I/O Redirection** [Mode 1](#) and [Mode 2](#) of this statement and [Mode 3](#) and [Mode 4](#) when *channel* is 0 actually get the input from the current standard input device (stdin). Normally this is the console keyboard. However, stdin may have been redirected when the program was invoked:

```
>myprog < text.file
```

When stdin has been redirected the text accepted by this statement comes from that file, device or pipe. The prompt character or *prompt-text* is output to the stdout device which may have been redirected. When stdout is redirected, the text accepted is not echoed to the standard output device.

A program can determine if the stdin device has been redirected to a file or device other than the console keyboard by using the [SYS.ENV\\$](#) function:

```
IF SYS.ENV$(34,"STDIN") = "Y"
```

The above test is true when stdin has been redirected.

**Restrictions** The *prompt-text* may only be specified with a *string literal expression*, that is, a string expression that starts with a string literal. For example:

```
LINPUT "Label",A$
LINPUT "Item "&STR$(I%),A$
LINPUT ""&PROMPT.STRING$,A$
```

The *channel* must refer to a file opened with INPUT or UPDATE. Refer to the [OPEN](#) statement for details.

When inputting from a direct access file, the *key* must be a numeric expression; inputting from a keyed or indexed access file, the *key* must be a string expression.

Data read from a disk file should not be data that was written with the [WRITE](#) or [MAT WRITE](#) statements. Use the [READ](#), [READNEXT](#), [READPREV](#), [MAT READ](#), [MAT READNEXT](#), or [MAT READPREV](#) statements to retrieve records of this type.

**See also** [INPUT](#), [LINPUT USING](#), [MAT INPUT](#), [OPEN](#), [OPTION PROMPT](#)

## Examples

*A routine to display help text. The first record of the help text file contains two numbers so it is read with the **INPUT** statement.*

```
800000  GENERIC.HELP.KEY:
800010
800020      EVENT.FLAG% = TRUE%
800030
800040      WINDOW STATUS PRIOR.TO.G.HELP%
800050
800060      OPEN #38:HELP.LIB$&GENERAL.HELP$, INPUT SEQUENTIAL
800070      INPUT #38: LINES%,WIDTH%
800080
800090      DEPTH% = MIN(LINES%,21)          ! Max depth is 21 lines
800100
~
```

*Subsequent records in the help text file may contain commas, leading spaces, etc., so they are read with the **LINPUT** statement to ensure that all characters of each line are accepted.*

```
800260  WINDOW SELECT 5
800270
800280  LINPUT #38: HELP.TEXT$
800290
800300  L% = 0          ! Number of lines displayed in window
800310
800320  WHILE NOT EOF(38)          ! Until end of help text
800330      IF L%>=DEPTH% THEN GOSUB RELEASE.HELP.PAGE
800340      L% = L%+1              ! Count lines displayed in window
800350      PRINT AT$(2,L%);HELP.TEXT$;
800360      LINPUT #38: HELP.TEXT$    ! Get next help text
800370      WEND
~
```

---

## LINPUT USING Statement

LINPUT USING accepts one string value from the console with length control and editing capabilities..

- 1 **LINPUT USING** *input-mask, string-variable* [, *start-pos*]

2 **LINPUT** *prompt* **USING** *input-mask, string-variable* [, *start-pos*]

---

*input-mask*           »   *string-expression*  
                              *string-literal-exp*

*string-literal-exp* »   *string-literal* [&*string-expression*]

### Operation

**Mode 1**—Display the current prompt string. The prompt for the input is the default “?” unless a different prompt string is specified by an [OPTION PROMPT](#) statement. as defined by the [OPTION PROMPT](#) statement. Preload the *string-variable* with the *input-mask*, display the *string-variable*, and then accept a string from the operator.

**Mode 2**—Identical to [Mode 1](#) except that the *prompt-text* is displayed followed by the current prompt string, followed by the preloaded *string-variable*.

### Notes

The primary differences between the [LINPUT](#) statement and the LINPUT USING statement are that LINPUT USING:

- ▶ Limits the length of the input field
- ▶ Can “preload” the input string allowing changes to be made
- ▶ Initial starting position of input can be specified
- ▶ Input is terminated by any control character
- ▶ No CR, LF is displayed upon termination





The input may come from EXEC language &STACK statements. If this program is invoked by an EXEC program that placed data on the EXEC stack, that data will be read by this statement. After all data on the stack is read, data comes from the console keyboard.

The general operation of this statement is:

1. Display the *prompt-text* (Mode 2 only).
2. Display the prompt string (OPTION PROMPT).
3. The length of *input-mask* determines the maximum length of the input field.
4. If *input-mask* is a *string literal expression*, clear the *string-variable*. (A string literal expression is a string expression whose first element is a string literal. For example: "&field\$, !"&field\$.) When the first character of *input-mask* is an exclamation character, automatic entry is enabled. In this mode, a carriage return is not needed. Entering the last character of the field causes automatic input termination.
5. When *input-mask* is not a *string literal expression*, initialize *string-variable* to the value of the *input-mask*, display the *string-variable*, and position the cursor to the first character position or to *start-pos* if specified. If *start-pos* is greater than the length of *input-mask*, position to last character in *input-mask*.
6. Accept one line of ASCII text, saving the data in *string-variable*. The maximum length of entry is restricted to the length of *input-mask*. After the maximum number of characters have been entered no more characters are accepted and the bell is rung every time a non-editing key is pressed. (Automatic entry causes the field to be accepted when the maximum number of characters are entered.)

Although *start-pos* can be specified with any LINPUT USING statement, it is only effective when the *input-mask* is a string expression, not a string literal expression.

During entry, certain editing keys may be used to change the contents of *string-variable*.

Key	Meaning
	Move left one position.
	Move right one position.
	Delete the character to the left of the cursor.
	Delete the character under the cursor.

Except for the editing keys listed above, any control key can be used to terminate field input. The INP function is assigned the value of the control key used, except for carriage returns entered after character position 1, which assigns a zero to the INP function. (Automatic entry also assigns a 0 to the INP function.)

Upon input termination, no editing is done on the *string-variable*, specifically, trailing spaces are not removed.



If an **ON EVENT**, **ON KEY**, **ON MOUSE** or **ON TIMEOUT** is in effect, a **LINPUT USING** statement may be terminated early without entering a carriage return or the number of characters requested.

When the input is from the console and an **ON TIMEOUT** statement is in effect and the operator does not enter any characters or editing keys before the time-out period elapses, the **LINPUT USING** statement is terminated with the data already entered by the operator assigned to the *string-variable*. The time-out event handler is then invoked. As each character or editing key is entered the time-out timer is reset.

**Restrictions** This statement cannot be used when i/o redirection is used. Because it is an editing-type operation, attempting to use this statement when stdin or stdout have been redirected cause error 56 to occur.

**See also** **INP**, **INPUT**, **LINPUT**, **MAT INPUT**, **OPTION PROMPT**

### Examples

*Using today's date as the initial string, accept a date from the operator.*

```
10000      LINPUT USING DATE(0),ENTRY.DATES$,4
```

*The starting input position is character 4, which is the day number of the date.*

*General purpose field input function.*

```
700020  DEF FN.INPUT$(INFIELD$,X%,Y%,L%,CASE.MODE$,PROMPT$,P2$,HELP$,
        SPECIAL$,DUPFIELD$)
700030
700040      SPECIFIC.HELP$ = UCASE$(HELP$)
700050      HELP.X% = X% \ HELP.Y% = Y%+2 ! Define help display position
700060
700070      OPTION CASE CASE.MODE$
700080      PRINT FN.PROMPT$(PROMPT$,P2$);
700090
```

*Set the input case mode and display any prompt messages associated with input field.*

*Use window to accept input.*

```
700100      WINDOW OPEN 4,X%,Y%+1,L%,1;FRAME;COLOR BLACK%,WHITE%; SELECT
700110
700120  GET.INFIELD:
700130
```



<i>Position for field input.</i>	700140	PRINT AT\$(1,1);
<i>Pad the current value of the input field to the maximum length allowed for the field. This is done to make sure that the input mask is the required width.</i>	700150 700160 700170 700180 700190 700200	<b>LINPUT USING RPAD\$(INFIELD\$,L%),INFIELD\$</b> IF EVENT.FLAG% ! Terminated by an ON KEY event? EVENT.FLAG% = 0 GOTO GET.INFIELD IFEND
<i>Test for certain control key terminations.</i>	700210 700220 700230 700240 700250 700260 700270 700280 700290	SELECT INP           ! Test termination control CASE 24           ! Ctrl+X = field clear INFIELD\$ = "" GOTO GET.INFIELD CASE 1           ! Ctrl+A = duplicate field INFIELD\$ = DUPFIELD\$ GOTO GET.INFIELD CEND
<i>The field input might have excess spaces left on the end because of the input mask. Remove them and save the field as the value of the function.</i>	700300 700310 700320 700330 700340	FN.INPUT\$ = TRIM\$(INFIELD\$)  IF PROMPT\$ THEN PRINT FN.PROMPT\$(",") ! Clear prompts  WINDOW SELECT 1, UPDATE OFF   ! Select main window
<i>Display the input value in the main window and remove the input window.</i>	700350 700360 700370 700380 700390	PRINT AT\$(X%,Y%);RPAD\$(INFIELD\$,L%); WINDOW CLOSE 4           ! Remove input window WINDOW SELECT 1, UPDATE ON   ! Refresh main window  FNEND

# LOCAL Statement

The LOCAL statement declares that certain variables or arrays are local in scope.

<b>LOCAL</b> <i>variable-list</i>	
<hr/>	
<i>variable-list</i>	» <i>variable-name</i> [, <i>variable-list</i> ] <i>array</i> [, <i>variable-list</i> ]
<i>array</i>	» <i>array-name</i> ( <i>subscript</i> ) <i>array-name</i> ( <i>subscript</i> <sub>1</sub> , <i>subscript</i> <sub>2</sub> )

**Operation**      The variables declared in the *variable-list* are marked as *local in scope* and *temporary in duration*. New storage is allocated for the variables and arrays and the storage is initialized to nulls (empty strings or zero valued numbers).

Within the subprogram or function, references to variable or array names that are specified in the *variable-list* refer to these local variables, not to variables of the same name in the main program storage area.

Upon exit from the subprogram or function the storage for the local variables is discarded.

**Notes**      The purpose of the LOCAL statement is to allow a subprogram or user-defined function to use variable and array names that are local in scope. A local variable name might be the same as one used in the main program or other program segments without affecting or using the value of that variable or array in the other program segments.

LOCAL variables and arrays are initialized to null strings or zeros each time that the subprogram or user-defined function is invoked. Upon exit from the function or subprogram the variables and arrays are cleared from memory.

STATIC variables and arrays are similar to LOCAL variables and arrays except that the storage is *static in duration*. It is not re-initialized each time the subprogram or function is invoked and the storage is not discarded upon exit.

**Dimensioning:** Arrays dimensioned with this statement may be one-dimensional or two-dimensional.

One-dimensional array: 

0	1	2	3	4
---	---	---	---	---

Two-dimensional array: 

0,0	0,1	0,2	0,3	0,4
1,0	1,1	1,2	1,3	1,4
2,0	2,1	2,2	2,3	2,4

The number of elements allocated for an array depends upon the **OPTION BASE** statement. When **OPTION BASE 0** is in effect (default), one is added to each of the dimension sizes specified to account for the zero indices.

For example:

```
OPTION BASE 0
LOCAL A(5)           ! 6 elements allocated
LOCAL B(12)          ! 13 elements allocated
LOCAL C(2,4)         ! 15 elements allocated

OPTION BASE 1
LOCAL A(5)           ! 5 elements allocated
LOCAL B(12)          ! 12 elements allocated
LOCAL C(2,4)         ! 8 elements allocated
```

## Defaults

In a subprogram definition, all variables and arrays are **LOCAL** by default. The **SHARED** statement must be used to declare that a variable or array is *global in scope* (shared with the main program or other program segments).

In a **user-defined function** definition, all variables and arrays are **SHARED** by default. The **LOCAL** statement must be used to declare that a variable or array is not shared with the main program.

## Restrictions

The **LOCAL** statement must be specified before any executable statements in the subprogram or user-defined function.

The **LOCAL** statement must appear on a line by itself (it may only be followed by a **REM** statement on the same line).

The **LOCAL** statement is only effective when used in a user-defined function definition. When used in the main program or in a subprogram, the **LOCAL** statement has no effect on variables and is the same as a **DIM** statement for the arrays specified in the *variable-list*. (Variables in the main program segment cannot be local in scope and variables in a subprogram are **LOCAL** by default.)

Examples

This function removes non-numeric characters from a string.

The `LOCAL` statement declares that all of the work variables used in the function are local to the function and do not affect any variables used by the main program.

704120

704130

704140

704150

704160

704170

704180

704190

704200

704210

704220

704230

704240

704250

DEF FN.DIGIT.STRIP\$(STRING\$)

**LOCAL RETURN.STRING\$, INDEX%, CHAR\$**

FOR INDEX% = 1 TO LEN(STRING\$)

CHAR\$ = STRING\$[INDEX%:INDEX%]

IF CHAR\$>="0" AND CHAR\$<="9"

RETURN.STRING\$ = RETURN.STRING\$&CHAR\$

IFEND

NEXT

LET FN.DIGIT.STRIP\$ = RETURN.STRING\$

FNEND

Statements

316

LOCAL

---

# LOCKED.BY Function

The LOCKED.BY function returns the partition number of the task locking the record or file that your program tried to access.

LOCKED.BY

Operation	The partition number of the task locking the record or file is returned.
Notes	This function returns the partition number. The account name of that user can be determined with the GET.SCR.ENTRY ToolKit function.
Restrictions	This function is only effective when executed as part of an <a href="#">ON ERROR</a> trap routine and when the routine was invoked by an error 48 or 49 (record or file locked error).
See also	<a href="#">ON ERROR</a>

Examples

```
OPTION LOCK 2           ! Wait 2 seconds for lock detection

ON ERROR GOTO ERROR.TRAP

OPEN #1: "SOME.FILE", UPDATE INDEXED
READ #1,"REC 1":RECORD$
CLOSE #1

...

ERROR.TRAP:

  IF ERR=48               ! Record locked error
    LOCKING.PID% = LOCKED.BY
    PRINT USING "Record is locked by PID 999.",LOCKING.PID%
  IFEND

RESUME
```

Statements

# LOG, LOG2, LOG10 Functions

LOG returns the natural logarithm of a value.

LOG2 returns the binary logarithm of a value (logarithm base 2).

LOG10 returns the common logarithm of a value (logarithm base 10).

**LOG**( *numeric-expression* )

**LOG2**( *numeric-expression* )

**LOG10**( *numeric-expression* )

**Operation** LOG computes and returns the value of  $\log_e \text{numeric-expression}$ .

LOG2 computes and returns the value of  $\log_2 \text{numeric-expression}$ .

LOG10 computes and returns the value of  $\log_{10} \text{numeric-expression}$ .

**Notes** The value of  $e$  is 2.718281828459.

Base- $n$  logs can be calculated for any value by dividing the natural logarithm of the value by the natural logarithm of the desired base.

$$\log_n x = (\log_e x) \div (\log_e n) = (\log(X)) / (\log(N))$$

**Restrictions** Negative values for *numeric-expression* cause these functions to return a zero.

**See also** [EXP](#)

**Examples**

1000	NATURAL.LOG = <b>LOG(VALUE)</b>
1010	BINARY.LOG = <b>LOG2(VALUE)</b>
1020	COMMON.LOG = <b>LOG10(VALUE)</b>

---

# LPAD\$ Function

LPAD\$ returns a fixed length string, space padded on the left.

LPAD\$( *string-expression*, *length* )

**Operation**            *string-expression* is expanded to a length of *length* by adding spaces to the beginning of the string. When the length of *string-expression* is greater than or equal to *length*, the string is unchanged. The result is returned.

**See also**            [RPAD\\$](#)

**Examples**            The following code might be used in a report generation program to display the report progress on the screen.

```
Display the status      55000 STATUS.DISPLAY:
headings.              55010
                        55020      PRINT AT$(40,12); "Name";
                        55030      PRINT AT$(40,14); "Page";
                        55040      PRINT AT$(40,16); "Line";
                        ~
Display the current     56000      PRINT AT$(45,12); LPAD$(CUST.NAME$,30);
customer name in       ~
the status area.
Right align the
name by using the
LPAD$ function.
```

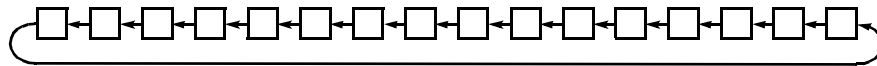
## LRL, LRR, LSL, LSR Functions

LRL returns the value of an integer after the “bits” of the number are rotated left a number of positions; LRR returns the value after it is rotated right; LSL returns the value after the bits are shifted left and LSR after it is shifted right.

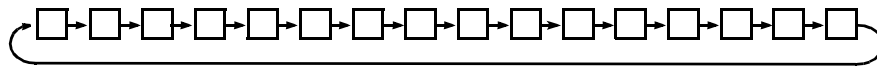
- 1 **LRL**( *numeric-expression*, *number-of-positions* )
- 2 **LRR**( *numeric-expression*, *number-of-positions* )
- 3 **LSL**( *numeric-expression*, *number-of-positions* )
- 4 **LSR**( *numeric-expression*, *number-of-positions* )

### Operation

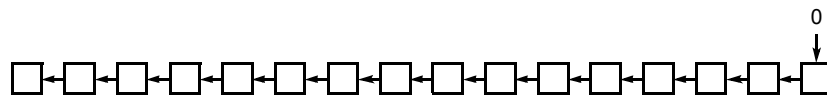
**Mode 1**—The value of *numeric-expression* is “integerized” and the 16 bits of that value are rotated left *number-of-positions*.



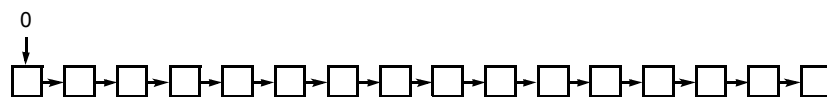
**Mode 2**—The value of *numeric-expression* is “integerized” and the 16 bits of that value are rotated right *number-of-positions*.



**Mode 3**—The value of *numeric-expression* is “integerized” and the 16 bits of that value are shifted left *number-of-positions*.



**Mode 4**—The value of *numeric-expression* is “integerized” and the 16 bits of that value are shifted right *number-of-positions*.





**Notes**

The LSL function effectively performs a multiply by two operation on the *numeric-expression*.

The LSR function effectively performs a divide by 2 operation on the *numeric-expression*.

**Examples**

In this example, the customer's monthly activity history has been saved in a bit-mapped integer, one bit per month.

This routine displays the customer activity for the prior 16 months, oldest month first.

First, the month headings are displayed. The low-order bit of the history value is tested with the binary AND operation.

The LRL function is used to shift the bits in preparation for the month test. Since it is a rotate operation, after sixteen iterations, the history value is back to its original value.

35 month names are defined to provide the 16 names that will be needed.

12000 DISPLAY.CUSTOMER.ACTIVITY:

12010

12020 DIM MONTHS\$(35)

12030 MAT READ MONTHS\$

12040

12050 CUR.MONTH% = VAL( DATE\$(0)[1:2] )

12060

12070 FOR MONTH% = CUR.MONTH%-16+24 TO CUR.MONTH%-1+24

12080     PRINT MONTHS\$(MONTH%); " ";

12090     NEXT

12100 PRINT

12110

12120 FOR MONTH% = 1 TO 16

12130     CUST.ACTIVE% = LRL (CUST.ACTIVE%,1) ! Shift to next mon

12140     IF CUST.ACTIVE% AND 1

12150         PRINT " X ";                             ! Show as active that month

12160     ELSE PRINT SPACE\$(4);                     ! No activity

12170     IFEND

12180     NEXT

12190

12200 RETURN

~

99000 DATA " ", " ", " ", " ", " ", " ", " ", " ", " ", "Sep", "Oct", "Nov", "Dec"

99010 DATA "Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug"

99020 DATA "Sep", "Oct", "Nov", "Dec", "Jan", "Feb", "Mar", "Apr"

99030 DATA "May", "Jun", "Jul", "Aug", "Sep", "Oct", "Nov"

Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec	Jan	Feb	Mar	Apr
X	X		X		X	X		X			X	X		X	X

LRL, LRR, LSL, LSR 321

The following code section is part of a month-end processing program. It uses one bit-mapped field to record the last 16 months activity for a customer, one bit per month.

<i>First, shift the bits left to make room for the new month.</i>	9500CUST.ACTIVITY% = <b>LSL</b> (CUST.ACTIVITY%,1) ! Shift
	9510
	9520IF CUSTOMER.ACTIVE%
	9530CUST.ACTIVITY% = CUST.ACTIVITY% OR 1 ! Mark
	9540IFEND
<i>If the customer was active this month then mark it by turning the bit on with the binary OR operation.</i>	9550
	~

This example picks off the interior attributes of a window's definition.

	1450WINDOW STATUS 10,WSTATS%
	1460
<i>The interior attributes are bit-mapped into a 16-bit integer (refer to the WINDOW STATUS statement).</i>	1470INTERIOR% = WSTATS%(17) ! attributes of window 10
	1480
	1490UL% = <b>LSR</b> (INTERIOR%,9) ! Underline attribute
<i>A binary AND is performed on the bit-mapped value to mask off the bits that are not part of the attribute in question.</i>	1500RV% = <b>LSR</b> (INTERIOR% AND 0100H,8) ! RV attribute
	1510BL% = <b>LSR</b> (INTERIOR% AND 0080H,7) ! Blink attribute
	1520BG% = <b>LSR</b> (INTERIOR% AND 0070H,4) ! Bg color
	1530FGINT% = <b>LSR</b> (INTERIOR% AND 0080H,3) ! Fg intensity
	1540FG% = INTERIOR% AND 0007H ! Fg color
	1550
<i>The LSR function is then used to shift the attribute bits to the low order position so that it can be assigned as a scalar value.</i>	~

---

## LTRIM\$ Function

LTRIM\$ returns a string with all leading spaces removed.

LTRIM\$( *string-expression* )

**Operation** All leading spaces of *string-expression* are removed and the result is returned.

**Notes** The functions LTRIM\$, TRIM\$, and RTRIM\$ all remove spaces from a string. They differ in where the extra spaces are removed:

Function	Action
LTRIM\$	Remove all spaces on the left side of the string (leading spaces only).
TRIM\$	Remove all leading and all trailing spaces; reduce all multiple, consecutive, embedded spaces to single spaces.
RTRIM\$	Remove all spaces on the right side of the string (trailing spaces only).

**See also** RTRIM\$, TRIM\$

### Examples

```
~  
1000      LINPUT USING RPAD$(FIELD$,40),FIELD$  
1010      FIELD$ = LTRIM$(RTRIM$(FIELD$))  
~
```

Statements

# MAT Statement

The matrix assignment statement, MAT, assigns one value to all elements of an array, copies the contents of one array to another, or initializes an array.

1 **MAT** *array-reference* = ( *expression* )

2 **MAT** *array-reference* = *array-name*

3 **MAT** *numeric-array-reference* = **CON** | **IDN** | **ZER**

---

*array-reference*      »    *array-name*  
                              *array-name*( *subscript* )  
                              *array-name*( *subscript*<sub>1</sub>, *subscript*<sub>2</sub> )

**Operation**      **Mode 1**—Sets each of the elements of *array-reference* to the value of *expression*. If **OPTION BASE 0** is in effect for the program, the 0 indexed elements are set.

**Mode 2**—The elements of *array-name* are assigned to the elements of *array-reference*. If *array-name* has fewer elements than *array-reference* then the extra elements of *array-reference* are cleared (set to 0 or null, as appropriate). If **OPTION BASE 0** is in effect for the program, the 0 indexed elements are set.

The arrays *array-reference* and *array-name* may be dissimilar. That is, one may be dimensioned as a one-dimensional array, the other as a two-dimensional array. During the copy operation, both arrays are treated as vector arrays or linear, one-dimensional arrays.

**Mode 3**—Sets the array to one of three patterns:

CON	Each element is set to one.
ZER	Each element is set to zero.
IDN	Each element is set to zero except on two-dimensional <i>numeric-array-reference</i> where the indices are equal (for instance: 1,1; 2,2; 3,3; <i>etc.</i> ), in which case the elements are set to one.

If **OPTION BASE 0** is in effect for the program, the 0 indexed elements are not set with **Mode 3**.

The MAT statement affects all of the elements of *array-reference* (except [Mode 3](#), which changes all but the 0 index elements). This does not mean that the entire array must be affected by the MAT statement. By using the alternate definitions of *array-reference*, an array can be temporarily re-dimensioned to a different size or a different number of dimensions.

To illustrate this, use the following code to create two small arrays and initialize them so the elements can be identified uniquely.

```
10 OPTION BASE 0
20
30 DIM A$(3,3),B$(3,3)
40
50 FOR I%=0 TO 3
60   FOR J% = 0 TO 3
70     A$(I%,J%) = STR$(I%)&" , "&STR$(J%)
80     B$(I%,J%) = I%*100+J%
80   NEXT
90 NEXT
```

As stated, normally, a MAT assignment affects the entire array:

Before						After				
A\$	0	1	2	3	MAT A\$ = ("X")	A\$	0	1	2	3
0	0,0	0,1	0,2	0,3		0	X	X	X	X
1	1,0	1,1	1,2	1,3		1	X	X	X	X
2	2,0	2,1	2,2	2,3		2	X	X	X	X
3	3,0	3,1	3,2	3,3		3	X	X	X	X

To change only the first 3 rows of the array use the syntax:

Before					MAT A\$(2,3) = ("X")	After				
A\$	0	1	2	3		A\$	0	1	2	3
0	0,0	0,1	0,2	0,3		0	X	X	X	X
1	1,0	1,1	1,2	1,3		1	X	X	X	X
2	2,0	2,1	2,2	2,3		2	X	X	X	X
3	3,0	3,1	3,2	3,3	3	3,0	3,1	3,2	3,3	

With this syntax the array is temporarily configured with a different size.

Arrays are maintained in memory as a single list of elements, whether the array is dimensioned with one or two dimensions. At any point the array can be treated as if it were dimensioned another way by merely reinterpreting the list references.

Because of this, it is also possible to temporarily treat a two-dimensional array as if it were dimensioned as a one-dimensional array, and vice versa:

Before					After				
B%	0	1	2	3	B%	0	1	2	3
0	0	1	2	3	0	9,999	9,999	9,999	9,999
1	100	101	102	103	1	9,999	9,999	9,999	103
2	200	201	202	203	2	200	201	202	203
3	300	301	302	303	3	300	301	302	303

MAT B%(6) = (9999)

A two-dimensional array is maintained as a list of elements, one after the other. Using the left array above, the B\$ array is kept as:

0	1	2	3	100	101	102	103	200	201	202	203	300	301	302	303
---	---	---	---	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----

This list can be reinterpreted many ways:

B%(15)  
 B%(1,7)  
 B%(2,4)  
 B%(3,3)  
 B%(4,2)  
 B%(5,1)  
 B%(7,1)  
 etc.

When an array is assigned values with the MAT statement, the number of elements affected by the assignment is computed by determining the number of elements in *array-reference*. For example, using the array dimensioned above:

```
MAT A$ = ("XYZ") \ REM 16 elements affected (4 rows X 4 columns)
MAT A$(2,1) = ("ABC") \ REM 6 elements affected (3 rows X 2 cols)
MAT A$(12) = (SPACE$(2)) \ REM 13 elements affected
```

The above calculations are greater than expected because of the OPTION BASE 0 statement which increases the number of rows and columns by one.

As stated earlier, the [Mode 3](#) form of the MAT statement does not affect the 0 index elements of arrays. To illustrate this, consider:

Before					After				
B%	0	1	2	3	B%	0	1	2	3
0	0	1	2	3	0	0	1	2	3
1	100	101	102	103	1	100	1	0	0
2	200	201	202	203	2	200	0	1	0
3	300	301	302	303	3	300	0	0	1

MAT B% = IDN

Notice that the top row and the left-hand column have unchanged values.

<b>Restrictions</b>	<p><i>array-reference</i> must match the type (string or numeric) of <i>expression</i>.</p> <p><i>array-reference</i> must match the type of <i>array-name</i>.</p> <p>The <i>array-reference</i>, <i>array-name</i> and <i>numeric-array-reference</i> must all be dimensioned arrays.</p> <p><i>array-reference</i> must not temporarily redimension the array to have more elements than it actually has. A “Subscript range error at line ...” occurs when this is attempted. If the program was compiled with the NOBOUND option this error is not reported or detected but it could cause program failure.</p>
<b>See also</b>	<p><a href="#">COMMON</a>, <a href="#">DIM</a>, <a href="#">LET</a>, <a href="#">MAT INPUT</a>, <a href="#">MAT PRINT</a>, <a href="#">MAT READ</a>, <a href="#">MAT READNEXT</a>, <a href="#">MAT READPREV</a>, <a href="#">MAT WRITE</a></p>

## Examples

<i>This routine is an</i>	201000	MOD.FILE:
<i>ON KEY event</i>		
<i>processing routine</i>	201010	
<i>that might be</i>	201020	EVENT.FLAG% = TRUE%
<i>invoked during the</i>	201030	
<i>"modifications</i>	201040	IF NOT NEW.ITEM% AND KEY\$REC\$(1) \ REM Key changed?
<i>prompt" of a file</i>	201050	DELETE #1,KEY\$ \ REM Remove old record
<i>maintenance pro-</i>	201060	IFEND
<i>gram.</i>	201070	
<i>After saving the</i>	201080	WRITE #1,REC\$(1): REC\$(2),REC\$(3),REC\$(4),REC\$(5),
<i>record to disk, the</i>		REC\$(6),REC\$(7)
<i>current record</i>	201090	
<i>fields are copied to</i>	201100	<b>MAT DUP\$ = REC\$</b> \ REM Save current record for use
<i>another array. The</i>		as duplication fields
<i>intention being that</i>	201110	<b>MAT DUP = REC</b>
<i>these copied fields</i>	201120	
<i>are then available</i>	201130	MOD.DONE% = TRUE%
<i>for duplication dur-</i>	201140	
<i>ing maintenance or</i>	201150	RESUME
<i>entry of the next</i>		
<i>record.</i>		
<i>Since arrays cannot</i>	1000	DIM FIELDS\$(40,40) \ REM Dimension for "worst case"
<i>be dynamically</i>	1010	
<i>allocated, the array</i>	~	
<i>is dimensioned with</i>	22030	INPUT "Rows and columns",ROWS%,COLS%
<i>the largest size that</i>	22040	
<i>might be used by</i>		
<i>the program.</i>		
<i>The actual array</i>	22050	<b>MAT FIELDS\$(ROWS%,COLS%) = ("")</b> \ REM Clear fields
<i>size is determined</i>	~	
<i>during program</i>		
<i>execution and the</i>		
<i>temporary redimen-</i>		
<i>sion abilities of the</i>		
<i>MAT statement are</i>		
<i>used to limit the</i>		
<i>array to the size</i>		
<i>desired.</i>		



## MAT INPUT Statement

MAT INPUT accepts data items from the console or an input file, saving the data in consecutive elements of an array.

```
1  MAT INPUT  array-reference
2  MAT INPUT  #channel: array-reference
3  MAT INPUT  #channel, key: array-reference

array-reference      »  array-name[,array-reference]
                       array-name(subscript)[, array-reference]
                       array-name(subscript1, subscript2)[, array-reference]
```

### Operation

**Mode 1**—Accept data from the console. One line of data is accepted from the console with each field in the line consecutively assigned to the elements of *array-reference*. A line of data is a string of characters terminated by a carriage return. The operation of this statement is similar to the [INPUT](#) statement:

```
DIM A$(5), B$(3)
MAT INPUT A$,B$
```

performs the same operation as:

```
INPUT A$(1),A$(2),A$(3),A$(4),A$(5),B$(1),B$(2),B$(3)
```

**Mode 2**—Accept ASCII formatted data from a sequential access data file. One record from the file is read; each field in the record is consecutively assigned to the elements of *array-reference*.

**Mode 3**—Accept ASCII formatted data from a direct, keyed, or indexed access data file. One record from the file is read; each field in the record is consecutively assigned to the elements of *array-reference*.

### Notes

Refer to the [MAT](#) statement, Notes section, for a description of its *array-reference* usage, which apply to this statement except that array elements with an index of 0 are never input with this statement.

Refer to the [INPUT](#) statement, Notes section, for a description of how fields are recognized during multiple-field input from the console, how to edit and terminate input from the console and the usage of the [INP](#) function.

When there are multiple array names specified, the elements in the first array are used first. When all of the elements in that array have been assigned values from the input line or record and there is still data remaining in the line or record, the data is assigned to the elements of the second or subsequent array.

The entire input line or record is always read with this statement. When there are more fields in the line or record input than array elements in *array-reference*, the extra input fields are ignored.

When there are more elements in *array-reference* than there are fields in the line or record, the extra elements are cleared to nulls: Extra string array fields are set to zero length strings; Extra numeric array fields are set to the value zero.

When no record exists ([Mode 2](#) or [Mode 3](#)), the EOF indicator is set and the elements in *array-reference* are set to nulls.

[Mode 2](#) and [Mode 3](#) cannot input a record written with the [WRITE](#) or [MAT WRITE](#) statements. Attempting to do so causes the elements in *array-reference* to be set to nulls. Use the [MAT READ](#) to retrieve records of this type.

Inputting a record from a file opened for UPDATE access causes that record to be locked so that other users are denied access until this record lock is released. Unless the file channel is opened with the MULTILOCK option, any attempt to input a record from the file channel releases all prior locks placed on records in this file channel. When the MULTILOCK option is used then other record locks are not released until a record is written to the file channel, the file channel is closed or the UNLOCK statement is used to release all of the locks on the file channel.

If an [ON EVENT](#), [ON KEY](#), [ON MOUSE](#) or [ON TIMEOUT](#) is in effect, a MAT INPUT statement may be terminated without entering a carriage return.



When the input is from the console and an [ON TIMEOUT](#) statement is in effect and the operator does not enter any characters or editing keys before the time-out period elapses, the MAT INPUT statement is terminated with the data already entered by the operator assigned to the *array-reference*. The time-out event handler is then invoked. As each character or editing key is entered the time-out timer is reset.

**I/O Redirection** [Mode 1](#) of this statement and [Mode 2](#) when *channel* is 0 actually gets the input from the current standard input device (stdin). Normally this is the console keyboard. However, stdin may have been redirected when the program was invoked:

```
>myprog < text.file
```

When stdin has been redirected the text accepted by this statement comes from that file, device or pipe.

When the standard output device (stdout) is redirected to a file or device other than the console display, the prompt character and the text accepted by this statement is echoed to stdout.

A program can determine if the stdin or stdout device has been redirected to a file or device other than the console by using the [SYS.ENV\\$](#) function:

```
IF SYS.ENV$(34, "STDIN") = "Y"
```

The above test is true when stdin has been redirected.

### Restrictions

The *array-reference* must refer to dimensioned arrays.

*array-reference* must not temporarily redimension the array to have more elements than it actually has. An “Inconsistent usage” error occurs after the entire existing array has been input.

When inputting from a file or device other than the console, the file channel must be opened with INPUT or UPDATE. Refer to the [OPEN](#) statement for details.

When inputting from a direct access file, the *key* must be a numeric expression. Inputting from a keyed or indexed access file requires that the *key* be a string expression.

### See also

[COMMON](#), [DIM](#), [INPUT](#), [MAT](#), [MAT PRINT](#), [MAT READ](#), [MAT READNEXT](#), [MAT READPREV](#), [MAT WRITE](#), [OPEN](#)

**Examples**

*The sales history  
file is opened and  
the current year's  
sales record is read  
with the **MAT  
INPUT** statement.*

*The creation of this  
record is shown in  
the **MAT PRINT**  
statement example.*

```
1000 DIM MONTHLY.SALES(12)
1010
1020 OPEN #1:"SALES.HISTORY", UPDATE INDEXED
1030
1040 MAT INPUT #1,DATE(0)[7:8]: MONTHLY.SALES
1050
1060 PRINT AT$(1,3);"The monthly sales for this year"
1030

1040 FOR I% = 1 TO 12
1050     PRINT "Month "&STR$(I%),MONTHLY.SALES(I%)
1060 NEXT
~
```

## MAT PRINT Statement

The MAT PRINT statement displays or writes one or more arrays to the console or an output file.

- 1 **MAT PRINT** *array-reference* [ *punct array-reference*] ... [ *punct*]
- 2 **MAT PRINT #channel:** *array-reference* [ *punct array-reference*]...
- 3 **MAT PRINT #channel, key:** *array-reference*

---

*array-reference*       »    *array-name*  
                              *array-name*( *subscript* )  
                              *array-name*( *subscript*<sub>1</sub>, *subscript*<sub>2</sub> )

*punct*                       »    ,  
                                      ;

### Operation

**Mode 1**—Displays an array or arrays on the console. The punctuation character may be a comma or semicolon. The trailing punctuation may be a comma, semicolon, or nothing. The punctuation character is applied to the formatting of the previous array's display.

Omitting the trailing punctuation character causes the preceding array to be displayed, one element per line.

Specifying a comma or semicolon for the punctuation character causes the preceding array to be displayed as a matrix, in multiple rows and columns. The comma character causes each element in a row to use one print field (21 characters); the semicolon character causes each element in a row to be displayed immediately adjacent to the preceding one (plus surrounding spaces if it is a numeric array).

When multiple arrays are listed, each array is displayed at the start of a new line.

**Mode 2**—Print an array or arrays to a sequential disk or tape file, printer, or communications port.

The elements of each array are output with no leading or trailing spaces, and each element is separated from the next by a comma. The punctuation character used in the statement does not matter.

When multiple arrays are listed, each array will be output at the start of a new line or record.

**Mode 3**—Print an array or arrays to a direct, keyed or indexed disk file.

The elements of each array are output with no leading or trailing spaces, and each element is separated from the next by a comma.

As with any write or print to a direct, keyed, or indexed disk file, the previous contents of the record with the same key are overwritten.

## Notes

Refer to the MAT statement, Notes section, for a description of its *array-reference* usage, which apply to this statement except that array elements with an index of 0 are never output with this statement. (With the exception of element 0,0, you can print all of a two-dimensional array's elements by temporarily redimensioning it as a one-dimensional array.)

When writing to a disk file with [Mode 2](#) or [Mode 3](#), the elements are written as if the option QUOTE were in effect for the file. Refer to the OPEN statement for a description of this option.

[Mode 1](#) of this statement is most useful as a debugging aid. But there is little control of the format of the display.

[Mode 2](#) and [Mode 3](#) of this statement are very useful for writing large amounts of data to a disk file with short and simple statements.

Information written with this statement can be read with the [INPUT](#), [MAT INPUT](#), or the [LINPUT](#) statements.

Printing a record to a file opened for UPDATE access without the MULTILOCK option causes all record locks on that file channel to be released.

**I/O Redirection** [Mode 1](#) of this statement and [Mode 2](#) when *channel* is 0 actually prints the text on the current standard output device (stdout). Normally this is the console display. However, stdout may have been redirected when the program was invoked:

```
>myprog > text.file
```

When stdout has been redirected the text displayed by this statement is output to that file, device or pipe.

A program can determine if the stdout device has been redirected to a file or device other than the console keyboard by using the [SYS.ENV\\$](#) function:

```
IF SYS.ENV$(34,"STDOUT") = "Y"
```

**Restrictions**      The *array-reference* must refer to dimensioned arrays.

*array-reference* must not temporarily redimension the array to have more elements than it actually has. An “Inconsistent usage” error occurs after the entire existing array has been printed.

When output to a file or device other than the console, the file channel must be opened with OUTPUT or UPDATE. Refer to the [OPEN](#) statement for details.

When printing to a direct access file, the *key* must be a numeric expression. Printing to keyed or indexed access files requires that the *key* be a string expression.

Since direct, keyed, and indexed files have fixed record lengths, when this statement prints an array to one of those file organizations ([Mode 3](#)) and the length of the record written is longer than the allocated record length, the trappable error “Truncated record” occurs.

**See also**            [COMMON](#), [DIM](#), [MAT WRITE](#), [OPEN](#), [PRINT](#), [PRINT USING](#)

<b>Examples</b>	1000	DIM MONTHLY.SALES(12)
	1010	
<i>The dollar sales</i>	1020	PRINT AT\$(1,3);"Enter the monthly sales for this year"
<i>totals for the cur-</i>	1030	
<i>rent 12 month</i>	1040	FOR I% = 1 TO 12
<i>period are col-</i>	1050	INPUT "Month "&STR\$(I%),MONTHLY.SALES(I%)
<i>lected.</i>	1060	NEXT
	1070	
	1080	OPEN #1:"SALES.HISTORY", UPDATE INDEXED
	1090	
<i>The sales totals are</i>	1100	<b>MAT PRINT #1,DATE(0)[7:8]: MONTHLY.SALES</b>
<i>written to disk.</i>	1110	
	1120	CLOSE #1

Statements

## MAT READ Statement

MAT READ accepts data items from [DATA](#) statements or an input file, saving the data in consecutive elements of an array or arrays.

- 1 **MAT READ** *array-reference*
- 2 **MAT READ** *#channel:* *array-reference*
- 3 **MAT READ** *#channel, key:* *array-reference*

---

*array-reference*      »    *array-name*[, *array-reference*]  
                              *array-name*( *subscript* )[, *array-reference*]  
                              *array-name*( *subscript*<sub>1</sub>, *subscript*<sub>2</sub> )[, *array-reference*]

### Operation

**Mode 1**—Data is read or copied from [DATA](#) statements in this program into the elements of *array-reference*. When there are insufficient data items left to assign to all of the elements of *array-reference*, a trappable error of “Out of data” occurs. Extra data items are not used and are available for the next MAT READ or [READ](#), [READNEXT](#), [READPREV](#) statement that executes.

**Mode 2**—Reads a record from a sequential access disk or tape file. One record from the file is read and each field in the record is consecutively assigned to the elements of *array-reference*.

**Mode 3**—Reads a record from a direct, keyed, or indexed access disk file. One record from the file is read and each field in the record is consecutively assigned to the elements of *array-reference*.

### Notes

Refer to the [MAT](#) statement, Notes section, for a description of its *array-reference* usage, which apply to this statement except that array elements with an index of 0 are never read with this statement. (With the exception of element 0,0, you can read all of a two-dimensional array’s elements by temporarily redimensioning it as a one-dimensional array.)

Refer to the [READ](#) statement, [DATA & RESTORE](#) section for a description of the usage of [DATA](#) statements and the [RESTORE](#) statement.

When there are multiple array names specified, the elements in the first array are used first. When all of the elements in that array have been assigned values from the [DATA](#) statements or input record and there is still



data remaining in the record, the data is assigned to the elements of the second or subsequent array.

When reading from a disk or tape file and there are more elements in *array-reference* than there are fields in the record, the extra elements are cleared to nulls. If there are fewer elements in *array-reference* than there are fields in the record, the extra record fields are ignored.

When no record exists matching the key ([Mode 2](#) and [Mode 3](#)), the [EOF](#) indicator is set and the elements in *array-reference* are cleared to nulls.

Records in files cannot be read if they were created by the [MAT PRINT](#) or the [PRINT](#) statement. Attempting to do so causes the elements of the *array-reference* to be cleared to nulls. Use the [MAT INPUT](#), [INPUT](#) or [LINPUT](#) statements to access records of this type.

Reading a record from a file that was opened with UPDATE mode causes a record-lock to be placed on the record. Unless the file channel was opened with the MULTILOCK option, any other record-lock in the file is removed. This unlock operation is performed whether or not this MAT READ is successful.

If an attempt is made to read a record that is locked by another user or task, this program will wait until the record-lock is released by that other program. If [OPTION LOCK](#) is in effect for this program, a trappable error number 48 is generated after the indicated time.

Unless option MULTILOCK is in effect for the file channel, record-locks are removed or released by reading another record from the same file channel, writing a record to the same file channel, or deleting this record. Performing the UNLOCK statement on this file channel, or closing the file channel always unlocks records in the file.

Refer to the *MultiUser BASIC Programmer's Guide* for additional information about using files.

## Restrictions

The *array-reference* must refer to dimensioned arrays.

*array-reference* must not temporarily redimension the array to have more elements than it actually has. An “Inconsistent usage” error occurs after the entire existing array has been read.

When reading from a file the file channel must be opened with INPUT or UPDATE. Refer to the [OPEN](#) statement for details.

When reading from a direct access file, the *key* must be a numeric expression. Reading from a keyed or indexed access file requires that the *key* be a string expression.

**See also** [COMMON](#), [DATA](#), [DIM](#), [MAT READNEXT](#), [MAT READPREV](#), [OPEN](#), [READ](#), [READNEXT](#), [READPREV](#), [RESTORE](#)

### Examples

*A record from an indexed access file is read and the fields are assigned to the first 10 fields in the array REC\$.*

*An array is dimensioned to hold 100 city names.*

*The MAT READ initializes the array by reading the next 100 DATA items.*

```

11000 READ.CUSTOMER:
11010
11020     MAT READ #1,KEY$: REC$(10)
11030
11040     IF EOF(1) THEN NEW.ITEM% = TRUE% \ REM Not found?
11050
11060     RETURN
~
900500     DIM REC$(50) \ REM General purpose rec field array
900510     DIM CITIES$(100)
~
900690     MAT READ CITIES$
~
980230 REM Bay area cities
980240
980250     DATA "Alameda"
980260     DATA "Alamo"
980270     DATA "Albany"
980280     DATA "Antioch"
980290     DATA "Aptos"
980300     DATA "Atherton"
980310     DATA "Belmont"
980320     DATA "Belvedere"
980330     DATA "Ben Lomond"
980340     DATA "Benicia"
980350     DATA "Berkeley"
~

```

# MAT READNEXT Statement

MAT READNEXT accepts data items from the next record in a direct, keyed, or indexed access file, saving the data in consecutive elements of an array or arrays.

MAT READNEXT #channel, key-variable: array-reference

array-reference

»

array-name[, array-reference]

array-name( subscript )[, array-reference]

array-name( subscript<sub>1</sub>, subscript<sub>2</sub>)[, array-reference]

**Operation** The next record in the direct, keyed, or indexed access file is read with the fields of that record assigned to the consecutive elements of *array-reference*. The key of the record read is assigned to the *key-variable* variable.

**Notes** Refer to the [MAT](#) statement, Notes section, for a description of its *array-reference* usage, which apply to this statement except that array elements with an index of 0 are never read with this statement. (Except for element 0,0, you can read all of a two-dimensional array's elements by temporarily redimensioning it as a one-dimensional array.)

When there are multiple array names specified, the elements in the first array are used first. When all of the elements in that array have been assigned values from the record and there is still data remaining in the record, the data is assigned to the elements of the second or subsequent array.

When there are more elements in *array-reference* than there are fields in the record, the extra elements are cleared to nulls. If there are fewer elements in *array-reference* than there are fields in the record, the extra fields are ignored.

When there are no more records in the file, the [EOF](#) indicator is set, the elements in *array-reference* and the *key-variable* variables are cleared to nulls.

This statement may only access direct, keyed, and indexed access files. Sequential files perform a read next naturally when a READ or [MAT READ](#) is performed.

Records in files cannot be read if they were created by the MAT PRINT or the PRINT statement. Attempting to do so sets the elements of the array-

Statements

reference to nulls. Use the [MAT INPUT](#), [INPUT](#) or [LINPUT](#) statements to access records of this type.

The next record in a direct access file skips any deleted and never written record positions. The next record in a keyed access file is the next physical record in the file. The next record in an indexed access file is the next record in the file according to the sorting order of the record keys.

Reading a record from a file that was opened with UPDATE mode causes a record-lock to be placed on the record. unless the file channel was opened with the MULTILOCK option, any other record-lock in the file is removed. This unlock operation is performed whether or not this MAT READNEXT is successful (end-of-file encountered).

If an attempt is made to read a record that is locked by another user or task, this program will wait until the record-lock is released by that other program. If OPTION LOCK is in effect for this program, a trappable error number 48 is generated after the indicated time.

Unless option MULTILOCK is in effect for the file channel, record-locks are removed or released by reading another record from the same file channel, writing a record to the same file channel or deleting this record. Performing the UNLOCK statement on this file channel, or closing the file channel always unlocks records in the file.

Refer to the *MultiUser BASIC Programmer's Guide* for a discussion on using files and record pointers.

#### Restrictions

The *array-reference* must refer to dimensioned arrays.

*array-reference* must not temporarily redimension the array to have more elements than it actually has. An “Inconsistent usage” error occurs after the entire existing array has been read.

When reading from a file the file channel must be opened with INPUT or UPDATE. Refer to the [OPEN](#) statement for details.

When reading from a direct access file, the *key-variable* must be a numeric variable. Reading from a keyed or indexed access file requires that the *key-variable* be a string variable.

#### See also

[COMMON](#), [DIM](#), [MAT READ](#), [MAT READPREV](#), [OPEN](#), [READ](#), [READNEXT](#), [READPREV](#)

## Examples

*The customer master file is opened and the first record in the file is read.*

10  
20  
30  
40  
50  
60

DIM REC\$(10)

OPEN #20: "CUSTOMER.MASTER", INPUT INDEXED

**MAT READNEXT #20,KEY\$: REC\$**

*As long as the last read was successful, display the customer name, city, state, and ZIP Code. Read the next record in the file and repeat the loop.*

70  
80  
90  
100  
110  
120  
130  
140

WHILE NOT EOF(20) \ REM Repeat until end of file

PRINT KEY\$;": ";REC\$(2);", ";REC\$(3);" ";REC\$(4)

**MAT READNEXT #20,KEY\$: REC\$**

WEND

*Close the file and exit the program.*

150  
160  
170

CLOSE #20

END

---

# MAT READPREV Statement

MAT READPREV accepts data items from the previous record in an indexed access file, saving the data in consecutive elements of an array or arrays.

**MAT READPREV** *#channel, key-variable: array-reference*

*array-reference*      »    *array-name*[, *array-reference*]  
                              *array-name*( *subscript* )[, *array-reference*]  
                              *array-name*( *subscript*<sub>1</sub>, *subscript*<sub>2</sub> )[, *array-reference*]

**Operation**

The previous record in the indexed access file is read with the fields of that record assigned to the consecutive elements of *array-reference*. The key of the record read is assigned to the *key-variable* variable.

**Notes**

Refer to the [MAT](#) statement, Notes section, for a description of its *array-reference* usage, which apply to this statement except that array elements with an index of 0 are never read with this statement. (Except for element 0,0, you can read all of a two-dimensional array's elements by temporarily redimensioning it as a one-dimensional array.)

When there are multiple array names specified, the elements in the first array are used first. When all of the elements in that array have been assigned values from the input record and there is still data remaining in the record, the data is assigned to the elements of the second or subsequent array.

When there are more elements in *array-reference* than there are fields in the record, the extra elements are cleared to nulls. If there are fewer elements in *array-reference* than there are fields in the record, the extra fields are ignored.

When there are no previous records in the file, the [EOF](#) indicator is set, the elements in *array-reference* and the *key-variable* variable are cleared to nulls.

Records in files cannot be read if they were created by the [MAT PRINT](#) or the [PRINT](#) statement. Attempting to do so sets the elements of the *array-reference* to nulls. Use the [MAT INPUT](#), [INPUT](#) or [LINPUT](#) statements to access records of this type.

The previous record in an indexed access file is the previous record in the file according to the sorting order of the record keys.

Reading a record from a file that was opened with UPDATE mode causes a record-lock to be placed on the record. unless the file channel was opened with the MULTILOCK option, any other record-lock in the file is removed. This unlock operation is performed whether or not this MAT READPREV successful (end-of-file encountered).

If an attempt is made to read a record that is locked by another user or task, this program will wait until the record-lock is released by that other program. If [OPTION LOCK](#) is in effect for this program, a trappable error number 48 is generated after the indicated time.

Unless option MULTILOCK is in effect for the file channel, record-locks are removed or released by reading another record from the same file channel, writing a record to the same file channel or deleting this record. Performing the [UNLOCK](#) statement on this file channel, or closing the file channel always unlocks records in the file.

Refer to the *MultiUser BASIC Programmer's Guide* for a discussion on using files and record pointers.

**Restrictions**

The *array-reference* must refer to dimensioned arrays.

This statement may only access indexed access files.

*array-reference* must not temporarily redimension the array to have more elements than it actually has. An “Inconsistent usage” error occurs after the entire existing array has been read.

When reading from a file the file channel must be opened with INPUT or UPDATE. Refer to the [OPEN](#) statement for details.

The *key-variable* must be a string variable.

**See also**

[COMMON](#), [DIM](#), [MAT READ](#), [MAT READNEXT](#), [OPEN](#), [READ](#), [READNEXT](#), [READPREV](#)

**Examples**

*The customer master file is opened and the last record in the file is read. A key of one or more CHR\$(255) will position to the end of an indexed file because 255 is greater than all other characters.*

*As long as the last read was successful, display the customer name, city, state, and ZIP Code. Read the previous record in the file and repeat the loop.*

*Close the file and exit the program.*

```

10  DIM REC$(10)
20
30  OPEN #20: "CUSTOMER.MASTER", INPUT INDEXED
40
50  READ #20,CHR$(255):REC$(1) \ REM Position to eof
50  MAT READPREV #20,KEY$: REC$
60

70  WHILE NOT EOF(20) \ REM Repeat until start of file
80
90      PRINT KEY$;" : ";REC$(2);", " ;REC$(3);" " ;REC$(4)
100
110     MAT READPREV #20,KEY$: REC$
120
130     WEND
140

150  CLOSE #20
160
170  END

```



## MAT SORT Statement

The MAT SORT statement creates a “sorted index array” of indices to another array.

**MAT** *numeric-array-reference* = **SORT**( *source-array-name* )

---

*numeric-array-reference*» *numeric-array-name*  
*numeric-array-name*( *subscript* )  
*numeric-array-name*( *subscript*<sub>1</sub>, *subscript*<sub>2</sub> )

### Operation

The indices of *source-array-name* are sorted according to the ascending value of the elements of *source-array-name* and saved in the *numeric-array-reference*.

### Notes

Refer to the [MAT](#) statement, Notes section, for a description of its *array-reference* usage, which applies to this statement’s *numeric-array-reference* usage.

The *source-array-name* and the *numeric-array-reference* are both treated as vector arrays. That is, they are treated as one-dimensional arrays. The indices saved in *numeric-array-reference* are the indices of *source-array-name*.

When [OPTION BASE 0](#) is in effect, the 0 index of *source-array-name* is ignored and the 0 index of *numeric-array-reference* is unchanged.

```
10 OPTION BASE 0
30
40 DIM B%(3,3),SORT.IDX%(15),C%(15)
50
60 FOR I% = 0 TO 3
70     FOR J% = 0 TO 3
80         B%(I%,J%) = RND*1000 \ REM Generate random integer
90     NEXT
100 NEXT
110
120 B%(0,0) = 999
130
140 MAT C% = B%
```

After execution of the above code, the array B% contains:

B%	0	1	2	3
0	999	211	242	136
1	904	31	15	1
2	133	746	36	309
3	469	827	898	280

Interpreted as a one-dimensional array:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
999	211	242	136	904	31	15	1	133	746	36	309	469	827	898	280

Since the B% array was assigned to the C% array, after performing:

```
150 MAT SORT.IDX% = SORT(C%)
```

the C% array will be unchanged and the SORT.IDX% array contains:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	7	6	5	10	8	3	1	2	15	11	12	9	13	14	4

Note that C%(0) was ignored and that SORT.IDX%(0) contains a zero, which is what it contained prior to the MAT SORT. The SORT.IDX% array can be used to index into the C% array in ascending sequence:

```
160
170 FOR I% = 1 TO 15
180   PRINT C%(SORT.IDX%(I%));", ";
190   NEXT
```

1 , 15 , 31 , 36 , 133 , 136 , 211 , 242 , 280 , 309 , 468 , 746 ,  
827 , 898 , 904 ,

The *source-array-name* is not changed by this statement. Only the indices to *source-array-name* are reordered and saved in *numeric-array-reference*.

The size of *numeric-array-reference* may be equal to, larger than, or smaller than the size of *source-array-name*. When the two arrays have the same number of elements, the indices to the entire *source-array-name* is sorted, ignoring the (0) index of *source-array-name* when present.

When *numeric-array-reference* has fewer elements than *source-array-name*, only as many elements as are in *numeric-array-reference* are sorted. When *numeric-array-reference* has more elements than *source-array-name*, the extra elements in *numeric-array-reference* are set to 0.

Thus, after performing

```
MAT SORT.IDX%(5) = SORT(C%)
```

the `SORT.IDX%` array will contain the sorted indices to the first 5 elements of `C%`:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	5	3	1	2	4	3	1	2	15	11	12	9	13	14	4

The elements after `SORT.IDX%(5)` have values that are unchanged from their prior contents. It is best to clear the array prior to performing a `MAT SORT`. In this case, only the first 5 elements of `SORT.IDX%` contain valid information.

The *source-array-name* may be a numeric or string array.

The *source-array-name* may be a two-dimensional array. However, the indices to that array will always be sorted as indices to a one-dimensional array. Thus, the result of a `MAT SORT` of a two-dimensional array are useless unless the array is redimensioned as a one-dimensional array or copied to a one-dimensional array, as in the example above.

**Restrictions** The *numeric-array-reference* and *source-array-name* must be dimensioned arrays.

*numeric-array-reference* must not temporarily redimension the array to have more elements than it actually has. An “Inconsistent usage” error occurs after the entire existing array has been assigned the sorted indices.

**See also** [COMMON](#), [DIM](#)

Examples

<i>Arrays are dimensioned with</i>	520	DIM CODES\$(51),STATES\$(51),SORT.IDX%(100)
<i>SORT.IDX% large enough to do this task and others.</i>	530	
	540	FOR I% = 1 TO 51
	550	READ CODES\$(I%),STATES\$(I%)
	560	NEXT
	570	
<i>The arrays are initialized and then sorted.</i>	580	<b>MAT SORT.IDX%(51) = SORT(STATES\$)</b>
	590	
	600	PRINT "The states of the U.S.A., sorted alphabetically, are:"
	610	
<i>The sorted indices of the state arrays are used to access those arrays in the proper sequence.</i>	620	FOR I% = 1 TO 51
	630	PRINT CODES\$(SORT.IDX%(I%));" ";
		STATES\$(SORT.IDX%(I%))
	640	NEXT
	~	
	990000	DATA "GA","Georgia"
	990010	DATA "CA","California"
	~	

## MAT WRITE Statement

MAT WRITE outputs an array or arrays to a file.

- 1 **MAT WRITE** *#channel:* *array-reference*
- 2 **MAT WRITE** *#channel, key:* *array-reference*

---

*array-reference*      »    *array-name*[, *array-reference*]  
                              *array-name*( *subscript* )[, *array-reference*]  
                              *array-name*( *subscript*<sub>1</sub>, *subscript*<sub>2</sub> )[, *array-reference*]

**Operation**      **Mode 1**—Writes an array to a formatted record in a sequential access disk or tape file.

**Mode 2**—Writes an array to a formatted record in a direct, keyed, or indexed access disk file.

**Notes**            Refer to the MAT statement, Notes section, for a description of its *array-reference* usage, which apply to this statement except that array elements with an index of 0 are never output with this statement. (With the exception of element 0,0, you can write all of a two-dimensional array's elements by temporarily redimensioning it as a vector array.)

When there are multiple array names specified, the elements in the first array are used first. When all of the elements in that array have been written the second and subsequent arrays are written to the same record following the first array.

Information written with this statement can be read with the [READ](#), [READNEXT](#), [READPREV](#), [MAT READ](#), [MAT READNEXT](#) or [MAT READPREV](#) statements.

When the file channel was opened with option UPDATE records are locked by each read operation on the file channel. Similar to the [WRITE](#) statement, unless the file was also opened with the option MULTILOCK, the MAT WRITE statement unlocks all records currently locked on the file channel. A file opened with MULTILOCK causes all records locked to remain locked until the file channel is closed or until a [UNLOCK](#) statement is performed on the file channel.

Restrictions

The *array-reference* must refer to dimensioned arrays.

The *channel* must refer to a file or device opened with OUTPUT or UPDATE. Refer to the [OPEN](#) statement for details.

When writing to a direct access file, the *key* must be a numeric expression. Writing to a keyed or indexed access file requires that the *key* be a string expression.

Since direct, keyed, and indexed files have fixed record lengths, when this statement writes an array to one of those file organizations ([Mode 2](#)) and the length of the record written is longer than the allocated record length, the trappable error “Truncated record” occurs.

See also

[COMMON](#), [DIM](#), [MAT PRINT](#), [OPEN](#), [WRITE](#)

Examples

This routine is an ON KEY event processing routine that might be invoked during the "modifications prompt" of a file maintenance program.

The record is saved to disk with the MAT WRITE statement.

```

201000  MOD.FILE:
201010
201020      EVENT.FLAG% = TRUE%
201030
201040      IF NOT NEW.ITEM% AND KEY$>ORIG.KEY$ ! key changed?
201050          DELETE #1,ORIG.KEY$ ! Remove old version
201060          IFEND
201070
201080      MAT WRITE #1,KEY$:REC$
201090
201100      MAT DUP$ = REC$      ! Save as duplication fields
201110
201120      RESUME

```

## MATCH Function

MATCH returns the starting position of a search string or pattern in another string.

**MATCH**( *string-expression*, *pattern-string* )

### Operation

The *string-expression* is compared to *pattern-string*. The value returned is the character position of the start of *pattern-string* in *string-expression*. A zero return value indicates that no match was found. The value returned can be used as a logical expression value as zero is the same as false.

The *pattern-string* contains special matching characters and literal characters. The special matching characters are:

Character	Meaning
?	Any single character matches
@	Any single letter or space matches
#	Any single digit matches (not decimal points or commas)
*?	Any number (0—255) of any characters match
*@	Any number (0—255) of letters or space characters match
*#	Any number (0—255) of digits match
%	A special <i>escape</i> character that allows specification of ? @ # and % characters as literals. This character is followed by the special character that you want treated as a normal character.

### See also

[SCH](#)

### Examples

```
13100 VALIDATE.SSN:
13110
13120     IF NOT MATCH$(SSN$,"###-##-####")
13130         VALID% = FALSE%
13140         PRINT "INVALID SSN - FORMAT: 123-45-6789"
13150     IFEND
13160
13170     RETURN
13180
13200 VALIDATE.VENDOR.CODE:
13210
13220     IF NOT MATCH$(VCODE$,"@@@###")
13230         VALID% = FALSE%
13240         PRINT "INVALID VENDOR CODE - FORMAT: ABC123"
13250     IFEND
13260
13270     RETURN
```

---

# MAX Function

MAX returns the larger value of two expressions.

MAX( *numeric-expression*<sub>1</sub>, *numeric-expression*<sub>2</sub> )

**Operation**      The larger value of *numeric-expression*<sub>1</sub> and *numeric-expression*<sub>2</sub> is returned.

**Notes**            The sequence of *numeric-expression*<sub>1</sub> and *numeric-expression*<sub>2</sub> does not matter. That is,

$$\text{MAX}(A,B) \Leftrightarrow \text{MAX}(B,A).$$

**See also**        [MIN](#)

**Examples**

```
1200      INPUT "Item selection: ",ITEM%
1210
1220      ITEM% = MAX(1,ITEM%) \ REM Make sure >= 1
1230      ITEM% = MIN(50,ITEM%) \ REM Make sure <= 50
1240
```



---

# MID\$ Function

MID\$ returns the middle portion of a string.

MID\$( *string-expression*, *from*, *count* )

**Operation** Starting with character position *from*, *count* characters of the *string-expression* are returned.

**Notes** The substring expression can perform the same operation:

MID\$(A\$,5,3) ⇔ A\$[5:7]

**See also** [LEFT\\$, RIGHT\\$](#)

## Examples

<i>This code section examines a string field and picks off the second word in the string.</i>	5300 TEMP\$ = TRIM\$(FIELD\$) \ REM Remove extra spaces 5310
<i>First, the locations of the first and second spaces are determined.</i>	5320 T1% = SCH(1,TEMP\$," ") \ REM Find 1st space 5330 T2% = SCH(T1%+1,TEMP\$," ") \ REM Find 2nd space 5340
<i>Then, using those locations, a middle portion of the field is extracted with the MID\$ function.</i>	5350 WORD2\$ = MID\$(TEMP\$,T1%+1,T2%-T1%-1) \ REM Get 2nd word 5360

Statements

---

# MIN Function

MIN returns the smaller value of two expressions.

MIN( *numeric-expression*<sub>1</sub>, *numeric-expression*<sub>2</sub> )

**Operation**      The smaller value of *numeric-expression*<sub>1</sub> and *numeric-expression*<sub>2</sub> is returned.

**Notes**            The sequence of *numeric-expression*<sub>1</sub> and *numeric-expression*<sub>2</sub> does not matter. That is,

MIN(A,B) ⇔ MIN(B,A).

**See also**        [MAX](#)

## Examples

*This code accepts a value from the operator and then forces the value to be in the range 1—23.*

*Both sections of code accomplish the same thing.*

*ITEM% is either in the range or set to the appropriate range boundary.*

```
1200     INPUT "Item selection: ",ITEM%
1210
1220     ITEM% = MIN(23,MAX(ITEM%,1)) \ REM make sure in range
1230
~
1200     INPUT "Item selection: ",ITEM%
1210
1220     IF ITEM%1 THEN ITEM% = 1
1230     IF ITEM%>23 THEN ITEM% = 23
1240
```

Statements

# MOD Function

MOD returns the *modulo* or remainder of two values.

MOD( *numeric-expression*, *base-number* )

**Operation**      The value of *numeric-expression* is divided by the value of *base-number* and the remainder is returned.

**Notes**            The sign of the return value will be the same as the sign of *numeric-expression*.

The modulo of a number can also be computed by the equation:

$$\text{MOD}(A,B) = \text{FP}(A/B) * B$$

or

$$A - \text{IP}(A/B) * B$$

**Examples**            Note: This example is correct but the purpose of the code is more efficiently performed by the STRFTIME\$ function.

<i>This code determines today's day of the week name.</i>	900230	DIM DAY.NAMES\$(7)
	900240	
	900250	MAT READ DAY.NAMES\$
	900260	
	900270	DAYNAME\$ = DAY.NAMES\$(1+MOD(DAY(DATES(0)),7))
<i>The MOD function is used with today's day number, which is the number of days since January 1, 1900, which, coincidentally, was a Monday.</i>	900280	~
<i>Since the array is dimensioned with OPTION BASE 1, a 1 is added to the MOD function return so that the index into the DAY.NAME\$ array is proper.</i>	990060	DATA "Sun","Mon","Tue","Wed","Thu","Fri","Sat,"
	~	

Statements

---

# MOUNT Statement

MOUNT informs the operating system that one or more diskettes, discs or removable hard disks have been changed.

**MOUNT** *string-expression*

**Operation**      The disks in the drives specified in *string-expression* are mounted.

**Notes**            The THEOS operating system maintains a copy of each drive's disk label and format. Attempting to access a diskette that has changed without the operating system's knowledge will cause an error because the system will notice that the new disk's label or format is different from its saved information.

The mount operation causes the THEOS operating system to read the label and format information of the current disk in the drive and to save that information for subsequent disk access.

Do not perform the MOUNT statement until the diskette, disk or disc has been changed.

More than one drive code can be specified with *string-expression*. For example:

Statements

**Examples**

```
MOUNT "FG" \ REM Change F and G

200      PRINT "Change diskettes"
210      PRINT "Okay to proceed ";
220
230      CHANGE$ = YESNO$
240
250      IF CHANGE$ = "Y" THEN MOUNT "FG"
```

# MOUSE.BUTTON, .COL, .FRAME, .ROW, .WINDOW Functions

The MOUSE.BUTTON, MOUSE.COL, MOUSE.ROW and MOUSE.WINDOW functions return the value of the last mouse button used or the column, row or window number of the mouse cursor when the last mouse event occurred.

The MOUSE.FRAME function returns a value indicating whether or not the mouse cursor was on a window frame when the last mouse event trap occurred.

MOUSE.BUTTON

MOUSE.COL

MOUSE.FRAME

MOUSE.ROW

MOUSE.WINDOW

- Operation

MOUSE.BUTTON returns the coded value of the mouse button pressed.

MOUSE.COL and MOUSE.ROW return the column and row of the mouse cursor when the last mouse event occurred.

MOUSE.WINDOW returns the window number of the mouse cursor when the last mouse event occurred.

MOUSE.FRAME returns the coded value of the mouse cursor position on a window's frame.

MOUSE.BUTTON Notes

MOUSE.BUTTON returns one of the following integer values:

Button	Code	Button	Code
Right down	1	Right double click	64
Center down	2	Center double click	128
Left down	4	Left double click	256
Right click	8	Right drag	512
Center click	16	Center drag	1,024
Left click	32	Left drag	2,048

A mouse *click* refers to a mouse button being depressed and then released; a *double click* refers to a mouse button being clicked twice in a short period

of time; a mouse *drag* refers to the situation where a mouse button is depressed and the mouse is moved prior to releasing the button.

**MOUSE.COL,  
MOUSE.ROW  
Notes**

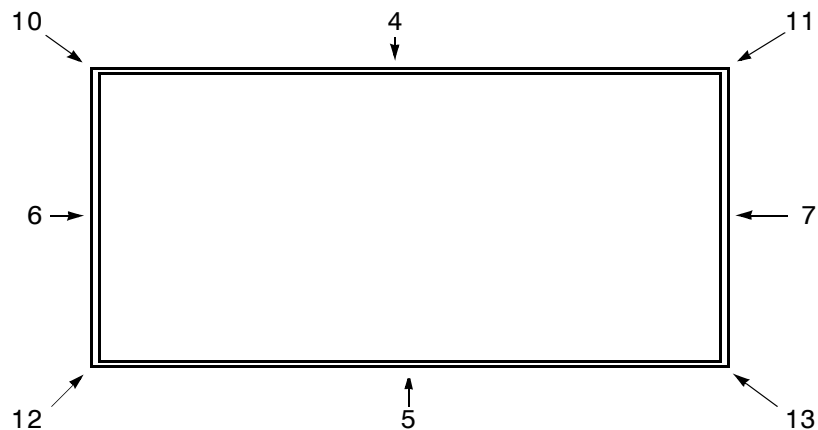
The MOUSE.COL and MOUSE.ROW values are relative to the origin of the top-most window under the mouse cursor. (If no windows are open then the values are relative to origin of window 0.) If the mouse cursor was inside of a window then the MOUSE.COL and MOUSE.ROW functions are relative to the origin of the interior of the window. If the mouse cursor was on the frame of the window then the MOUSE.COL and MOUSE.ROW functions are relative to the origin of the frame, not the interior. (The origin of the frame is one column and row less than the origin of the interior.)

**MOUSE.FRAME  
Notes**

The MOUSE.FRAME function returns one of the following integer values:

Location	Code	Location	Code
Not on frame	0		
Top frame	4	Top left corner	10
Bottom frame	5	Top right corner	11
Left frame	6	Bottom left corner	12
Right frame	7	Bottom right corner	13

These location codes correspond to these frame locations:



When a non-zero is returned by MOUSE.FRAME, the MOUSE.COL and MOUSE.ROW functions can be used to identify the specific location of the mouse cursor on the frame. For instance, when the MOUSE.FRAME returns a four indicating that the mouse cursor was on the top frame of a window, the MOUSE.COL function will return a value indicating the specific column number of the frame member. Similarly, when the MOUSE.FRAME returns a six indicating that the mouse cursor was on the left frame of a window,

the MOUSE.ROW function returns a value indicating the specific row number of the frame member.

The value of the unused MOUSE.COL or MOUSE.ROW is set to zero.

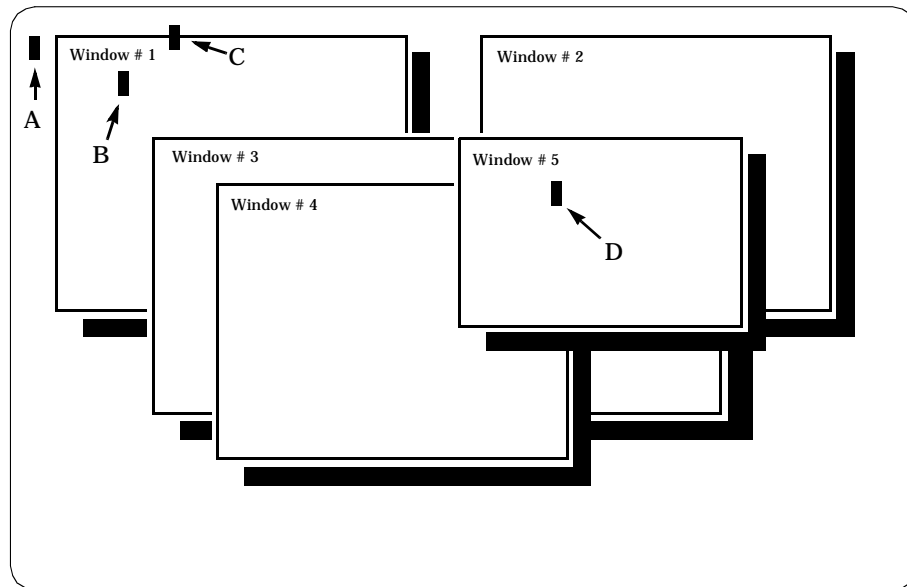
**MOUSE.WINDOW  
Notes**

When the program is running in a non-windowed environment, the MOUSE.WINDOW value is always zero. When running in a windowed environment, the MOUSE.WINDOW value is the number of the top-most, visible window under the mouse cursor.

When the mouse cursor is over the *shadow* of a window it is actually in the window below the window with the shadow.

In the illustration on the following page there are four mouse cursor locations labeled A, B, C and D.

- A. This mouse cursor location is positioned outside of any window. When a mouse event occurs at that location the MOUSE.WINDOW function returns a zero since the mouse cursor is in window number zero. The values for MOUSE.COL and MOUSE.ROW will be the mouse cursor location relative to the window zero origin which is the screen origin. The value for MOUSE.FRAME will be zero.
- B. Mouse cursor location “B” is inside of window # 1 and the MOUSE.WINDOW function returns the value one. The values for MOUSE.COL and MOUSE.ROW will be six and two respectively as they are relative to the window origin. The value for MOUSE.FRAME will be zero.
- C. Mouse cursor location “C” is on the frame of window # 1. The MOUSE.WINDOW function returns the value one and the MOUSE.FRAME will be four because the cursor is on the top portion of the window frame. The MOUSE.ROW function returns a zero and the MOUSE.COL returns a 14.
- D. Mouse cursor location “D” is at a location with five windows underneath it. From the lowest to the highest, these windows are 0, 2, 3, 4 and 5. Because window number 5 is the top-most visible window the MOUSE.WINDOW function returns a five. For the same reason MOUSE.COL and MOUSE.ROW return a ten and a two respectively as those are the column and row values relative to the origin of window five. The MOUSE.FRAME returns a zero.



### Restrictions

These functions only return values for the mouse event that was trapped. For instance, when the ON LEFT CLICK statement is the only event trapping statement in effect, and the operator performs a double click or a mouse drag, the event trap is not invoked. Any values returned by these functions will reflect the mouse cursor and status at the time of the last mouse event trapped, not the current mouse cursor location on the screen.

The values of these functions are only guaranteed to be accurate if it is used as the first statement in the mouse event handler routine or if the first statement was an ON MOUSE OFF statement. This is due to the fact that only the specific event being trapped is suspended when the event handler is invoked. Other mouse events are not disabled or suspended and it is possible that the operator might generate another mouse event before this mouse event has been completely handled.

### Statements

### See also

[ON MOUSE](#)



## Examples

```

4000 MOUSE.CLICK:!! Mouse event handler for mouse button events
4010
4020     ON MOUSE OFF                ! Suspend mouse events
4030
4040     BUTTON.TYPE% = MOUSE.BUTTON ! Save type of mouse action
4050     BUTTON% = 0                 ! Initially, no match
4060
4070     FOR MI%=1 TO 10              ! For each defined "button"
4080
4090         IF MOUSE.WINDOW=BUTTON.WIN%(MI%) ! Possible candidate?
4100         IF BUTTON.STATUS%(MI%)           ! Button enabled?
4110             IF MOUSE.COL>BUTTON.X%(MI%) AND
4120                 MOUSE.COL<BUTTON.TO.X%(MI%)
4130                 IF MOUSE.ROW>=BUTTON.Y%(MI%) AND
4140                     MOUSE.ROW<=BUTTON.TO.Y%(MI%)
4150                     BUTTON% = MI%           ! We have a match!
4160                     BREAK ! Exit FOR-NEXT
4170                     IFEND
4180                 IFEND
4190             IFEND
4200         NEXT
4210     ON MOUSE ON                ! Resume mouse trapping
4220
4230     RESUME                      ! Return to program

```

---

# MSEC Function

MSEC returns the current time of day, expressed as the number of milliseconds since midnight.

MSEC

Operation	The number of milliseconds since midnight is returned.
Notes	<p>The MSEC function should only be used to compare differences between two times, rather than the current time of day.</p> <p>MSEC actually returns the number of system hardware timer ticks since midnight. The number of system hardware ticks in one second is 996. Significant inaccuracies may occur if this function is used to keep track of long periods of time.</p>
See also	<a href="#">SECOND</a> , <a href="#">TIME\$</a>
Examples	The following routine waits for the operator to type a single character. While waiting, a time of day clock is maintained on the screen.

Statements

```
This is a straight forward method of performing this operation.
1000 GET .SELECTION:
1010
1020     MARK.TIME = MSEC
1030
1040     WHILE SELECTION$=" "
1050
1060         IF MSEC-MARK.TIME>1000
1070             GOSUB DISPLAY.TIME
1080             IFEND
1090
1100             PRINT AT$(40,22); \ GET #0:SELECTION$
1130             WEND
1140
1150     RETURN
~
2000 DISPLAY.TIME:
2010
2020     PRINT AT$(73,1);TIME$(0);
2030     MARK.TIME = MSEC
2040
2050     RETURN
```

# NBR Function

NBR returns a true/false value indicating whether or not a string expression contains a valid representation of a number.

NBR( *string-expression* )

**Operation** *string-expression* is analyzed to determine if it contains only characters that are valid for a number. The characters that are valid for a number include:

- ▶ Leading and trailing spaces
- ▶ One plus or minus sign
- ▶ One period (or comma if [OPTION COMMA](#) is in effect)
- ▶ The letter E (for scientific notation) followed by an optional plus or minus sign
- ▶ The digits 0–9.

When the *string-expression* contains any characters that are not valid for a number or are in the wrong position, a false or zero value is returned. When all of the characters are valid for a number and are in the proper position and sequence, a true or –1 value is returned.

## Examples

*This routine checks the input field for a valid dollar amount.*

```
23000  VALIDATE.AMOUNT :
23010
23020      SELECT
```

*The NBR function is used to verify that the dollar amount is a valid number.*

```
23030      CASE NOT NBR(FIELD$)
23040          VALID% = FALSE%
23050          P2MSG$ = "NUMERIC VALUE REQUIRED"
23060      CASE VAL(FIELD$)<>ROUND(VAL(FIELD$),2)
23070          VALID% = FALSE%
23080          P2MSG$ = "FRACTIONAL PENNIES ARE NOT ALLOWED"
23090      CASE VAL(FIELD$)>=10^(INL%(FLD%)-3)
23100          VALID% = FALSE%
23110          P2MSG$ = "DOLLAR VALUE TOO LARGE"
23120      CEND
23130
23140      RETURN
```

Statements

---

## NEXT Statement

The NEXT statement marks the end and repeat point of a [FOR-NEXT](#) program structure.

```
1 NEXT
2 NEXT variable-name
```

**Operation**      The NEXT statement is part of a program structure that starts with the FOR statement and ends with the NEXT statement. These two statements are separated by zero or more statements that comprise the statements of the loop.

**Mode 1**—The normal mode for this statement. The current [FOR-NEXT](#) structure is repeated.

**Mode 2**—The list of open or active [FOR-NEXT](#) program structures is searched for the most recent one that uses *variable-name* as its control variable. That structure is then repeated.

This second form of the NEXT statement is provided for compatibility with prior versions of BASIC and it should not be used for new code.

Refer to the [FOR](#) statement description for additional information about [FOR-NEXT](#) program structures, restrictions and examples.

**See also**      [BREAK](#), [CONTINUE](#), [FOR](#)

---

# NEXT.FILE Function

The NEXT.FILE function returns the lowest input/output channel number that is not currently in use by an open file.

NEXT.FILE

Operation	The list of input/output channels is scanned for the lowest channel not in use.
Returns	The first or lowest available channel number is returned.
Notes	Input/output channels are numbered from 0–999.  Channel number 0 is always open as the console  This function does not reserve the channel number.
See also	<a href="#">OPEN</a>

Examples	10000	AUTO.ANSWER.ON:	! Set modem answer on
	10010		
	10020	COM% = NEXT.FILE	! Get next channel number
	10030	OPEN #COM%: "COM1", OUTPUT SEQUENTIAL	
	10040	PRINT #COM%: "ATS0=1";CHR(13);	! Set rings to answer=1
	10050	CLOSE #COM%	! Finished with file
	10060		
	10070	RETURN	

Statements

---

## OCT, OCTOF\$ Functions

OCT returns the numeric value of an *octal* character string.

OCTOF\$ returns an octal character string of a numeric expression.

**OCT**( *string-expression* )

**OCTOF\$**( *integer-expression* )

**Operation**      The OCT function analyzes *string-expression* looking for octal digits (0–7). When the first non-octal digit is found, analysis stops and the string of digits is evaluated as a numeric value expressed in base 8.

The OCTOF\$ function evaluates *integer-expression* and generates the six octal digit characters representing that value.

**Notes**            The OCT function is the complement to the OCTOF\$ function.

Only the last six octal digits found are used. For example:

OCT("12345671") ⇔ OCT("345671")

**Restrictions**    OCT returns integer values (–32768 to +32767).

**Examples**        10 INPUT "Enter octal value",OCT.VALUE\$  
20  
30 PRINT OCT.VALUE\$;"0 =";OCT(OCT.VALUE\$)

Enter octal value? 123

1230 = 83

See also the example in the HEX function description.

# ON ERROR Statement

ON ERROR specifies the location of the programmer-defined trappable-error handling routine.

1

ON ERROR GOTO *line-reference*

2

ON ERROR GOTO 0

Operation	<p><b>Mode 1</b>—Enables error trapping by specifying the location of the trappable-error handling routine. The <i>line-reference</i> is saved and when a trappable-error occurs, control is transferred to <i>line-reference</i>.</p> <p><b>Mode 2</b>—Any <i>line-reference</i> saved from a previous execution of an ON ERROR statement is discarded and subsequent trappable-errors will not be trapped by the program.</p>
Notes	<p>The trappable-errors are listed in Appendix D: “<a href="#">Error Codes and Messages</a>,” starting on page <a href="#">640</a>.</p> <p>Normally, when a trappable-error occurs and error trapping is disabled MultiUser BASIC displays its own error message and terminates the execution of the program. When error trapping is enabled (with <a href="#">Mode 1</a>) and a trappable-error occurs, control is transferred to the routine at <i>line-reference</i>. The error handling routine can then use the <a href="#">ERR</a> and <a href="#">ERL</a> functions to determine which error occurred and the program location causing the error. Appropriate action can then be taken.</p> <p>The trappable-error handling routine must be terminated with a <a href="#">RESUME</a> statement, at which point control returns to the statement causing the error. (<a href="#">RESUME</a> does allow the program to resume control at a different location, if desired. Refer to the description of the <a href="#">RESUME</a> statement.)</p>
Defaults	<p>At the beginning of the execution of a program from a <a href="#">CHAIN</a>, <a href="#">LINK</a> or <a href="#">RUN</a>, error trapping is disabled.</p>
Restrictions	<p>The <i>line-reference</i> must exist in this program. Like all event service routines, <i>line-reference</i> must be part of the main program code. That is, it may not be defined inside of a subprogram (between <a href="#">SUB</a> and <a href="#">END SUB</a> statements) or inside a user-defined function (between <a href="#">DEF FN</a> and <a href="#">FNEND</a> statements).</p>

Statements

**Examples**

*This routine is about to open a special file that may not exist. A trap is specified just for this operation. If the operation is successful the normal error trap is respecified.*

```
602000 CHECK.DATEBOOK:
602010
602020     ON ERROR GOTO CALENDAR.OPEN.TRAP \ REM No file trap
602030
602040     FILENAME$ = "SYSTEM.DATEBOOK."&OPNAME$
602050     OPEN #14: FILENAME$, INPUT INDEXED
602060
602070     ON ERROR GOTO ERROR.TRAP \ REM Normal error trapping
~
```

*The normal exit from the subroutine is given a line label that can be referred to by the open error trap.*

```
602620 CHECK.DATEBOOK.RETURN:
602630
602640     ON ERROR GOTO ERROR.TRAP \ REM Reset error trap
602650
602660     RETURN
~
```

*CALENDAR.OPEN.TRAP is only executed when there is an error during the OPEN statement. In this case, if the file cannot be found then the entire operation is ignored by RESUMING to the subroutine exit point.*

```
602840 CALENDAR.OPEN.TRAP:
602850
602860 REM If the calendar data file cannot be found,
        then ignore calendar
602870
602880     RESUME CHECK.DATEBOOK.RETURN
~
```

*The normal trappable error handling routine for this program only checks for Break,C errors (ERR=1). If that occurs the routine confirms with the operator that that is what is desired and either exits the program through the program termination routine, or continues processing with the statement that was executing when the Break,C was entered.*

```
890000 ERROR.TRAP:
890010
890020     IF ERR=1 \ REM Break,C?
890030         PRINT CRT$("KOFF");CRT$("BELL");
890040         ANS$ = FN.PROMPT.YESNO$(2,"DO YOU WISH TO EXIT?",
            2, "PRESS 'Y' TO CONFIRM")
890050         IF ANS$="Y" THEN RESUME END.RUN \ REM Exit program
890060         PRINT FN.PROMPT$(0,"",0,"");
890070         RESUME \ REM Continue normal execution
```

*If the error was not a Break,C this routine exits to the general error trapping program.*

```
890080     ELSE ERR.NBR% = ERR \ ERR.LIN = ERL
890090         LINK "ERRORTRP" \ REM Branch to error program
890100         IFEND
```



---

## ON EVENT Statement

The ON EVENT statement specifies the location of the programmer-defined event handling routine.

- 1 **ON EVENT** ( *semaphore* ) **GOTO** *line-reference*
  - 2 **ON EVENT** ( *semaphore* ) **GOTO** 0
  - 3 **ON EVENT** **ON** | **OFF**

These statements do not perform any operations themselves; they merely set and save the conditions for event detection.

### Operation

An event is the setting of a semaphore to the **on** status. Normally, this is done by another task using the **SET** function. It may also be done by this program with the **SET** function, or the **TIMER** statement.

**Mode 1**—Enables event trapping by specifying that, when event *semaphore* occurs, the event handler at *line-reference* is to be invoked asynchronously. That is, control will be temporarily transferred to *line-reference* when the event occurs, no matter what statement is being executed at the time. That event handler performs whatever tasks are associated with the event and then returns control to the interrupted program with the **RESUME** statement.

**Mode 2**—Disables the event handler for a specific event *semaphore*.

**Mode 3**—Temporarily suspends event trapping (**OFF**), or resumes event trapping (**ON**).

### Notes

An event occurs when *semaphore* is set on. Setting *semaphore* on for a semaphore that has a **Mode 1** event trap set, causes the indicated routine to be invoked. Prior to entry to that routine, the *semaphore* is reset.

The invoking of an event handler is not truly asynchronous. A program that has an ON EVENT statement executes slightly differently than a program without any ON EVENT statement. With an ON EVENT statement, MultiUser BASIC tests for events prior to executing each statement. If an event occurs during the execution of a statement (such as an **INPUT** or **LINPUT** statement) the event is not detected, and the event handler is not invoked until the current statement finishes its execution.

The `WAIT #0` statement will be interrupted when an event occurs. When the event handler's `RESUME` statement is executed, control returns to the statement following the `WAIT #0` statement.

The `WAIT EVENT` statement is also interrupted when an event occurs. For example, a program has executed an `ON EVENT` statement for semaphore A and the program is currently performing a `WAIT EVENT` for semaphore B. When semaphore A is set by an external task, the event handler will be invoked for semaphore A. Execution of the event handler's `RESUME` statement will return control to the `WAIT EVENT` statement.

When a semaphore is first catalogued (with the `SEMAPHORE` function), it should be cleared. The `SEMAPHORE` function does not clear the status of a semaphore and it is possible that the status of the semaphore might be set. Executing an `ON EVENT` trap for a semaphore that is currently set causes the trap routine to be invoked immediately.

In a multitask environment where several tasks have specified an `ON EVENT` trap for a specific semaphore, only one of the tasks will service the event. When the event occurs (semaphore set) the next task to begin execution of a statement will receive and service the event. Before control is transferred to the event trap location, the semaphore is reset.

When an event service routine begins execution, the status of the semaphore is off, or reset. The service routine can explicitly set the semaphore on again.

The `ON EVENT OFF` statement suspends all event service. It does not discard or clear the status of any semaphores. If, while `ON EVENT OFF` is in effect, an event is set on, it will become a pending event. When event servicing is resumed (with `ON EVENT ON`), any and all pending events are serviced.

#### Restrictions

The range of valid values for *semaphore* is 0–63. Specifying a *semaphore* value outside this range is not detected as an error, however, no event will ever occur and thus, the event trap will never be invoked.

The *line-reference* must exist in the program. Like all event service routines, *line-reference* must be part of the main program code. That is, it may not be defined inside of a subprogram (between `SUB` and `END SUB` statements) or inside a user-defined function (between `DEF FN` and `FNEND` statements).

#### See also

`EVENT`, `ON KEY`, `RESET`, `RESUME`, `SEMAPHORE`, `SET`, `SYS.ENV$`, `TIMER`, `WAIT EVENT`

## Examples

<i>Use a single-line window for heading.</i>	1000	WINDOW OPEN 1, 1,1,80,1; SELECT UPDATE ON
	1010	PRINT AT\$(1,1);"Sample Program";
	1020	
<i>Remainder of screen is normal program area.</i>	1030	WINDOW OPEN 2, 1,2, 80,23; SELECT UPDATE ON
	1040	
<i>Define a time of day semaphore for displaying the clock.</i>	1050	TIME.OF.DAY% = SEMAPHORE("Timer")
	1060	
	1070	ON EVENT(TIME.OF.DAY%) GOTO DISPLAY.TIME
	1080	
<i>The SET function invokes the time-of-day display routine now. Program the next display for a change in the minute value of the clock.</i>	1090	X% = SET(TIME.OF.DAY%) ! Display first time now
	1100	
	1110	TIMER TIME.OF.DAY% SYNC 60 ! Display time on the minute
	1120	
<i>This GET.INPUT routine is just a filler for illustration purposes.</i>	2000	GET.INPUT:
	2010	
	2020	PRINT AT\$(1,5);CRT\$("EOL");
	2030	
	2040	LINPUT "Input",TEXT\$
	2050	
	2060	GOTO GET.INPUT
	2070	
<i>The DISPLAY.TIME routine saves the currently selected window and selects the heading window for its own purposes. The current time-of-day is displayed in hh:mm and then the prior active window is reselected.</i>	10000	DISPLAY.TIME:
	10010	
	10020	WINDOW STATUS PRIOR.WINDOW% ! Save current window
	10030	WINDOW SELECT 1, UPDATE ON
	10040	
	10050	PRINT AT\$(75,1);LEFT\$(TIME\$(0),5);
	10060	
	10070	WINDOW REFRESH 1
	10080	WINDOW SELECT PRIOR.WINDOW%, UPDATE ON
	10090	
<i>The time-of-day event handler is reprogrammed because it was disabled when the current event occurred.</i>	10100	TIMER TIME.OF.DAY% SYNC 60
	10110	
	10120	RESUME

---

## ON GOSUB Statement

ON GOSUB calls one of several subroutines depending upon the value of an expression.

**ON** *numeric-expression* **GOSUB** *line-reference-list*

---

*line-reference-list* » *line-reference* [, *line-reference-list*]

**Operation**      The *numeric-expression* is evaluated and the integer portion of the result is used to index into the *line-reference-list*. A **GOSUB** is then performed to the indexed line reference. For example, if SEL% equals four, the statement:

```
ON SEL% GOSUB ROUTINE.A,ROUTINE.B, ,ROUTINE.D,ROUTINE.E
```

performs a **GOSUB** to ROUTINE.D.

**Notes**            The value of *numeric-expression* may be less than one or greater than the number of *line-references*. When this is the case, none of the line-references are invoked and the next statement executed is the one following the ON GOSUB statement.

Line-references may be omitted from the list (as in the above example, between references ROUTINE.B and ROUTINE.D). When the value of the *numeric-expression* corresponds to that line reference position, no subroutine is invoked and the next statement executed is the one following the ON GOSUB statement.

The ON GOSUB statement operates similar to a **SELECT-CEND** programming structure (refer to following example). The **SELECT-CEND** is the recommended programming technique because it is easier to see the control flow and it is more easily maintained. The ON GOSUB, on the other hand, is more compact and is faster to execute.

For additional information about subroutines refer to the description of the **GOSUB** and **RETURN** statements.

**Restrictions**    Each of the line-references in *line-reference-list* must exist in the program.

**See also**        **GOSUB**, **ON GOTO**, **RETURN**

Examples

*These two routines  
perform the exact  
same operation.*

2010  
2020  
2030  
2010

ON ITEM.SELECT% GOSUB READ.NEXT,READ.PREV,READ.IT

*The latter routine  
using the **SELECT-  
CEND** structure is  
a better program-  
ming technique.*

2020  
2030  
2040  
2050  
2060  
2070  
2080  
2090

SELECT ITEM.SELECT%  
CASE 1  
GOSUB READ.NEXT  
CASE 2  
GOSUB READ.PREV  
CASE 3  
GOSUB READ.IT  
CEND

# ON GOTO Statement

ON GOTO branches to one of several locations depending upon the value of an expression.

**ON** *numeric-expression* **GOTO** *line-reference-list*

*line-reference-list* » *line-reference* [, *line-reference-list*]

**Operation** The *numeric-expression* is evaluated and the integer portion of the result is used to index into the *line-reference-list*. A **GOTO** is then performed to the indexed line reference. For example, if SEL% equals 3 then the statement:

ON SEL% GOTO ROUTINE.A, ,ROUTINE.C,ROUTINE.D,ROUTINE.E

performs a **GOTO** to ROUTINE.C.

**Notes** The value of *numeric-expression* may be less than 1 or greater than the number of *line-references*. When this is the case, none of the *line-references* are invoked and the next statement executed is the one following the ON GOTO statement.

*Line-references* may be omitted from the list (as in the above example, between references ROUTINE.A and ROUTINE.C). When the value of the *numeric-expression* corresponds to that line reference position, no branch is performed and the next statement executed is the one following the ON GOTO statement.

The ON GOTO statement operates similar to a **SELECT-CEND** programming structure (refer to following example). The **SELECT-CEND** is the recommended programming technique because it is easier to see the control flow and it is more easily maintained. The ON GOTO, on the other hand, is more compact and is faster to execute.

**Restrictions** Each of the line-references in *line-reference-list* must exist.

Refer to the **GOTO** statement for special considerations.

**See also** **GOSUB**, **GOTO**, **ON GOSUB**

**Examples** 2020 ON ITEM.SELECT% GOTO READ.NEXT,READ.PREV,READ.IT

# ON KEY Statement

ON KEY specifies the location of the programmer-defined routine for handling special key events.

- 1

ON KEY ( *key-token* ) GOTO *line-reference*
- 2

ON KEY ( *key-token* ) GOTO 0
- 3

ON KEY ON | OFF

**Operation**      **Mode 1**—An event trap is enabled for *key-token*. Whenever that key or key combination is pressed during program execution, the routine specified by *line-reference* is invoked as an asynchronous event trap. That is, the normal execution of your program is interrupted and the statements starting at *line-reference* are executed until a RESUME statement is encountered.

Upon entry to the event routine the value of ON.KEY.TOKEN is set to the *key-token* value.

The RESUME statement causes normal execution of the program to continue.

**Mode 2**—The event trap defined for *key-token* is disabled.

**Mode 3**—Event trapping for all keys is *suspended* or *resumed*. Initially, ON KEY trapping is in the resumed status.

**Notes**      Only one event trap may be defined for a specific key or key combination at any one time. When a second ON KEY is specified for the same key the first one is disabled.

The *key-token* for alphabetic keys may be specified with any of three codes: the ASCII value of the uppercase letter (values 65–90 such as 70 for F), the ASCII value of the lowercase letter (values 97–122 such as 98 for b) or the ordinal value of the letter (values 1–26 such as 3 for C). Thus, the *key-token* for the letter key K may be specified as 11, 75 or 107. The specification may be made by using the ASC function. For example:

ON KEY (ASC("a")) GOTO TRAP.A

An event trap may be specified for two or three of the key values available for a single key. For example, a trap on key 1, a trap on key 65 and a trap





Statements

on key 97. However, only the last trap specified for a particular key combination is effective and any earlier ones are discarded. When that last trap is disabled (by using [Mode 2](#) of the statement), the prior trap is not enabled. For example:

```

1000  ON KEY(1) GOTO KEY.A
1010  ON KEY(65) GOTO KEY.A
1020  ON KEY(97) GOTO KEY.A
1030  ON KEY(1+4096) GOTO KEY.SHIFT.A
~
10000KEY.A:
10010
~
10090  ON KEY(ON.KEY.TOKEN) GOTO 0






























```

In the above code, the  key, when pressed, causes a trap and the [ON.KEY.TOKEN](#) is set to 97 (see description of [ON.KEY.TOKEN](#) below and in the function descriptions). When line 10090 is executed, that trap is disabled and pressing the  key will not be trapped. The key combination + will still be trapped.

Care should be taken when writing an ON KEY service routine that handles more than one key value. While the routine is servicing one key value it is possible that the operator may press another key that reinvokes the same routine. This condition requires the programmer to include all of the concepts of reentrant programming. The simplest solution is to suspend ON KEY trapping during the service routine (ON KEY OFF).




If it is necessary to allow reentrant execution of the service routine, then make sure that no global variables are modified by the routine. (Use a user-defined function to gain the capabilities of local variables.)




The *key-token* for function keys, the escape key, and the special editing keys is coded according to the following table:


Key	Code	Key	Code	Key	Code	Key	Code
	256		261		269		277
	257		262		270		278
	258		263		271		279
	259		264		272		280
	260		265		273		281
			266		274		282
			267		275		283
			268		276		284

Note that keyboards with the “Windows” and “Menu” keys, these keys can be detected with ON KEY. The left Windows key is mapped to , the right Windows key is  and the Menu key is .






In addition to these unmodified keys, the following modifiers can be added to the key code value to specify key combinations. For example, you can specify an event trap when the key combination ++ is pressed. This would be done by specifying the value 20749 (4096+16384+269).

Key modifier	Code	Hex Value
	4,096	1000H
	8,192	2000H
	16,384	4000H

When a key event is trapped the value of the `ON.KEY.TOKEN` function is set to the *key-token* value specified for the key. For example, if a trap for  was specified with a *key-token* of 1 then the `ON.KEY.TOKEN` value is set to 1, not 65 or 97.

The `ON.KEY.TOKEN` is similar in concept to the `INP` function: when any trapped key or key combination is pressed the `ON.KEY.TOKEN` is set to the value of the key or key combination. This value can be tested in the event-trap routine to determine which key or key combination was pressed. With this ability, a single routine can be written to handle several key events with the specific handling determined by the `ON.KEY.TOKEN` value.

When a trapped key is pressed while a program is executing an `INPUT`, `LINPUT`, `LINPUT USING` or `WAIT` statement, input is terminated as if  were pressed and the trap routine is invoked. (The program can determine whether input was terminated with an ON KEY trap or an actual entry of  by checking the `ON.KEY.TOKEN` value. Actual entry of  causes the `ON.KEY.TOKEN` to be set to zero.)

The current status of ON KEY trapping may be found with the `SYS.ENV$` function. That function can also set the mode.

## Restrictions

ON KEY trapping is available on the main console, THEOS WorkStation consoles and most terminals. Use the CRT command, ON KEY test to confirm that ON KEY trapping is effective on the consoles in the environment that your programs will be used.

Alphabetic keys, function keys, editing keys and the escape key are the only keys that may be trapped with this statement. All other key code values cause an error 59 to occur ("Invalid ON KEY token.")

When an event handling routine is invoked by a specific key, trapping for that key is suspended until the RESUME statement is executed or another ON KEY is executed for the same key.

The key combinations **Ctrl**+**Alt**+**Del** and **⇧ Shift**+**Ctrl**+**Alt**+**Del** cannot be trapped by this process.

Like all event service routines, *line-reference* must be part of the main program code. That is, it may not be defined inside of a subprogram (between **SUB** and **END SUB** statements) or inside a user-defined function (between **DEF FN** and **FNEND** statements).

In a multitasking program, only one of the tasks may have an active ON KEY trap defined for a specific key code. When a copy of the current program is started as a subtask of the current program (Mode 1 of the **ACTIVATE** statement) the started subtask will have all ON KEY event trapping disabled.

There is a limit of 256 simultaneous ON KEYs that may be in effect at one time.

**See also**      [ON EVENT](#), [ON.KEY.TOKEN](#), [RESUME](#), [SYS.ENVS](#)

## Examples

The following example code is a fragment of a program. The setup routine is performed first (line 900000), defining various constants and event traps. At line 901000 a trap is set for the key **F9**. When that key is pressed in the future, the QUIT.KEY routine at line 700000 is invoked.

<i>This routine is invoked when the operator presses <b>F9</b>.</i>	700000 QUIT.KEY:
	700010
	700020 WINDOW STATUS PRIOR.TO.QUIT%
	700030
<i>A new window is opened and the operator is asked if they are certain.</i>	700040 WINDOW OPEN 8,23,10,35,3; FRAME DOUBLE, RIGHT,
	COLOR WHITE%,RED%; COLOR WHITE%,RED%; SELECT
	700050
	700060 PRINT AT\$(2,2);"Do you really wish to QUIT? ";
	700070 REPLY\$ = YESNO\$
	700080
<i>Before exiting, the screen is cleaned up by selecting the prior window, then closing and removing this quit window.</i>	700090 WINDOW SELECT PRIOR.TO.QUIT%, UPDATE ON
	700100 WINDOW CLOSE 8, REMOVE
	700120
	700130 IF REPLY\$="N"
	700140 RESUME
	700150 ELSE
	700160 IF ON.KEY.TOKEN=F9%
	700170 RESUME END.PROGRAM
	700180 ELSE RESUME END.APPLICATION
	700190 IFEND
	700200 IFEND
	~
	900000 SETUP:
	900010
	900020 ESC% = 256
	900030 TAB.KEY% = 265
	~
	900070 F1% = 269 \ F2% = 270 \ F3% = 271 \ F4% = 272
	900080 F5% = 273 \ F6% = 274 \ F7% = 275 \ F8% = 276
	900090 F9% = 277 \ F10% = 278 \ F11% = 279 \ F12% = 280
	900100 ALT% = 4000H
	900110 CTRL% = 2000H
	900120 SHIFT% = 1000H
	~
	901000 ON KEY(F9%) GOTO QUIT.KEY
<i>Define the keys that will invoke the quit window.</i>	901010 ON KEY(SHIFT%+F9%) GOTO QUIT.KEY
	~
	902000 ON ERROR GOTO ERROR.TRAP
	902010
	902020 RETURN

---

# ON.KEY.TOKEN Function

ON.KEY.TOKEN returns the key or key combination value used to invoke an ON KEY routine.

ON.KEY.TOKEN

Operation	The key or key combination value that invoked an event trap set by an <a href="#">ON KEY</a> statement is returned.
Notes	<p>The values returned by this function correspond to the various key and key combinations that may be trapped by the <a href="#">ON KEY</a> statement.</p> <p>The <a href="#">ON KEY</a> statement allows the alphabetic keys to be specified with one of three values for each key. For example, the key <b>A</b> may be specified with a 1 (ordinal number), 65 (uppercase value) or 97 (lowercase value). When the key is trapped the ON.KEY.TOKEN is set to the same value specified with the <a href="#">ON KEY</a> statement.</p> <p>For example, when <b>A</b> key is trapped and the specification was ON KEY(1) the ON.KEY.TOKEN is set to 1; if that same key were specified with ON KEY(65) GOTO then the ON.KEY.TOKEN is set to 65.</p> <p>The ON.KEY.TOKEN is automatically cleared (set to 0) prior to the execution of an <a href="#">INPUT</a> or <a href="#">LINPUT</a> statement. If a trapped key is pressed during the execution of these statements, the input is terminated as if <b>Enter</b> were pressed, the ON.KEY.TOKEN is set, and the trap routine is invoked. When the trap routine's <a href="#">RESUME</a> statement is executed, control returns to the statement following the terminated <a href="#">INPUT</a> or <a href="#">LINPUT</a>. The ON.KEY.TOKEN function will have the value of the trapped key. An ON.KEY.TOKEN value of zero indicates that the input was terminated normally (by actually pressing <b>Enter</b> or a control key in position one).</p>
Restrictions	ON KEY trapping is available on the main console, THEOS WorkStation consoles and most terminals. Use the CRT command, ON KEY test to confirm that ON KEY trapping is effective on the consoles in the environment that your programs will be used.
See also	<a href="#">INP</a> , <a href="#">ON KEY</a>
Examples	An example usage of the ON.KEY.TOKEN is given in the description of the <a href="#">ON KEY</a> statement.

# ON MOUSE Statements

The various forms of the ON MOUSE statement specify the location of the programmer-defined routine for handling various mouse-related events.

1 ON MOUSE GOTO *line-reference*

2 ON MOUSE GOTO 0

3 ON MOUSE ON | OFF

4 ON *button* [*action*] GOTO *line-reference*

5 ON *button* [*action*] GOTO 0

6 ON *button* [*action*] ON | OFF

*button*

»

LEFT  
RIGHT  
CENTER

*action*

»

CLICK  
DCLICK

These statements do not perform any operations themselves; they merely set and save the conditions for mouse detection.

## Operation

Mouse events can be detected and trapped with a general purpose mouse handling routine or with specific routines for specific mouse actions. The ON MOUSE statement is used to specify detection and trapping for any mouse event. The ON LEFT, ON RIGHT and ON CENTER statements are used to specify detection and trapping for specific mouse events.

**Mode 1**—An event trap is enabled for any mouse action. When any mouse button is clicked, double-clicked or dragged the event handler specified here is invoked (note that [Mode 4](#) type specific events take precedence).

**Mode 2**—The general mouse event handler is disabled. Specific mouse event handlers enabled with ON LEFT, ON RIGHT or ON CENTER statements are not affected.

**Mode 3**—All mouse event detection is *suspended* (OFF) or *resumed* (ON). Although specific mouse event handlers are also suspended and resumed

Statements

with this statement, any specific mouse event handler that was also suspended with its own `ON button action OFF` statement is not resumed.

**Mode 4**—An event trap is *enabled* for a specific mouse action. If *action* is not specified then `CLICK` is assumed.

When the event occurs the event handler specified here is invoked unless the event detection is suspended ([Mode 6](#)) or all mouse events are suspended ([Mode 3](#)).

**Mode 5**—The specific mouse event handler is *disabled*. If *action* is not specified then `CLICK` is assumed.

**Mode 6**—The specific mouse event is *suspended* (OFF) or *resumed* (ON). If *action* is not specified then `CLICK` is assumed.

## Notes

A mouse event refers to a mouse button click, double click or drag, not mouse movement.

Mouse event trapping is performed by the same mechanism used for other event trapping. If multiple event types occur they are processed with the following order of precedence:

- ▶ `ON ERROR` events
- ▶ `ON EVENT` events
- ▶ `ON KEY` events
- ▶ Mouse events
  - ▶ `ON button action` events
  - ▶ `ON MOUSE` events
- ▶ `ON TIMEOUT` events

The exact timing used for differentiating a *click* action from a *double click* action is defined by the device driver. When both single clicks and double clicks are being tracked the driver waits through a “guard time” to determine if a second click is being entered and then either reports a single click or a double click, as appropriate.

When a mouse event occurs and is trapped the specific event that occurred is suspended from further trapping until the mouse event handler is exited with the `RESUME` statement. At this time the values for `MOUSE.BUTTON`, `MOUSE.COL`, `MOUSE.FRAME`, `MOUSE.ROW` and `MOUSE.WINDOW`, are set to the appropriate values.

When multiple types of mouse events are enabled the information about the event and the mouse cursor is only guaranteed to be accurate at the

time the first statement is executed in the mouse event handler routine. This is due to the fact that only the specific mouse event type is suspended while in the event handler. Mouse events are not automatically disabled or suspended when the event handler is invoked and another mouse event might occur during the processing of the first mouse event.

If this possibility might cause your program to behave undesirably, the first statement in the mouse event handler should suspend further mouse events. Mouse events can be suspended with either [Mode 3](#) or [Mode 6](#) of the ON MOUSE statements:

```
ON MOUSE OFF
```

or

```
ON button action OFF
```

Be sure to resume mouse event trapping with the corresponding statement before exiting the mouse event handler.

```
ON MOUSE ON
```

or




```
ON button action ON
```







## Defaults

The initial status of mouse event detection is disabled but resumed, just as if your program had executed the statements:

```
ON MOUSE GOTO 0
ON MOUSE ON
```

For [Mode 4](#), [Mode 5](#) and [Mode 6](#), when *action* is not specified then CLICK is the default action referred to.

When no mouse trapping is being performed it is still possible for the operator to use the mouse during execution of [INPUT](#), [LINPUT](#) and [LINPUT USING](#) statements. In this situation the mouse can be used for two functions: relative motion of the text cursor (simulates pressing the  and  keys) and input termination (simulates pressing .

Dragging the mouse to the left or to the right with the left mouse button pressed generates  and  keys codes and causes the text cursor to move left or right, as allowed by the input statement used. Dragging the mouse up or down generates  and  keys codes and causes input termination. Double clicking the left mouse button generates an  key code and terminates input. Double clicking the right mouse button terminates input and set the INP to 31, just as if the  key were pressed.

## Restrictions

The mouse event handler routine must be located in the main program area, not between `DEF FN-FNEND` statements nor between `SUB-END SUB` statements. (Include files are part of the main program area unless they are included in a function definition or a subprogram.)

In a multiple-task environment, only one task may have mouse event trapping enabled at any one time. When a copy of the current program is started as a subtask of the current program (Mode 1 of the `ACTIVATE` statement) the started subtask will have all mouse event trapping disabled.

Mouse event trapping is always disabled at the start of a program's execution. This is true even when the program was executed as the result of a `CHAIN` or `LINK` from a program that enabled mouse event trapping.

## See also

`MOUSE.BUTTON`, `MOUSE.COL`, `MOUSE.FRAME`, `MOUSE.ROW`, `MOUSE.WINDOW`

## Examples

The following code is a portion of the sample include file named `MOUSE`. This is a set of routines that allow the definition and display of mouse buttons on the screen in a windowed environment. Not shown here are the routines that defined and displayed the mouse buttons. These definitions are saved in a set of arrays describing the location of the upper left and lower right corners of the buttons along with other important information about the button position and attributes.

```
1100    ON LEFT CLICK GOTO MOUSE.CLICK    ! Trap left mouse clicks
1110    ON RIGHT CLICK GOTO MOUSE.CLICK   ! Trap right mouse clicks
...
4000  MOUSE.CLICK: ! Mouse event handler for mouse button events
4010
4020    ON MOUSE OFF                      ! Suspend mouse events
4030
4040    BUTTON.TYPE% = MOUSE.BUTTON       ! Save type of mouse action
4050    BUTTON% = 0                       ! Initially, no match
4060
4070    FOR MI%=1 TO 10                   ! For each defined "button"
4080
4090        IF MOUSE.WINDOW=BUTTON.WIN%(MI%) ! Possible candidate?
4100        IF BUTTON.STATUS%(MI%)          ! Button enabled?
4110        IF MOUSE.COL>BUTTON.X%(MI%) AND
            MOUSE.COL<BUTTON.TO.X%(MI%)!
4120        IF MOUSE.ROW>=BUTTON.Y%(MI%) AND
            MOUSE.ROW<=BUTTON.TO.Y%(MI%)
4130        BUTTON% = MI%                 ! We have a match!
4140        BREAK ! Exit FOR-NEXT
4150        IFEND
4160    IFEND
4170    IFEND
4180    IFEND
4190    NEXT
4200
4210    ON MOUSE ON                        ! Resume mouse trapping
4220
4230    RESUME                            ! Return to program
```



## ON TIMEOUT Statement



The ON TIMEOUT statement provides the ability to detect and handle the situation when no characters are received from the keyboard after a specified amount of time has elapsed.

- 1 **ON TIMEOUT**( *time* ) **GOTO** *line-reference*
- 2 **ON TIMEOUT**( *time* ) **GOTO** 0
- 3 **ON TIMEOUT OFF** | **ON**

This statement does not perform any operations itself; it merely sets and saves the conditions for operator time-out detection that will be performed on subsequent [INPUT](#), [LINPUT](#), [LINPUT USING](#), [MAT INPUT](#), [WINDOW CHOICE](#), [WINDOW EDIT](#) and [WAIT](#) statements and the [ERRMSG\\$](#) and [YESNO\\$](#) functions.

### Operation

**Mode 1**—The value of *time* and the location of the time-out event handler is saved, replacing any previously set time and location. This statement *enables* time-out detection.

**Mode 2**—The time-out time and location is cleared, *disabling* time-out detection and processing. The *time* value is required but not used.

**Mode 3**—Time-out detection is *suspended* (OFF) or *resumed* (ON). The time-out time and the location of the event handler is not changed.

### Notes

The value for *time* is expressed in seconds and must be in the range of 1–4,294,000. The value of *time* is used to initialize a count-down timer used during keyboard input statements and functions.

Some common values for *time*:

Length	time	Length	time
1 minute	60	10 minutes	600
2 minutes	120	15 minutes	900
3 minutes	180	20 minutes	1,200
4 minutes	240	30 minutes	1,800
5 minutes	300	1 hour	3,600

When time-outs are enabled and a keyboard input statement or function is being executed, the count-down timer is reset at the start of the statement or function and it is reset after every character is received.

Mouse activity does not reset the count-down timer. However, when mouse event trapping is enabled, a mouse event causes termination of an input statement or function, thus terminating timed-event processing.

When the timer counts down to zero the keyboard input statement or function is exited. Any characters entered prior to the time-out event are retained as part of the data input.

The time-out event handler is programmed like other event handlers used for [ON EVENT](#), [ON KEY](#), [ON MOUSE](#), *etc.* and requires a [RESUME](#) statement to exit the routine.

In a multitasking application, each task may have its own time-out routine enabled with its own time limit. Unlike other event processes, time-out events are not invoked by a shared resource like a mouse, keyboard or semaphore but are invoked by the absence of an action.

#### Defaults

The initial status of time-out detection is disabled but resumed, just as if your program had executed the statements:

```
ON TIMEOUT GOTO 0
ON TIMEOUT ON
```

#### Restrictions

No time-out detection is performed unless one of the following statements or functions is executed and the input is from the keyboard: [ERRMSG\\$](#), [INPUT](#), [LINPUT](#), [LINPUT USING](#), [MAT INPUT](#), [WINDOW CHOICE](#), [WINDOW EDIT](#), [WAIT](#) or [YESNO\\$](#).

Mouse events can affect time-out processing because they cause a carriage return to be processed as if it were generated by the keyboard.

Like all event service routines, *line-reference* must be part of the main program code. That is, it may not be defined inside of a subprogram (between [SUB](#) and [END SUB](#) statements) or inside a user-defined function (between [DEF FN](#) and [FNEND](#) statements).

When a time-out event occurs the time-out event service routine is invoked. However, unlike other event service routines, the time-outs are not disabled while processing the time-out event service routine. Therefore, do not use any input statements or functions in the service routine unless you disable time-outs or define a new time-out service routine.

#### See also

[ON EVENT](#), [RESUME](#)

## Examples

```
3010      ON TIMEOUT (300) GOTO NO.RESPONSE
3020
3030      SELECTION$ = "" \ TIMEOUTS% = 0    ! Initialize
3030      WHILE SELECTION$ = ""
3040          LINPUT USING "!", SELECTION$
...
10000 NO.RESPONSE:
10010     TIMEOUTS% = TIMEOUTS%+1           ! Count the time-outs
10020     IF TIMEOUTS%>3                     ! Allow only three time-
        outs
10030         LINK "MAIN"                   ! Exit to main program
10040     ELSE PRINT AT(1,24); "MAKE SELECTION"; CRT$("BELL");
10050     IFEND
10060     RESUME
```

# OPEN Statement

OPEN defines the file and access method associated with an input/output file channel.

- 1 **OPEN** *#channel: device-name, mode, SEQUENTIAL* [, *options*]
- 2 **OPEN** *#channel: file-desc, mode, method* [, *options*]
- 3 **OPEN** *#channel: spooler-specs, OUTPUT, SEQUENTIAL* [, *options*]
- 4 **OPEN** *#channel: VDI-device-name, OUTPUT, SEQUENTIAL*

<i>mode</i>	»	<u>INPUT</u> <u>OUTPUT</u> <u>UPDATE</u>
<i>method</i>	»	<u>DIRECT</u> <u>INDEXED</u> <u>KEYED</u> <u>SEQUENTIAL</u>
<i>options</i>	»	<u>EXTEND</u> <u>FORMAT</u> <u>LOCK</u> <u>MULTILOCK</u> <u>QUOTE</u>
<i>spooler-specs</i>	»	<i>report-name</i> [. <i>q</i> [ <i>c</i> [ <i>hold</i> ]]]: <b>PRT</b> [ <i>n</i> ]
<i>q</i>	»	queue letter (A—Z, a—z, \$, ?, %, &, *, +, -, <, =, >, ^, ~)
<i>c</i>	»	copies (1—255)
<i>hold</i>	»	<b>H</b> <b>N</b>

**Operation**      **Mode 1**—Used to open devices for input or output. For example: the console (CON), printer (PRT), communications ports (COM1), or tapes (TAP1). Only the **FORMAT** and **QUOTE** options are valid for this mode of the OPEN statement.

The standard input device (stdin) or the standard output device (stdout) cannot be opened with this statement. To access stdin or stdout merely refer to channel 0 in the various forms of the **INPUT** and **PRINT** or the **GET** and **PUT** statements.

**Mode 2**—This is the most common form of the OPEN statement and is used to open disk files for access.

**Mode 3**—Used to open a spooled printer file. Only the **FORMAT**, **LOCK** and **QUOTE** options are valid for this mode of the OPEN statement. If the printer indicated (PRT $nn$ ) is not accessed via the spooler, the spooler-specs are ignored.

**Mode 4**—Used to open an attached VDI device. No options are valid for this mode.

## Notes

The OPEN statement is a complex statement with several objects in the statement.

Object	Description
<i>channel</i>	An integer expression whose value is in the range 1—1000. This <i>channel</i> is used in all subsequent statements to refer to this open file or device.
<i>device-name</i>	Specifies the byte-oriented logical device name of the device used for input or output <ul style="list-style-type: none"> <li>CON Specifies the console terminal.</li> <li>CONO Specifies the console display only.</li> <li>CONI Specifies the console keyboard only.</li> <li>PRT<math>nn</math> Specifies one of the attached printers.</li> <li>COM<math>n</math> Specifies one of the attached communications ports.</li> <li>TAP<math>n</math> Specifies one of the attached tape drives.</li> </ul>
<i>file-desc</i>	Specifies the name of the disk or tape file that is to be accessed. The <i>file-desc</i> may be a complete and explicit specification of the file, including the owning account name, path name(s), and fn, ft, mn, and fd. For example: <pre>"ACCOUNT\MAIN\SUBDIR\DATA.LIBRARY.FILE:A"</pre> <p>The various portions of this file description may be omitted and the system will use defaults for the missing components:</p> <ul style="list-style-type: none"> <li>Omit <i>account</i> Use current account.</li> <li>Omit <i>path</i> Use current working directory (cwd).</li> <li>Omit <i>fn.ft</i> Use default library if defined; otherwise error.</li> <li>Omit <i>fd</i> Use current working directory's fd. When no cwd, use default search sequence, as defined for account environment.</li> </ul>

Object	Description
<i>mode</i>	<p>A keyword specifying the direction of access to the file/device.</p> <p><u>INPUT</u> Data will be read only from the file/device. Cannot use PRINT or WRITE to the file. Record locks are not performed on INPUT only files.</p> <p><u>OUTPUT</u> Data will be “write only” to the file/device. Cannot use <u>INPUT</u> or READ from the file.</p> <p><u>UPDATE</u> Data will be read from and written to the file/device. Record locking is performed on each record read. Record locks are removed by reading a different record, writing to the file, using the <u>UNLOCK</u> statement, or closing the file.</p> <p>When used with disk files and <u>SEQUENTIAL</u> method, this mode is equivalent to <u>OUTPUT</u> mode.</p>
<i>method</i>	<p>A keyword specifying the organization method of the file/device. Indicates how the records in the file are accessed.</p> <p><u>SEQUENTIAL</u> Indicates that record or bytes/characters are accessed sequentially, one after another. To read a character all previous characters must be read. May be used with files or devices.</p> <p><u>DIRECT</u> Indicates that records are accessed randomly, by a <u>numeric</u> record key. Applies to disk files only.</p> <p><u>INDEXED</u> Indicates that records are accessed randomly or sequentially, by an <u>alphanumeric</u> record key. When accessing the records sequentially (READNEXT, READPREV), the sequence is the sorted order of the keys. Applies to disk files only.</p> <p><u>KEYED</u> Indicates that records are accessed randomly or sequentially, by an <u>alphanumeric</u> record key. When accessing the records sequentially (READNEXT, READPREV), the sequence is the “hashed” order of the keys in the file. Applies to disk files only.</p>

Object	Description
<i>options</i>	Specifies options used during access to the file/device:
<u>FORMAT</u>	Indicates that each record PRINTed to the file will contain a leading ANSI forms control character. Applies to the <a href="#">PRINT</a> and <a href="#">PRINT USING</a> statements only. Refer to the <a href="#">PRINT</a> statement description for a list of the ANSI forms control characters.
<u>EXTEND</u>	Indicates that a file opened with <a href="#">SEQUENTIAL</a> access method will be written to at the end of any existing data in the file. If the file does not exist, it will be created. When this option is not specified for a <a href="#">PRINT USING OUTPUT</a> file any existing file will first be erased.
<u>LOCK</u>	Indicates that a lock is to be placed on all access to the file so that no other users may read or write to the file while this program has the file open.  If another user has this file open already, this program will wait until that other user closes the file. The <a href="#">OPTION LOCK</a> statement may be used to generate a trappable error in this situation.  When used with a printer, the option LOCK indicates that no page eject is performed when the file is closed.
<u>MULTILOCK</u>	Indicates that all records read from this file are to be locked and to remain locked until the file is closed or until an <a href="#">UNLOCK</a> statement is performed for this file channel. Normally, records are locked until the program performs another access to the file.
<u>QUOTE</u>	Indicates that ASCII fields written to the file will be enclosed in double quote pairs if the field contains leading spaces, trailing spaces, embedded double spaces, or commas. Applies to <a href="#">OUTPUT</a> and <a href="#">UPDATE</a> files when <a href="#">PRINTing</a> or <a href="#">MAT PRINTing</a> to the file/device.

The program may only open files or devices that are attached to the current user, either as a private device or as a public device. Refer to the

ATTACH and SYSGEN commands in the *THEOS System Reference* manual for information about attaching devices.

A device attached as a public resource may only be used one at a time. This means that when a user opens the publicly attached COM1 then any other user attempting to open COM1 will wait until that first user is finished with its access to COM1. A blinking question mark is displayed, indicating that the program is waiting for a shared, public resource.

A disk file on a publicly attached disk drive can be treated just like other public resources like printers, communications ports, etc. Specify the **LOCK** option to indicate that this program is to be the only program that has access to the file, until it is closed. A wait is performed if any other user already has the file open.

The **OPTION LOCK** statement can be used to set the amount of time your program waits for the release of the shared resource. When **OPTION LOCK** is not in effect for a file channel, the program waits until the resource or file is available.

Refer to the *MultiUser BASIC Programmer's Guide* for additional information regarding multiuser operation and using files.

Specifying that a disk file is **OUTPUT SEQUENTIAL**, causes any existing stream file by the same name, to be erased. (If there is an existing non-stream file with the same name then a "File exists" error occurs.) Specifying that the file is **OUTPUT SEQUENTIAL, EXTEND**, does not erase the file and the subsequent output to the file is written to the end of the existing file.

## Statements

### Printer Files

A printer file may be opened with **Mode 1** or **Mode 2** of the OPEN statement. Either will work whether the printer is spooled or nonspooled.

Opening multiple files to a single, nonspooled printer is allowed, but all of the files will be accessing the same printer and the output for each of the open channels might be intermixed on the same page. When the printer is a spooled printer, the two file channels, and their output, are kept separate, as if there were two separate printers on the system.

### Spooled Printers

As stated above, a spooled printer can be accessed by using either **Mode 1** or **Mode 3** of the OPEN statement. However, only **Mode 3** allows the program to specify the report name, spooler queue, copies and hold status of the report.

When **Mode 1** is used to open a spooled printer, the queue, copies and hold status are set according to the current attachment of that printer. For example, if the current attachment is:



## THEOS 32 Device Attachment

Logical Name	Physical Name	Device Number	Options
D	HARD2	4:2:1	"????????", PUBLIC
F	FLOPPY5¼	1:1:0	"????????",HDL50,STP3,PUBLIC
S	HARDDISK	3:2:3	"THEOS",107.4mb,PUBLIC
CON	VGA	5:3:7	L80,P24,PCTERM
PRT1	SPOOLER		L80,P58,QUEUE=A,COPIES=1,C135
PRT2	SPOOLER		L80,P58,QUEUS=D,COPIES=3

Then,

```
OPEN #4: "PRT", OUTPUT SEQUENTIAL
OPEN #16: "PRT2", OUTPUT SEQUENTIAL
```

will open channel 4 to the spooler with a QUEUE=A, COPIES=1, and NOHOLD status; channel 16 will be opened to the spooler with a QUEUE=D, COPIES=3, and NOHOLD status.

By using [Mode 3](#) of the OPEN statement, these defaults can be overridden:

```
OPEN #4: "PAYROLL.C1H:PRT1", OUTPUT SEQUENTIAL
OPEN #16: "REGISTER.A1N:PRT1", OUTPUT SEQUENTIAL
```

opens channel 4 to the spooler with a QUEUE=C, COPIES=1, and HOLD status; channel 16 is opened to the spooler with a QUEUE=A, COPIES=1, and NOHOLD status. Notice that the *spooler-specs* allows the program to specify a *report-name*. This *report-name* is seen whenever a SPOOLER LIST command is given and helps identify reports that are still in the spooler's list of reports available to print.

To specify queue letters outside of the range A—Z you must precede the letter with the dollar sign character ( \$ ). When the \$ character is not used, the queue letter is folded to uppercase before the report is given to the spooler. For instance:

```
OPEN #4: "PAYROLL.$C1H:PRT1", OUTPUT SEQUENTIAL
```

This statement opens channel 4 with a queue of “c” (lowercase C).

The *report-name* has the same restrictions as a fn: 1—8 characters in length, alphanumeric or underscore characters only.

## Named Pipes

A “named pipe” may be opened with this statement. A named pipe is a special type of file that is used for interprocess communication. A named pipe can be used between any two tasks, programs, or users. For example, between two users on the system at the same time or between two users on the system at different times, between two subtasks, etc.

To use a named pipe the applications must agree upon the name of the pipe. All pipes are named as if they were a file in the directory `/PIPE/`. The two programs that use a named pipe each open the pipe as a file. For example, the name of the pipe is `WORK.DATA`. The program that will write to the pipe opens it with the statement:

```
OPEN #40: "/PIPE/WORK.DATA", OUTPUT SEQUENTIAL
```

and the program that will read from the pipe opens it with:

```
OPEN #30: "/PIPE/WORK.DATA", INPUT SEQUENTIAL
```

It doesn't matter if the receiving program opens the pipe first—the program will wait for the pipe to be created by the originator.

A program writing lots of data to the pipe might be delayed by the receiving program if the receiving program doesn't empty the pipe. Pipes have a fixed buffer size of 4K bytes. When the buffer becomes full, the writing program's execution pauses until the buffer is read and space becomes available.

This is not normally a consideration since pipes are not intended to be used to transfer large data bases.

### Named Pipe Restrictions

A named pipe is implemented as a memory file. Its duration lasts until it is written and closed and read and closed. The named pipe does not exist after the system is reset.

### Restrictions

The *channel* must have a value in the range 1–999. Values outside of this range generate a trappable error number 38 “Invalid file channel ... out of range.” The *channel* must not be already open or a trappable error 28 will occur: “File “*file-desc*” is still open.”

A file's actual access method must match the *method* specified with the OPEN statement or a nontrappable error of “Invalid access method for “*file-spec*”.” occurs.

When opening any file, other than an `OUTPUT SEQUENTIAL` file, the file must already exist or a trappable error 30 will occur: “File “*file-desc*” not found.”

The file must not have “read protection” when opened with **INPUT** or **UPDATE** or write protection when opened with **OUTPUT** or **UPDATE** or a trappable error 33 will occur: “File "file-desc" is protected.”

If an existing file is opened for **OUTPUT SEQUENTIAL**, and that file has erase protection, a trappable error 33 occurs.

See also **CLOSE**

Examples

<i>This routine opens five files: a spooled printer for a report, the member CUS-TOMER from the default library, and three more files from that library.</i>	1720 NAME.LIST: 1730 1740 <b>OPEN #14: "CUSTLIST.A1N:PRT1", OUTPUT SEQUENTIAL</b> 1750 1760 <b>OPEN #1: "CUSTOMER", INPUT INDEXED</b> 1770 <b>OPEN #2: "CUSTXREF", INPUT INDEXED</b> 1780 1790 <b>OPEN #3: "INVOICES", INPUT INDEXED</b> 1800 <b>OPEN #4: "INV\$HIST", INPUT INDEXED</b> ~
<i>The ERROR.LOG file is opened so records can be added to the end of the file.</i>	1000 <b>OPEN #17: "ERROR.LOG", OUTPUT SEQUENTIAL, EXTEND</b> ~
<i>The COM1 device is opened and data is sent to set the answer mode to one ring.</i>	10240 <b>OPEN #40: "COM1", OUTPUT SEQUENTIAL</b> 10250       PRINT #40: "ATSO=1" \ REM Set auto answer on 10260       CLOSE #40

# OPTION Statement

The OPTION statement sets or changes various parameters affecting execution of the program.

**OPTION** *option* [, *option*]....

<i>option</i>	»	<b>BASE 0   1</b> <b>BCD</b> <b>CALL</b> <i>entry-module-name</i> <b>CASE</b> <i>case-string-exp</i> <b>COMMA</b> <b>DATASIZE</b> <i>size</i> <b>DEGREE</b> <b>IEEE</b> <b>LIBRARY</b> <i>object-library-name</i> <b>LOCK</b> <i>seconds</i> <b>LOGICAL</b> <b>NOCOMMA</b> <b>PRIV</b> <i>level</i> <b>PROMPT</b> <i>prompt-string</i> <b>RADIAN</b> <b>SERIAL</b> <i>serial-number</i> <b>STACKSIZE</b> <i>size</i> <b>VERSION</b> <i>version</i> [, <i>copyright</i> ]
<i>case-string-exp</i>	»	single character string of <b>U   L   M</b>
<i>seconds</i>	»	numeric expression
<i>level</i>	»	numeric literal
<i>prompt-string</i>	»	string literal
<i>serial-number</i>	»	<i>integer</i> <sub>1</sub> - <i>integer</i> <sub>2</sub>
<i>version</i>	»	<i>integer</i> <sub>1</sub> [. <i>integer</i> <sub>2</sub> [. <i>integer</i> <sub>3</sub> ]]
<i>copyright</i>	»	string literal
<i>size</i>	»	Size of data or stack in K (1,024 bytes)

**Operation**      The operating mode of the program is changed according to the option(s) specified.

## Options

Options are of two types: directives and executable.

Directive type options apply to the entire program, and frequently to any subsequent programs that are [CHAINED](#) to or [LINKed](#) to, from this program. These options are labeled with 'D' in the following table.

Executable options affect subsequent operation of the program and can be changed many times during the execution of a program or set of programs. These are labeled with an 'E' in the table.

Option	Type	Description
CALL	E	Establishes the program modules containing entry point vectors for C language functions used in the program. This option is only effective during the execution of interpretive programs and it is ignored by the compiler. Refer to the <i>MultiUser BASIC Programmer's Guide</i> for additional information.  Multiple OPTION CALL statements may be used in a program allowing the program to use functions in various toolkit and user-defined modules. A maximum of 32 OPTION CALL statements may be used in a program.
LIBRARY	D	Specifies an object library used during compilation of the program.  Multiple OPTION LIBRARY statements may be used allowing multiple object libraries to be searched and used during the compilation of the program. As many as 32 OPTION LIBRARY statements may be used in a program.

## Examples

```
10 REM Program: SAMPLE
20
30  OPTION VERSION 1.0,"Copyright 2001 by Your Company"
40  OPTION SERIAL 102-54321, PRIV 3, BASE 1, BCD, NOCOMMA
50  OPTION PROMPT ""
60  OPTION LIBRARY "B3220TK"
70  OPTION LIBRARY "MYLIB.OBJLIB"
80  OPTION CALL "MYFCTS"
90  OPTION CALL ""
~
1000  OPTION CASE "U"
1010  LINPUT "Enter your name: " USING SPACE$(10), OP.NAME$
1020
1030  OPTION CASE "M", PROMPT ": "
1040  PRINT "Enter choice: Quit, Continue, Stop, Restart"
1050  LINPUT USING "!", SELECTION$
```

---

## ORD Function

ORD returns the ASCII code value of a single character.

ORD( *string-expression* )

<b>Operation</b>	The ASCII code value of the first character in the <i>string-expression</i> is returned.
<b>Notes</b>	<p>This function is a synonym to the <a href="#">ASC</a> function.</p> <p>Refer to Appendix B: “<a href="#">THEOS Character Set</a>” for a table of the ASCII code values.</p>
<b>See also</b>	<a href="#">ASC</a>

# OTHERWISE Statement

The OTHERWISE statement is used as part of a [SELECT-CEND](#) programming structure. OTHERWISE marks the end of all [CASE](#) statements and specifies the series of statements to execute if no [CASE](#) statement was true.

OTHERWISE

**Operation** If no [CASE](#) statement was true in this program structure, then the statements following this OTHERWISE statement will execute. If any [CASE](#) statement was true, control is transferred to the closing [CEND](#) statement.

**Notes** [CASE](#) statements are evaluated in the sequence specified in the program structure. When the result of a [CASE](#) statement's relational expression is false, the statements following the [CASE](#) are bypassed until another [CASE](#), OTHERWISE or [CEND](#) statement is encountered.

When the result of a [CASE](#) statement is true the statements following the [CASE](#) statement are executed until another [CASE](#), OTHERWISE or [CEND](#) statement is encountered, at which point control is transferred to the terminating [CEND](#) statement, skipping all of the remaining [CASE](#) and OTHERWISE statements in the program structure.

**Restrictions** There should be one, and only one, OTHERWISE statement in a [SELECT](#) program structure and it should follow all [CASE](#) statements in the structure. Although no error is reported or detected when there are multiple OTHERWISE statements in the program structure or when [CASE](#) statements follow an OTHERWISE statement, those superfluous statements groups are never executed.

**See also** [BREAK](#), [CASE](#), [CEND](#), [SELECT](#)

Examples

```
~
20060      SELECT INTYPE$(FLD%)
20070          CASE "$" \ REM Field is dollar field
20080              FIELD$ = STR$(REC(FLD%))
20090              DFLD$ = STR$(DUP(FLD%))
20100          OTHERWISE \ REM None of the above
20110              FIELD$ = REC$(FLD%)
20120              DFLD$ = DUP$(FLD%)
20130      CEND
~
```

---

# OVR\$ Function

OVR\$ returns a string with another string overlaid onto or into another string.

OVR\$( *string-expression, from, length, overlay-string-expression* )

**Operation**      The *overlay-string-expression* is overlaid onto *string-expression*. This is done by first expanding or truncating *overlay-string-expression* to a length of *length*. This modified *overlay-string-expression* is then used to replace the portion of *string-expression* starting at position *from*. Any part of *string-expression* that was longer than the overlaid string is retained.

**Notes**            The substring operator can also be used to perform string overlays.

**See also**         [LET](#)

## Examples

Replace, or insert, if necessary, the two characters at columns 17 and 18 of MAIL.ADDR\$(3) with the two characters from the field STATE\$.

10100      MAIL.ADDR\$(3) = OVR\$(MAIL.ADDR\$(3),17,2,STATE\$)

Assigns the result back to MAIL.ADDR\$(3).

Replace, or insert, if necessary, the nine characters starting at column 20, with the nine characters of the ZIPCODE\$ field. Assign the result back to MAIL.ADDR\$(3).

10110      MAIL.ADDR\$(3) = OVR\$(MAIL.ADDR\$(3),20,9,ZIPCODE\$)

~

Statements



---

# PAGE Function

PAGE returns one less than the attached page length of an I/O channel.

PAGE( *channel* )

**Operation**      The page length (base 0) of the attached console or printer opened on *channel* is returned.

When *channel* is for the console, the page length returned will be one less than the attached page length or, if any windows are open, one less than the depth of the currently active window.

**Notes**            A *channel* of zero always refers to the standard output device (stdout), which is normally the console.

This function is only usable for consoles and printers. If the *channel* refers to a disk or tape file, or to a communications device, a zero is always returned.

**I/O Redirection**    When *channel* is 0 this statement actually refers to the current standard output device (stdout). Normally this is the console display. However, stdout may have been redirected when the program was invoked:

```
>myprog > text.file
```

When stdout has been redirected and *channel* is 0, the PAGE function returns the attached page length of the stdout device if it is a non-spoiled printer. When stdout is not the console display or a printer, it returns a zero.

A program can determine if the stdout device has been redirected to a file or device other than the console by using the [SYS.ENV\\$](#) function:

```
IF SYS.ENV$(34,"STDOUT") = "Y"
```

The above test is true when stdout has been redirected.

**Restrictions**      The *channel* must be 0 or refer to an open file channel.

**See also**          [LINE](#)

Statements

## Examples

*This program displays an entire text file on the screen.*

*The **PAGE** function is used to detect when a page wait should be performed, allowing the operator to read the information before it scrolls off of the screen.*

*This same code will operate correctly when the display is to a window that is smaller than the full screen.*

```
10 OPEN #1: "SAMPLE.TEXT", INPUT SEQUENTIAL
20
30 LINE.COUNT% = 0 \ REM Initialize line count
40 LINPUT #1: RECORD$ \ REM Get the first text line
50
60 WHILE NOT EOF(1) \ REM Repeat until end of file
70     IF LINE.COUNT%>=PAGE(0) \ REM Bottom of screen?
80         WAIT \ REM Wait for operator to release screen
90         LINE.COUNT% = 0
100        IFEND
110        LINE.COUNT% = LINE.COUNT%+1 \ REM Count line
120        PRINT LEFT$(RECORD$,LINE(0))
130        LINPUT #1: RECORD$ \ REM Get another text line
140        WEND
150
160 CLOSE #1
170
180 END
```

---

## PI Function

PI returns the value of  $\pi$  (3.14159265359).

PI

**Operation**      The constant value of 3.14159265359 is returned.

**Notes**            Although there are several ways that  $\pi$  can be computed, such as  $\text{ATAN}(1)*4$ , this function is the most efficient because it is an embedded constant in MultiUser BASIC.

**Examples**

```
1200      INPUT "Radius of circle: ",RADIUS
1210
1220      PRINT "Circumference =";2*PI*RADIUS
1230      PRINT "Area =";PI*RADIUS^2
```

---

## PLOT Statement

PLOT is a VDI statement used to draw a point, line, or connected lines.

- 1 **PLOT** [*#channel*:] [*color*;] *x*, *y*
  - 2 **PLOT** [*#channel*:] [*color*;] *x*<sub>1</sub>, *y*<sub>1</sub>; *x*<sub>2</sub>, *y*<sub>2</sub> [; ...; *x*<sub>*n*</sub>, *y*<sub>*n*</sub>]

**Operation**      **Mode 1**—A marker is drawn at location *x,y*. The current marker size, and style are used. The current marker color is used unless the *color* is specified with this statement. Refer to the [SET MARKER](#) statement for a description of the various colors, sizes, and styles of markers available.

**Mode 2**—One or more lines are drawn from location *x*<sub>1</sub>,*y*<sub>1</sub> to *x*<sub>2</sub>,*y*<sub>2</sub>, from location *x*<sub>2</sub>,*y*<sub>2</sub> to *x*<sub>3</sub>,*y*<sub>3</sub>, from location *x*<sub>3</sub>,*y*<sub>3</sub> to *x*<sub>4</sub>,*y*<sub>4</sub>, etc. The current line width and style are used, and the current line color is used unless the *color* is specified with this statement.

**Notes**            Refer to the [SET LINE](#) statement for a description of the various colors, sizes, and styles of lines available.

Refer to the *MultiUser BASIC Programmer's Guide* for a description of VDI devices, statements, and a complete program example.

Invalid parameters do not cause errors. Invalid coordinates are plotted as the maximum value (32767); invalid colors are set to the maximum color code.

The *channel* number must refer to an open channel. That channel, or the default VDI1 device, must be graphics capable or the trappable error message number 37 "Graphics not available." is reported.

**See also**        [SET LINE](#), [SET MARKER](#)

Examples

*A simple line graph is drawn. The first plot statement draws the x and y axis lines.*

*Plot the line graph.*

*Draw markers at each point of the graph.*

```
2000 SET LINE STYLE 1, COLOR 7
2010
2020 PLOT 1000,22000;1000,1000;22000,1000 \ REM Draw axes
2030
2040 SET LINE STYLE 2
2050
2060 PLOT 5000,2200;10000,15000;15000,8900;20000,12000
2070
2080 SET MARKER STYLE 4, SIZE 100, COLOR 4
2090
2100 PLOT 5000,2200 \ REM Plot markers at line segment ends
2110 PLOT 10000,15000
2120 PLOT 15000,8900
2130 PLOT 20000,12000
~
```

---

## PLOT ARC Statement

PLOT ARC is a VDI statement used to draw an arc of a circle.

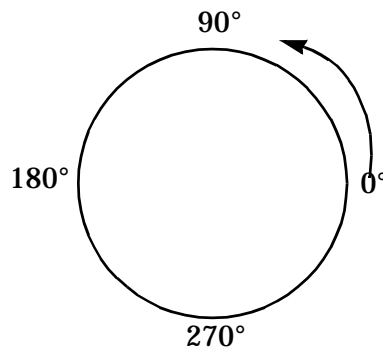
**PLOT ARC** [#channel:] [color;] *x,y, radius, angle<sub>1</sub>, angle<sub>2</sub>*

**Operation** An arc of a circle is drawn with the circle centered at location *x,y* and the with a radius of *radius*. The arc is drawn counterclockwise from *angle<sub>1</sub>* to *angle<sub>2</sub>*. The current line width is used with a line style of 1, solid. The current line color is used unless the color is specified with this statement.

**Notes** Refer to the [SET LINE](#) statement for a description of the various colors, sizes, and styles of lines available.

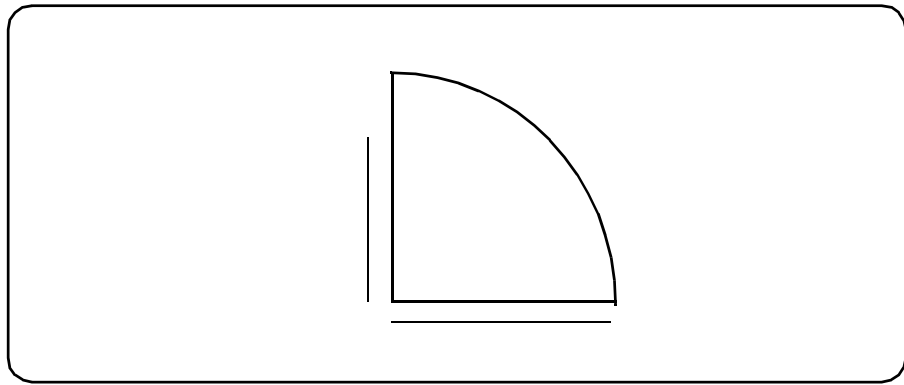
Refer to the *MultiUser BASIC Programmer's Guide* for a description of VDI devices, statements, and a complete program example.

The angle specification is in radians or degrees, depending upon the current setting of the [OPTION RADIAN](#) or [OPTION DEGREE](#). The angle specifications are counterclockwise with the 0 angle at the 3 o'clock position:



Arcs are always drawn as part of a round circle, even when the horizontal and vertical scale of the device are not equal. When the scales are unequal, the points along the circumference are adjusted to make the arc round. The only circumference points that are *radius* distance from the center point *x,y* are the points on the horizontal radius.

```
10 OPTION DEGREE
20 PLOT ARC 16000,16000;10000,0,90
30 PLOT 16000,15000;26000,15000 \ REM Horizontal radius
40 PLOT 15000,16000;15000,26000 \ REM Vertical radius
```



The two radii drawn are both 10,000 units long. But, because the vertical size of the display area is smaller than the horizontal size, the vertical radius appears shorter.

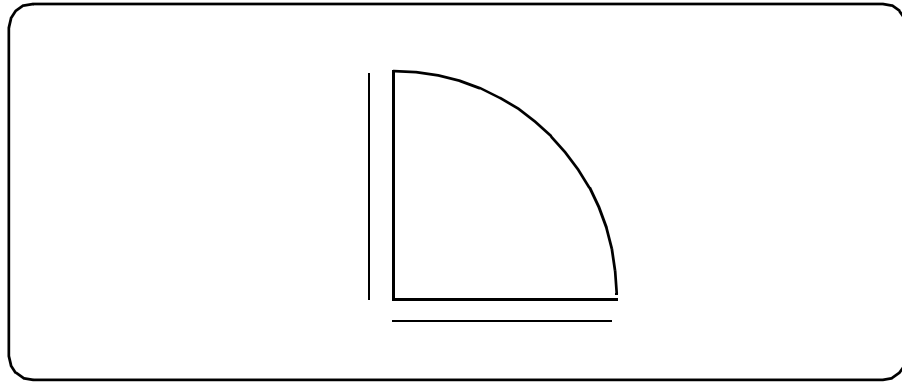
When necessary, the program can determine the difference in scale and adjust for the difference. Use the [VDI](#) statement, open function, to access the device specifications and capabilities. Some of the output array values returned include the horizontal and vertical scales. Use this information to adjust the vertical coordinate of a point or line drawn on the radius of a circle or arc.

```

10  OPTION BASE 1
20  GOSUB  COMPUTE.YADJUST
30  PLOT ARC 16000,16000;10000,0,90
40  PLOT 16000,15000;26000,15000 \ REM Horizontal radius
50  PLOT 15000,16000;15000,16000+10000*YADJUST \ REM Vertical
~
1000  COMPUTE.YADJUST:
1010
1020  DIM VDI.C%(6),VDI.I%(10),VDI.PI%(1),VDI.O%(45),
      VDI.PO%(6,2)
1030
1040  VDI.C%(1) = 1 \ REM Command is open
1050  VDI.C%(4) = 0 \ REM No input
1060
1070  VDI VDI.C%,VDI.I%,VDI.PI%,VDI.O%,VDI.PO% \ REM Open
1080
1090  YADJUST = ((FLOAT(VDI.O%(1))+1)/(VDI.O%(2)+1))*
      VDI.O%(4)/VDI.O%(5)
1100
1110  RETURN

```

Refer to the *MultiUser BASIC Programmer's Guide* for a description of the y coordinate adjustment.



**Restrictions** Invalid parameters do not cause errors: Invalid coordinates are plotted as the maximum value (32767); invalid colors are set to the maximum color code; invalid angles are modularized to the range 0—360 degrees (0— $2\pi$  radians).

The *channel* number must refer to an open channel. That channel, or the default VDI1 device, must be graphics capable or the trappable error message number 37 “Graphics not available.” is reported.

**See also** [FILL PIE](#), [PLOT CIRCLE](#), [PLOT PIE](#), [SET LINE](#)

Statements

**Examples**

```
OPTION DEGREE 10      OPTION DEGREE, BASE 1
is used so that
angles can be spec-
ified in degrees
instead of radians.

20
30   DIM A(9) \ REM Data items
40
50   GOSUB COMPUTE.YADJUST \ REM Get scaling factor
60
70   MAT READ A \ REM Get data items
80
Draw pie slices.      90   OX% = 10000 \ OY% = 16000 \ REM Pie chart center
100
110  FOR I% = 1 TO 8
120      PLOT PIE OX%,OY%;8000,A(I%),A(I%+1)
130      NEXT
140
```



<i>Highlight one item</i>	150	<b>PLOT ARC</b> <b>OX%,OY%;9000,A(2),A(3)</b> \ REM Highlight slice
<i>with an arc drawn</i>	160	
<i>next to the slice and</i>		
<i>a line from the arc.</i>		
 <i>COS and SIN are</i>	170	ANG = A(2)+(A(3)-A(2))/2 \ REM Compute mid slice angle
<i>used to compute the</i>	180	
<i>point on the cir-</i>	190	PX1% = OX%+9000*COS(ANG)
<i>cumference that is</i>	200	PY1% = OY%+9000*SIN(ANG)*YADJUST
<i>in the middle of the</i>	210	PX2% = OX%+12000*COS(ANG)
<i>selected slice.</i>	220	PY2% = OY%+12000*SIN(ANG)*YADJUST
	230	
<i>Draw the lines.</i>	240	PLOT PX1%,PY1%;PX2%,PY2%;PX2%+5000,PY2%
	~	
	1000	COMPUTE.YADJUST:
	1010	
	1020	DIM VC%(6),VI%(10),VPI%(1),V0%(45),VP0%(6,2)
	1030	
	1040	VC%(1) = 1 \ REM Command is open
	1050	VC%(4) = 0
	1060	
<i>Use the VDI state-</i>	1070	VDI VC%,VI%,VPI%,V0%,VP0%
<i>ment to access the</i>	1080	
<i>horizontal and ver-</i>		
<i>tical scales.</i>		
 <i>FLOAT is used to</i>	1090	YADJUST = ((FLOAT(V0%(1))+1)/(V0%(2)+1))*V0%(4)/V0%(5)
<i>ensure that the</i>	1100	
<i>result is not inte-</i>	1110	RETURN
<i>gerized.</i>		

---

## PLOT BAR Statement

PLOT BAR is a VDI statement used to draw a rectangle.

**PLOT BAR** [#*channel*:] [*color*;] *x1*, *y1*; *x2*, *y2*

**Operation**      A rectangle is drawn. The position and size of the rectangle is defined by the two opposite corners *x1,y1* and *x2,y2*. The current line width is used with a line style of 1, solid. The current line color is used unless the *color* is specified with this statement.

**Notes**            Refer to the [SET LINE](#) statement for a description of the various colors, sizes, and styles of lines available. Refer to the *MultiUser BASIC Programmer's Guide* for a description of VDI devices, statements and a complete program example.

The interior of the bar is not filled and is transparent to objects previously drawn that lie underneath the bar.

The rectangle is drawn with the corners precisely at the coordinates specified. An attempt to draw a square rectangle might be thwarted by a non-uniform horizontal and vertical scale, as used on most consoles and printers. For example:

```
PLOT BAR 5000,5000;10000,10000
```

would be expected to draw a square of size 5000. Because of the mapping of the VDI world coordinate system to the actual device coordinates, the rectangle will not be square unless the device's horizontal and vertical scales are equal.

A program can take into account a difference in the device's horizontal and vertical scaling by using the [VDI](#) statement. Refer to the [PLOT PIE](#) and [PLOT ARC](#) statements for an example of this adjustment to device coordinates.

Invalid parameters do not cause errors. Invalid coordinates are plotted as the maximum value (32767); invalid colors are set to the maximum color code.

The *channel* number must refer to an open channel. That channel, or the default VDI1 device, must be graphics capable or the trappable error message number 37 "Graphics not available." is reported.

**Examples**

```

10      OPTION BASE 1
20
30      DIM A(8)
40
50      MAT READ A
60
70      FOR I% = 1 TO 8 \ REM Determine largest value
80          MAX.VALUE = MAX(MAX.VALUE,A(I%))
90      NEXT
100
110     PLOT 5000,30000;5000,5000;30000,5000 \ REM Draw axes
120
130     BAR.WIDTH% = 25000/16
140
150     BAR.START% = 5000+BAR.WIDTH%/2
160
170     FOR I% = 1 TO 8
180         BAR.HEIGHT% = 5000+25000*A(I%)/MAX.VALUE
190         BAR.END% = BAR.START%+BAR.WIDTH%
200         PLOT BAR BAR.START%,5000;BAR.END%,BAR.HEIGHT%
210         BAR.START% = BAR.END%+BAR.WIDTH%
220     NEXT
~
9000    DATA 25,55,40,95,30,55,15,45

```

---

## PLOT CIRCLE Statement

PLOT CIRCLE is a VDI statement used to draw a circle in a color and style.

**PLOT CIRCLE** [#*channel*:] [*color*;] *x*, *y*, *radius*

**Operation**      A circle is drawn centered at location *x,y* with a radius of *radius*. The current line width is used with a line style of 1, solid. The current line color is used unless the *color* is specified with this statement.

**Notes**            Refer to the [SET LINE](#) statement for a description of the various colors, sizes and styles of lines available.

Refer to the *MultiUser BASIC Programmer's Guide* for a description of VDI devices, statements and a complete program example.

The interior of the circle is not filled and is transparent to objects previously drawn that lie underneath the circle.

Circles are always drawn round, even when the horizontal and vertical scales of the device are not equal. When the scales are not equal, the points along the circumference of a circle are adjusted to make the circle round. The only circumference points that are *radius* distance from the center point *x,y* are the points along the horizontal radius. For example:

```
10 PLOT CIRCLE 16000,16000;10000
20 PLOT 16000,16000;26000,16000 \ REM Horizontal radius
30 PLOT 16000,16000;16000,26000 \ REM Vertical radius
```

Notice that the vertical radius does not reach the circumference. This is due to a difference in scale in the horizontal and vertical directions, typically found on terminals and printers.

Refer to the [PLOT ARC](#) or [PLOT PIE](#) statements for a description of a method that the program can utilize to accommodate this nonuniform scaling.

Invalid parameters do not cause errors. Invalid coordinates or lengths (radius) are plotted as the maximum value (32767); invalid colors are set to the maximum color code.

The *channel* number must refer to an open channel. That channel, or the default VDI1 device, must be graphics capable or the trappable error message number 37 "Graphics not available." is reported.

**Examples**

```

10      OPTION BASE 1
20
30      DIM VALUES(5),LABELS$(5)
40
50      READ TITLE$
60      MAT READ LABELS$ \ MAT READ VALUES
70
80      FOR I% = 1 TO 5
90          TOT.VALUE = TOT.VALUE+VALUES(I%)
100         NEXT
110
120     FACTOR = 30000/(TOT.VALUE*2) \ REM Compute scaling
130
140     INITIAL.X% = 0 \ REM X coordinate of left side
150
160     SET TEXT SIZE 4 \ REM Size for title
170     TEXT.X% = 16000-LEN(TITLE$)*700/2
180     TEXT TEXT.X%,28000;TITLE$
190
200     SET TEXT SIZE 1 \ REM Size for labels
210     FOR I% = 1 TO 5
220         RADIUS% = VALUES(I%)*FACTOR
230         X% = INITIAL.X%+RADIUS% \ REM Center coordinate
240         PLOT CIRCLE X%,16000;RADIUS%
250         TEXT.X% = X%-LABELS$(I%)*350/2 \ REM Compute start
260         TEXT TEXT.X%,15750;LABELS$(I%)
270         INITIAL.X% = INITIAL.X%+2*RADIUS%
280     NEXT
290
~
9000    DATA "Sales Comparison"
9010
9020    DATA "John","Bob","Susan","Jerry","Monica"
9030    DATA 60,20,40,30,35
9040
9050    END

```

*Get the data for the displays.*

*Compute the total amount of all values.*

*Calculate the scaling factors for the circles.*

*Display the graph title.*

*Display circles in sizes comparable to the data values.*

*Place the data labels in the center of each circle.*

---

## PLOT PIE Statement

PLOT PIE is a VDI statement used to draw a “pie-shaped” area.

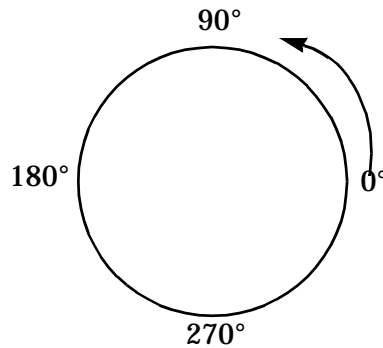
**PLOT PIE** [#channel:] [color;] *x, y, radius, angle<sub>1</sub>, angle<sub>2</sub>*

**Operation**      A “pie-slice” is drawn. A pie-slice is a wedge of a circle centered at location *x,y* with a radius of *radius*. The pie-slice arc is drawn counterclockwise from *angle<sub>1</sub>* to *angle<sub>2</sub>*. The current line width is used with a line style of 1, solid. The current line color is used unless the *color* is specified with this statement.

**Notes**            Refer to the [SET LINE](#) statement for a description of the various colors, sizes, and styles of lines available.

Refer to the *MultiUser BASIC Programmer's Guide* for a description of VDI devices, statements, and a complete program example.

The *angle* specifications are in angles or degrees, depending upon the current setting of the [OPTION RADIAN](#) or [OPTION DEGREE](#). The angle specifications are counterclockwise with the 0 angle at the 3 o'clock position:



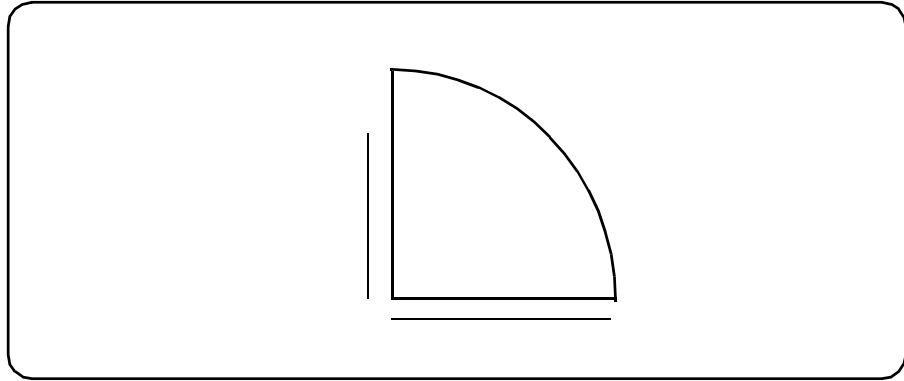
The interior of the pie-slice is not filled and is transparent to objects previously drawn that lie underneath the slice.

Pie slices are always drawn as part of a round circle, even when the horizontal and vertical scale of the device are not equal. When the scales are unequal, the points along the circumference are adjusted to make the arc round. The only circumference points that are *radius* distance from the center point *x,y* are the points on the horizontal radius.

```

10 OPTION DEGREE
20 PLOT PIE 16000,16000;10000,0,90
30 PLOT 16000,15000;26000,15000 \ REM Horizontal radius
40 PLOT 15000,16000;15000,26000 \ REM Vertical radius

```



The two radii drawn are both 10,000 units long. But, because the vertical size of the display area is smaller than the horizontal size, the vertical radius appears shorter.

When necessary, the program can determine the difference in scale and make accommodations for it. Use the [VDI](#) statement, open function, to access the device specifications and capabilities. Some of the output array values returned include the horizontal and vertical scales. Use this information to adjust the vertical coordinate of a point or line drawn on the radius of a circle or arc.

```

10 OPTION BASE 1
20 GOSUB COMPUTE.YADJUST
30 PLOT PIE 16000,16000;10000,0,90
40 PLOT 16000,15000;26000,15000 \ REM Horizontal radius
50 PLOT 15000,16000;15000,16000+10000*YADJUST \ REM Vertical
~
1000 COMPUTE.YADJUST:
1010
1020 DIM VDI.C%(6),VDI.I%(10),VDI.PI%(1),VDI.O%(45),
VDI.PO%(6,2)
1030
1040 VDI.C%(1) = 1 \ REM Command is open
1050 VDI.C%(4) = 0 \ REM No input
1060
1070 VDI VDI.C%,VDI.I%,VDI.PI%,VDI.O%,VDI.PO% \ REM Open
1080
1090 YADJUST = ((FLOAT(VDI.O%(1))+1)/(VDI.O%(2)+1))*
VDI.O%(4)/VDI.O%(5)
1100
1110 RETURN

```

Statements

## Restrictions

Invalid parameters do not cause errors. Invalid coordinates or lengths (radius) are plotted as the maximum value (32767); invalid colors are set to the maximum color code; invalid angles are modularized to the range 0–360 degrees or 0– $2\pi$  radians.

The *channel* number must refer to an open channel. That channel, or the default VDI device, must be graphics capable or the trappable error message number 37 “Graphics not available.” is reported.

## See also

[FILL PIE](#), [SET LINE](#)

## Examples

```
OPTION DEGREE      10      OPTION BASE 1, DEGREE
is used so that    20
angles can be spec- 30      DIM A(9)
ified in degrees   40
instead of radians. 50      GOSUB COMPUTE.YADJUST
                   60
Get y-scaling fac- 70      MAT READ A
tor.               80      OX% = 10000 \ OY% = 16000 \ REM Circle center
                   90
Draw the pie.      100     FOR I% = 1 TO 8
                   110     PLOT PIE OX%,OY%;8000,A(I%),A(I%+1)
                   120     NEXT
                   130
Highlight the sec-  140     SET FILL STYLE 6
ond slice with a   150     FILL PIE OX%,OY%;8000,A(2),A(3)
pattern and a line 160     ANG = A(2)+(A(3)-A(2))/2 \ REM Compute mid-angle
drawn from the
slice to a text label.
COS and SIN are    170     PX1% = OX%+8000*COS(ANG) \ REM mid-angle tangent point
used to compute the 180     PY1% = OY%+8000*SIN(ANG)*YADJUST
point on the cir-  190     PX2% = OX%+1200*COS(ANG)
cumference that is 200     PY2% = OY%+12000*SIN(ANG)*YADJUST
in the middle of the 210
selected slice.    220     PLOT PX1%,PY1%;PX2%,PY2%;PX2%+5000,PY2%
                   230
                   240     SET TEXT SIZE 4
                   250     TEXT PX2%+5300,PY2%-400;"Sample"
                   ~
1000 COMPUTE.YADJUST:
1010
1020     DIM VC%(6),VI%(1),VPI%(1),VO%(45),VPO%(6,2)
1030
1040     VC%(1) = 1 \ REM Command is open
1050     VC%(4) = 0 \ REM No inputs
1060
1070     VDI VC%,VI%,VPI%,VO%,VPO%
1080
```

Use the VDI statement to access the horizontal and vertical scales.



<i>The FLOAT function is used to ensure that the result is not integerized.</i>	1090	YADJUST = ((FLOAT(V0%(1))+1)/(V0%(2)+1))*V0%(4)/V0%(5)
	1100	
	1110	RETURN
	1120	
	9000	DATA 0,25,80,120,215,245,300,315,360
	9010	
	9020	END

---

# POS Function

POS returns the current column position of output for an I/O channel.

POS( *channel* )

**Operation**      The current cursor column number for the terminal or printer opened on channel *channel* is returned. If the *channel* refers to a disk file then the number of characters written on the last output to that channel is returned.

**Notes**            A *channel* of zero always refers to the console or standard output device (stdout).

                      The position number returned for *channel* zero is base 0.

                      For example:

```
PRINT AT$(10,10);POS(0)
```

                      displays the value 9.

**See also**         [TAB](#)

**Examples**         ~

<i>Print the customer name.</i>	11000	PRINT #14: LPAD\$(CUST.ID\$,5);" ";CUSTNAME\$;
	11010	
<i>If the customer name was short</i>	11020	IF <b>POS(14)</b> <=40
<i>enough, print the</i>	11030	PRINT #14: TAB(40);CUST.PHONE\$
<i>phone number at</i>	11040	ELSE PRINT #14: TAB(60);CUST.PHONE\$
<i>column 40; other-</i>	11050	IFEND
<i>wise, print it at col-</i>		
<i>umn 60.</i>		

Statements

# PRINT Statement

The PRINT statement displays or writes ASCII data to a terminal, device, or disk file.

1   **PRINT** [*comma*] [*expression* *punct*]...

2   **PRINT** #*channel*: [*comma*] [*expression* *punct*]...

3   **PRINT** #*channel*, *key*: [*comma*] [*expression* *punct*]...

4   **PRINT** IOLIST *listname* [*punct*]

5   **PRINT** #*channel*: IOLIST *listname* [*punct*]

6   **PRINT** #*channel*, *key*: IOLIST *listname* [*punct*]

*comma*

»   ,

*punct*

»   ,

;

- Operation

**Mode 1**—Display data on the console, starting with the current cursor position.

**Mode 2**—Print data to a sequential device or file.

**Mode 3**—Print data to a direct, keyed or indexed data file record, starting at the beginning of the record.

**Mode 4**—Display data on the console, starting with the current cursor position, using *listname* as the list of variables to print.

**Mode 5**—Print data to a sequential device or file, using *listname* as the list of variables to print.

**Mode 6**—Print data to a direct, keyed or indexed data file record, starting at the beginning of the record, using *listname* as the list of variables to print.

Notes

In [Mode 1](#), [Mode 2](#) and [Mode 3](#), the punctuation characters before, between and after the expressions in the PRINT statement specify the number of space characters output before the next expression is printed: Commas cause the cursor or “print-head” to align to the next print field (21

column boundary); Semicolons cause no space to display. (The option QUOTE used on the [OPEN](#) statement changes the meaning of the punctuation characters. See note, below.)

The terminating punctuation character has an additional meaning. Normally, the PRINT statement prints lines or records, terminated by a carriage return. When a terminating punctuation character is used the cursor or print-head is positioned to the next print field (comma) or not moved (semicolon) and no terminating carriage return is output. When this is done, the next PRINT statement to the same device or file displays its text on the same line as this PRINT statement's output.

For example:

```
PRINT "This is the first line"
PRINT "This is the second line";
PRINT "and so is this."
```

displays:

```
This is the first line
This is the second line and so is this.
```

When printing to a direct, keyed or indexed file ([Mode 3](#) and [Mode 6](#)), the terminating punctuation on the PRINT statement does suppress the printing of the carriage return. However, the next print or write to the file will be at the beginning of the record for the key specified.

Printing a record to a file opened for UPDATE access without the MULTILOCK option causes all record locks on that file channel to be released.

#### ■ Printing to the Console (Mode 1 and 4)

When printing to the console with [Mode 1](#) of the PRINT statement, the AT\$ function may be used to move the cursor to any location on the console screen. The punctuation character following the AT\$ function expression has the same meaning as described above. The normal syntax is to follow an AT\$ function expression with a semicolon to keep the cursor at the location specified by the AT\$ function.

For example:

```
PRINT AT$(5,3); \ REM Position to column 5, line 3
PRINT AT$(8,10) \ REM Position to column 1, line 11
```

The AT\$ function may be output to devices or files other than the console, but that would print only the value of the AT\$ function expression. This value positions the cursor only when it is output to the console.

The **CLS\$** function, when printed to the console, not only clears the screen, but also moves the cursor to the top line, first character (position 1,1). It should also be followed by the semicolon character.

The **TAB** function may only be used with **Mode 1** and **Mode 2** of the PRINT statement. It causes the cursor or print-head to be moved to the column indicated. If the cursor or print-head is already past that column on the current line, then a new line or record is started and the cursor or print-head is positioned to that column on that new line.

For example:

```
PRINT "X";TAB(5);"Y"
PRINT "AAAAA",TAB(5);"B"
```

displays:

```
X   Y
AAAAA
      B
```

**Numeric expressions:** The ASCII value of a numeric expression printed with this statement will always have a trailing space character plus any spaces specified by the punctuation character. Positive numeric expressions will have a leading space character (negative numeric expressions have a leading minus sign character). To print a number without the surrounding spaces print the **STR\$** function of the numeric value or use the **QUOTE** option, as described next.

#### ■ OPEN with Option QUOTE and PRINT Statement

Using **Mode 2** or **Mode 3** with a file or device that was opened with option **QUOTE** causes paired quotes to surround any expression that contains commas, leading or trailing spaces or embedded multiple spaces.

**Comma punctuation:** Using the comma punctuation character on a PRINT to an option **QUOTE** device or file causes a space, comma to be printed followed by the value of the next expression.

**Semicolon punctuation:** Semicolons cause no space characters to be printed.

**Terminating punctuation:** Terminating punctuation characters cause no carriage return to be printed. The next PRINT statement will not start a new line or record.

**Numeric expressions:** The printing of a numeric expression is different when option **QUOTE** is in effect: No spaces are added to the value of the expression. Numeric expressions print as if the **STR\$** function is used.

```
PRINT #10: 1;2;-3;4.5;-6
PRINT #10: 1,2,-3,4.5,-6
```

prints as:

Without option QUOTE	With option QUOTE
1 2 -3 4.5 -6	12-34.5-6
1 2 -3 4.5 -6	1,2,-3,4.5,-6

On the first line printed, because semicolons were used between the numeric expressions the display with the QUOTE option doesn't use spaces.

The QUOTE option should be used on data files or devices whose records are created with the PRINT statement and retrieved with the [INPUT](#) statement. Do not use the QUOTE option for a file that is "block formatted." Use the [PRINT USING](#) statement without the QUOTE option in effect for this type of output.

#### ■ OPTION COMMA with PRINT Statement

With [OPTION COMMA](#) in effect the printing of numeric values changes. Specifically, the decimal point is printed as a comma character and the separating character between fields depends upon the QUOTE option.

```
PRINT #10: "12345678901234567890"
PRINT #10: 1.23,45.67
```

with [OPTION NOCOMMA](#), prints as:

Without option QUOTE	With option QUOTE
1.23 45.67	1.23,45.67

with [OPTION COMMA](#), prints as:

Without option QUOTE	With option QUOTE
1,23 45,67	1,23;45,67

Note that the semicolon character is used to separate the fields when the QUOTE option is in effect.

## ■ OPEN with Option FORMAT and PRINT Statement (ANSI Forms Control)

Using [Mode 2](#) or [Mode 3](#) with a file or device that was [OPEN](#)ed with option [FORMAT](#) indicates that each record output to the file or device uses ANSI forms control characters to control the page advancement. Although the [FORMAT](#) option is intended to be used on printers only it may be used for any type of file.

The ANSI forms control standard specifies that the first character of each record is the page advancement control. This character is never printed. If the special characters are not added to each line, the first character that was intended to be displayed will be interpreted as the ANSI forms control character.

Character	Meaning
1	Start a new page.
+	Do not advance—overprint the previous line. This can only be done if the printer does not perform an automatic line advance with each carriage return. (Option ALF not specified on printer attachment.)
0	Advance two lines (skip one blank line)
-	Advance three lines (skip two blank lines)
	All other characters are not printed and cause one line to advance. By convention, the space character is used for this purpose.

Remember that terminating a [PRINT](#) statement with a punctuation character causes no carriage return to be printed. This means that the next [PRINT](#) statement does not start a new record and does not start with an ANSI forms control character.

When option [FORMAT](#) is not specified on the [OPEN](#) each record output starts a new line unless it is terminated with a semicolon.

The key for a direct-access file must be a positive valued, numeric expression. The key for keyed- and indexed-access files must be a string expression.

## ■ PRINT with IOLIST

The IOLIST keyword in [Mode 4](#), [Mode 5](#) and [Mode 6](#) operate exactly as if the list of variables in the IOLIST were specified with the PRINT statement, separated by semicolons. For instance:

```
IOLIST CUSTOMER = NAME$,ADDR$,CITY$,STATE$,BALANCE
PRINT #4: IOLIST CUSTOMER
```

executes the same as:

```
PRINT #4: NAME$;ADDR$;CITY$;STATE$;BALANCE
```

Terminating punctuation may be used with the IOLIST keyword and operates the same as the comparable statement without the IOLIST keyword. That is:

```
IOLIST CUSTOMER = NAME$,ADDR$,CITY$,STATE$,BALANCE
PRINT #4: IOLIST CUSTOMER;
```

executes the same as:

```
PRINT #4: NAME$;ADDR$;CITY$;STATE$;BALANCE;
```

## ■ Printer Bypass

Sometimes it is desirable to send a character to a printer or console without having the character be converted, in any way, by the operating system. This can be done by preceding the character with the *bypass* character. For consoles, the bypass character is ESC, value 27; for printers, the bypass character is character value 255.

Printing the bypass character means that the bypass character is not sent to the printer but the character following is. That following character is sent without any conversion or analysis. For example, the character will not be counted so no carriage return will be appended if the print head is at the end of the line; the character will not be interpreted as a display attribute by the THEOS operating system.



**I/O Redirection** [Mode 1](#) and [Mode 4](#) of this statement and [Mode 2](#) and [Mode 5](#) when *channel* is 0 actually print the text on the current standard output device (stdout). Normally this is the console display. However, stdout may have been redirected when the program was invoked:

```
>myprog > text.file
```

When stdout has been redirected the text displayed by this statement is output to that file, device or pipe.

A program can determine if the stdout device has been redirected to a file or device other than the console keyboard by using the [SYS.ENV\\$](#) function:

```
IF SYS.ENV$(34, "STDOUT") = "Y"
```

The above test is true when stdout has been redirected.

**Restrictions** The *channel* must be open for OUTPUT or UPDATE.

A *channel* that was opened with access method DIRECT, INDEXED or KEYED, requires that [Mode 3](#) or [Mode 6](#) of the PRINT statement be used. Using [Mode 2](#) with this type of file causes a trappable-error 35: "Invalid access mode." Similarly, using [Mode 3](#) or [Mode 6](#) with a file that was opened with access method SEQUENTIAL, also causes an error 35.

Outputting a [TAB](#) function to a DIRECT, KEYED or INDEXED file causes a trappable-error 35.

**See also** [AT\\$](#), [CLS\\$](#), [IOLIST](#), [LINE](#), [MAT PRINT](#), [OPEN](#), [PAGE](#), [POS](#), [PRINT USING](#), [TAB](#)

## Examples

<p><i>This subroutine displays a screen mask. Notice that all of the PRINT statements are terminated with a semicolon. That is done because the AT\$ function is used to position the cursor to exact coordinates.</i></p>	<pre> 1670 DISPLAY.SCREEN: 1680 1690     PRINT AT\$(13,5); "Account name:"; 1700 1710     FOR I% = 1 TO 8 1720         PRINT AT\$(LABELX%(I%)-2,Y%(I%)); 1730         PRINT USING " ' ' e", CHR\$(I%+64), LABEL\$(I%); 1740         PRINT AT\$(X%(I%)-1,Y%(I%)); " ["; 1750             AT\$(X%(I%)+L%(I%),Y%(I%)); "]" ; 1760     NEXT 1770     RETURN ~ </pre>
--	--

*This second routine opens a disk file used to log transactions. The file is opened with **EXTEND** to cause new records to write to the current end-of-file and with option **QUOTE**.*

```
10000 LOG:
10010
10020 OPEN #31:"PROGRAM.LOG", OUTPUT SEQUENTIAL, EXTEND,
      QUOTE
```

*The **PRINT** statement uses comma punctuation so that the option **QUOTE** will separate each field with a single space, comma.*

```
10030 PRINT #31: DATE$(0), TIME$(0), VAL(PID$), OPNAME$,
      PROGRAM$, MSG$
10040 CLOSE #31
10050
10060 RETURN
```

*The **IOLIST** feature of the **PRINT** statement prints the fields as if they were separated by semicolons. For numeric fields, this means that the numbers are printed with spaces between them.*

```
10100 LOG2:
10110
10120 OPEN #58: "DATA.FILE", OUTPUT SEQUENTIAL, EXTEND
10130
10140 IOLIST BALANCES = CURRENT, PRIOR, PRIOR.30, PRIOR.60
10150
10160 PRINT #58: IOLIST BALANCES
10170
10180 CLOSE #58
10190
10200 RETURN
```

# PRINT USING Statement

The PRINT USING statement writes formatted ASCII data to a terminal, device, or disk file.

1   **PRINT USING** *mask* [, *expression-list*] [;]

2   **PRINT #** *channel*: **USING** *mask* [, *expression-list*] [;]

3   **PRINT #** *channel*, *key*: **USING** *mask* [, *expression-list*]

4   **PRINT USING** *mask* **IOLIST** *listname* [, *expression-list*] [;]

5   **PRINT #** *channel*: **USING** *mask* **IOLIST** *listname* [, *expression-list*] [;]

6   **PRINT #** *channel*, *key*: **USING** *mask* **IOLIST** *listname* [, *expression-list*]

*mask*                    »   *string-expression*

*expression-list*       »   *expression* [ , *expression-list*]

- Operation

**Mode 1**—Displays on the console the values of *expression-list* formatted according to *mask*, starting with the current cursor position.

**Mode 2**—Prints to a sequential device or file the values of *expression-list* formatted according to *mask*.

**Mode 3**—Prints to a direct, keyed, or indexed access data file record starting at the beginning of the record the values of *expression-list* formatted according to *mask*.

**Mode 4**—Displays on the console the values of *listname* formatted according to *mask*, starting with the current cursor position.

**Mode 5**—Prints to a sequential device or file the values of *listname* formatted according to *mask*.

**Mode 6**—Prints to a direct, keyed, or indexed access data file record starting at the beginning of the record the values of *listname* formatted according to *mask*.

Notes

The PRINT USING statement formats string and numeric expressions according to the specifications in the *mask* expression. This *mask* expression provides the complete specifications of the format, including string lit-

Statements

erals that print with the expressions, spacing between expression values, justification of string expression values, truncation and padding of string expression values, and all of the capabilities of the [FORMAT\\$](#) function formatting for numeric expressions.

The *mask* is a string expression whose value consists of characters that print exactly as they are in the *mask* and codes that indicate the printing format for the string and numeric expressions in *expression-list*.

Code	Meaning	Example
!	An exclamation character indicates that a single character of the string expression value is to print. Only the first character of the string expression is printed.	"!", "THEOS"  Prints: T
'	A single quote character marks the beginning of a string format specification. This specification may be followed by zero or more justification and length specifications.	"'RRRRRRR", "THEOS"  Prints: THEOS
	'L A single quote followed by one or more "L" or "l" characters means that the string value is left justified in this area.	"'LLLLLLX", "THEOS"  Prints: THEOS X
	'C A single quote followed by one or more "C" or "c" characters means that the string value is center justified in this area.	"'CCCCCCCX", "THEOS"  Prints: THEOS X
	'R A single quote followed by one or more "R" or "r" characters means that the string value is right justified in this area.	"'RRRRRRRX", "THEOS"  Prints: THEOSX

Code	Meaning	Example
	<p>The total length of the string printed is specified by the number of characters in the specification, including the single quote character. For example:</p> <pre>PRINT USING "'LLLL",ANSWER\$;</pre> <p>prints the first 5 characters of the variable ANSWER\$. When there are fewer than 5 characters in the field the value will be printed, followed by sufficient spaces to create a 5 character print area. If the mask specifies center justification, the necessary spaces are divided evenly and print before and after the string value. A right justification specification causes the necessary spaces to print before the string value.</p>	<pre>"'LLL","THEOS"</pre> <p>Prints: THEO</p>

Code	Meaning	Example
	<p>'E</p> <p>A single quote followed by one or more "E" or "e" characters means that the entire string is printed, left justified.</p> <p>Unlike the L, C, and R specifications, the E does not restrict the number of characters printed, although it will always print at least as many characters as are included in the specification. For example:</p> <pre>PRINT USING "'EEEX", "YES"</pre> <p>Prints:</p> <pre>YES X</pre> <p>The mask specification is for an expandable field followed by the literal character "X".</p> <p>There are four characters in the specification (the single quote and the three E's. The string to print has only three characters so they are printed and a space is output to match the minimum length specified. The similar statement:</p> <pre>PRINT USING "'EEEX", "THEOS SOFTWARE CORP."</pre> <p>prints</p> <pre>THEOS SOFTWARE CORP.X</pre> <p>The string expression value is longer than four characters so it is printed with no space padding</p>	<pre>"'E 'E", "THEOS", "SYSTEM"</pre> <p>Prints:</p> <pre>THEOS SYSTEM</pre>

Code	Meaning	Example
\	<p>A backslant character is used as a pair of backslant characters to be an alternate method of specifying 'LLL...</p> <p>Separate the backslants by zero or more nonbackslant characters to define the length of the string printed. Thus, a backslant followed by three spaces and another backslant means that five characters of the string are printed.</p>	<p>"\XX\","THEOS"</p> <p>Prints: THEO</p>
<p>The following mask specifications apply to the formatting of numeric expressions. The masks for numeric expressions specify the complete format for displaying the expression values. This includes leading zeros, comma separators, decimal position, number of characters to the right of the decimal, sign placement and form, asterisk fill, and floating dollar sign.</p> <p>These numeric mask specifications may all be combined. For example, "###,###.##DB".</p>		
#	Format the number with leading zeros suppressed (except for the 1's position).	<p>"####.##",1.23</p> <p>Prints: 1.23</p>
9	Format the number normally, with leading zeros. Each 9 specifies a print position for a numeric. If a 9 is used anywhere to the left of the decimal point, the entire specification is treated as 9's.	<p>"9999.99",1.23</p> <p>Prints: 0001.23</p> <p>"###9.99",1.23</p> <p>Prints: 0001.23</p>
Z	<p>Format the number with leading zeros suppressed except for the 1's position.</p> <p>At least two consecutive Z characters must be used or the format is not recognized.</p>	<p>"ZZZZ.ZZ",12.34,0.56</p> <p>Prints: 12.34 0.56</p>
,	Use commas to separate thousands, millions, etc. Note: the comma and period characters are interchanged when OPTION COMMA is in effect. Only one comma is necessary; additional commas are treated like # characters.	<p>"###,###.##",1234.56</p> <p>Prints: 1,234.56</p>

Code	Meaning	Example
**	Format the number with leading zeros replaced with asterisks (except for the 1's position). These two asterisks must appear at the beginning of the specification. Additional or embedded asterisks are treated as # characters. Numeric values that might be negative must use a trailing sign specification.	<pre> "***,###.##-", -2.34 Prints: *****2.34-  "*****.***", .98 Prints: *****0.98 </pre>
\$\$	Format the number with leading zeros replaced with a floating dollar sign (except for the 1's position). These two dollar sign characters must appear at the beginning of the specification. Additional or embedded dollar signs are treated as # characters. Numeric values that might be negative must use a trailing sign specification.  This specification may be combined with the asterisk fill specification.	<pre> "\$\$\$,###-", -8452.34 Prints:   \$8,452-  "\$\$\$\$\$.\$\$", .98 Prints:   \$0.98 </pre>
+	Used only at the end of a number specification to indicate that a trailing sign character is printed.	<pre> "###,###.##+", 4723.6 Prints:   4,723.60+ </pre>
-	Used only at the end of a number specification to indicate that a trailing negative sign character is printed when the value is negative; a space is printed when the value is positive.	<pre> "###,###.##-", 4723.6 Prints:   4,723.60 "###,###.##-", -45 Prints:   45.00- </pre>
DB	Used only at the end of a number specification to indicate that negative values will print with a trailing DB indicator; positive values print with two spaces at the end.	<pre> "###DB", -13,55 Prints:   13DB   55 </pre>
CR	Used only at the end of a number specification to indicate that negative values will print with a trailing CR indicator; positive values print with two spaces at the end.	<pre> "###CR", -13,55 Prints:   13CR   55 </pre>



Code	Meaning	Example
>	Used only at the end of a number specification to indicate that negative values will print with surrounding braces; positive values print with surrounding single spaces. This specification may not be used with the 9's specification as leading zeros must be suppressed.	<pre>#####.##&gt;",-32.45 Prints: &lt;32.45&gt; \$\$\$\$\$.\$\$&gt;",-15 Prints: &lt;\$15.00&gt;</pre>
<p>The following specifications are used only at the end of a number specification to indicate that a form of "scientific notation" format be used. True scientific notation would be <math>9.99 \times 10^n</math>. With these specifications, the notation will be 9.99En.</p> <p>These specifications indicate the number of positions used for the exponent. They may not be used with the 9's specification.</p> <p>This modified scientific notation will never have leading zeros. The value is adjusted to match the number of significant digits specified.</p> <p>Caution: this format does not detect any errors if the value is too large or too small to be displayed with the full exponent. For example:</p> <pre>PRINT USING "#.###^^",123456789012</pre> <p>prints:</p> <pre>1.235E1</pre>		
^^	One unsigned digit is used.	<pre>###.##^^",12345 Prints: 123.45E2</pre>
^^^	One signed digit is used.	<pre>#.####^^^",12345678 Prints: 1.2346E+7</pre>
^^^ ^	Two signed digits are used.	<pre>###.####^^^",PI Prints: 314.1593E-02</pre>
^^^ ^^	Three signed digits are used.	<pre>#.####^^^",.012345 Prints: 1.2345E-002</pre>

The field specifications in the mask are used in a one-to-one relationship to format the fields in *expression-list*. For example:

```
PRINT USING "'E ###.## #### $$###.##",NAME$,UNIT.PRICE,QTY,AMOUNT
```

Additionally, if there are more expressions in *expression-list* than there are field specifications in *mask*, the *mask* is reused. Each time that the *mask* is reused, a new line or record is started.

For example:

```
PRINT USING "####",1,2,3,4; \ PRINT "X"
```

displays:

```
1
2
3
4X
```

The semicolon punctuation at the end of the PRINT USING statement only takes effect when the last expression is printed.

Printing a record to a file opened for UPDATE access without the MULTILOCK option causes all record locks on that file channel to be released.

#### ■ OPEN with Option COMMA and PRINT USING Statement

When **OPTION COMMA** is in effect, the formatting of numeric values is changed. Specifically, the decimal point is printed as a comma character and the separator between thousands and hundreds is a printed as a period character.

For example:

```
PRINT #10: USING "##,###.## ##,###.##",1234.56,8974.32
```

with **OPTION NOCOMMA**, prints as:

```
1,234.56      8,974.32
```

with **OPTION COMMA**, prints as:

```
1.234,56      8.974,32
```

## ■ PRINT USING with IOLIST

The IOLIST keyword in [Mode 4](#), [Mode 5](#) and [Mode 6](#) operate exactly as if the list of variables in the IOLIST were specified with the PRINT USING statement. For instance:

```
MASK$ = "'LLLLLLLLLL 'LLLLLLLLLL 'LLLLLLLLLL 'L"  
IOLIST CUSTOMER = NAME$, ADDR$, CITY$, STATES$  
PRINT #4: USING MASK$, IOLIST CUSTOMER
```

executes the same as:

```
PRINT #4: USING MASK$, NAME$;ADDR$;CITY$;STATES$
```

Terminating punctuation may be used with the IOLIST keyword and operates the same as the comparable statement without the IOLIST keyword. That is:

```
IOLIST CUSTOMER = NAME$,ADDR$,CITY$,ST$  
PRINT #4: USING OUTPUT.MASK$, IOLIST CUSTOMER;
```

executes the same as:

```
PRINT #4: USING OUTPUT.MASK$, NAME$,ADDR$,CITY$,ST$;
```

**I/O Redirection** [Mode 1](#) and [Mode 4](#) of this statement and [Mode 2](#) and [Mode 5](#) when *channel* is 0 actually print the text on the current standard output device (stdout). Normally this is the console display. However, stdout may have been redirected when the program was invoked:

```
>myprog > text.file
```

When stdout has been redirected the text displayed by this statement is output to that file, device or pipe.

A program can determine if the stdout device has been redirected to a file or device other than the console keyboard by using the [SYS.ENV\\$](#) function:

```
IF SYS.ENV$(34,"STDOUT") = "Y"
```

The above test is true when stdout has been redirected.

### Restrictions

The *key* for a direct-access file must be a positive valued, numeric expression. The *key* for keyed- and indexed-access files must be a string expression.

The *channel* must refer to a *channel* that is open for OUTPUT or UPDATE.

A *channel* that was opened with access method DIRECT, INDEXED or KEYED, requires that [Mode 3](#) of the PRINT USING statement be used. Using [Mode 2](#) with this type of file causes a trappable-error 35: “Invalid access mode.” Similarly, using [Mode 3](#) with a file that was opened with access method SEQUENTIAL, also causes an error 35.

Numeric expression values that cannot be represented using the *mask* specifications will print a percent symbol (%) followed by the unformatted value.

For example:

```
PRINT USING "#### ##.###", -3456,123.45
```

prints:

```
%-3456 % 123.45
```

because the first value requires five places to print the number with the required sign, and the second value has three digits to the left of the decimal, but the *mask* only provides for two.

Specifying a string mask for a numeric expression value also prints the value with the percent symbol.

Specifying a numeric mask for a string expression value causes the PRINT USING statement to terminate execution; control will pass to the next statement in the program. Any portions of the PRINT USING prior to this mismatch will print.

**See also**

[FORMAT\\$](#) function, [IOLIST](#), [MAT PRINT](#), [OPEN](#) and [PRINT](#) statements

---

*PRINT USING* 437



# PUBLIC Declaration

PUBLIC declares specific variables and arrays used in a program to be *global in scope* and *static in duration* with the main program and all subprograms of the main program.

<b>PUBLIC</b> <i>variable-list</i>		
<hr/>		
<i>variable-list</i>	»	<i>variable-name</i> [, <i>variable-list</i> ] <i>array</i> [, <i>variable-list</i> ]
<i>array</i>	»	<i>array-name</i> ( <i>subscript</i> ) <i>array-name</i> ( <i>subscript</i> <sub>1</sub> , <i>subscript</i> <sub>2</sub> )

**Operation**      The variables declared in the list are marked as *public variables*; arrays declared in the list are *dimensioned* and marked as *public arrays*.

**Notes**            Variables and arrays declared as public are global in scope. This means that the variable can be referenced from any location in the program (main section, subprograms, defined functions) and the value of the variable or an element of the array set in one portion of the program can be used in other portions of the program.

Additionally, a variable or array declared as public does not need to be declared as **SHARED** in a subprogram or user-defined function. They are automatically shared in subprograms and functions.

```
PUBLIC A, B, C
  A = 1
  B = 2
  C = 3
  CALL PROGRAM1
  PRINT "B =";B
...
SUB PROGRAM1
  LOCAL B
  B = A + C
  PRINT "B =";B
END SUB

-RUN
B = 4
B = 2
```

In the above code sample, the subprogram PROGRAM1 does not have to specify that the variables A and C are the public variables defined in the main-line. However, it does have to specify that the variable B is a **LOCAL**

variable so that the subprogram will not use or change the public variable of the same name.

**Restrictions**      The PUBLIC statement must appear on a line by itself (it may only be followed by a [REM](#) statement on the same line).

**See also**            [COMMON](#), [DIM](#), [LOCAL](#), [REDIM](#), [SHARED](#), [STATIC](#)

**Examples**

```
1000 PUBLIC OPTION%
      ...
1400 OPTION% = 48
      ...
5000 SUB PROGRAM1
5010
5020     PRINT OPTION%
5030
5040 END SUB
```

*The subprogram prints the value 46 because the variable OPTION% is public.*



---

## PUT Statement

PUT writes bytes and characters to terminals, devices and files.

- 1 **PUT** *expression-list*
  - 2 **PUT** *#channel: expression-list*

**Operation**      **Mode 1**—Writes characters or bytes to the standard output device (stdout) which is normally the console. When stdout is the console class code translations are performed.

**Mode 2**—Writes characters or bytes to an open file or device channel. When the file is the console, operates the same as mode 1.

**Notes**            Each numeric expression in *expression-list* is output as a single byte, using only the least significant eight bits. String expressions in *expression-list* are output in their entirety.

To cause a byte or character value to be output without class code translation precede the value with 27 for consoles and 255 for printers. This applies to consoles and printers but not communications ports or data files as there is no class code associated with them. Refer to the *THEOS System Reference Manual* for additional information about *printer bypass*.

**I/O Redirection** [Mode 1](#) of this statement and [Mode 2](#) when *channel* is 0 actually puts the characters on the current standard output device (stdout). Normally this is the console display. However, stdout may have been redirected when the program was invoked:

```
>myprog > text.file
```

When stdout has been redirected the characters displayed by this statement are output to that file, device or pipe.

A program can determine if the stdout device has been redirected to a file or device other than the console keyboard by using the [SYS.ENV\\$](#) function:

```
IF SYS.ENV$(34,"STDOUT") = "Y"
```

The above test is true when stdout has been redirected.

**Restrictions**      The *channel* must refer to an open file channel opened with OUTPUT SEQUENTIAL or UPDATE SEQUENTIAL. (Channel number 0 is always open to the standard output device as UPDATE SEQUENTIAL.)

**See also**            [GET](#), [OPEN](#), [PRINT](#)

**Examples**

<i>Open the printer</i>	20	OPEN #1: "PRT", OUTPUT SEQUENTIAL, LOCK
<i>with LOCK, meaning,</i>	30	
<i>do not perform</i>	40	PUT #1: 255,27,"(8U",255,27," )10U"
<i>form-feed upon</i>		
<i>close.</i>		
<i>Outputs initialization</i>	50	PUT #1: 255,27,"&100",255,27,"(8u",255,27,"(s0p12h12v0s0b2T"
<i>strings to program</i>	60	
<i>a laser printer</i>	70	CLOSE #1
<i>to use specific fonts</i>		
<i>or type-faces.</i>		

---

# PUT COMMON Statement

PUT COMMON copies common data variable values from the current subtask to the parent task.

PUT COMMON *variable-list*

Operation	The values of the variables in <i>variable-list</i> are copied to the parent task.
Notes	Refer to the <a href="#">GET COMMON</a> statement for a description of the operation of <a href="#">GET COMMON</a> and PUT COMMON, with example.
Restrictions	<p>It is possible that the main task has insufficient memory available to store the data transferred with this statement. In that event, the trappable error message number 62 “PUT COMMON failed because main task has no more memory.” is reported.</p> <p>The variables in <i>variable-list</i> must be declared as common variables in both the current subtask and in the main task. The trappable error message number 68 “COMMON variable “aaaaa” not found in main task.” is reported.</p>
See also	<a href="#">COMMON</a> , <a href="#">GET COMMON</a>

---

## QUIT Statement

QUIT terminates execution of the program and returns control to the calling environment.

- 1 **QUIT**
  - 2 **QUIT** *numeric-expression*
  - 3 **QUIT** *string-expression*

**Operation**      The QUIT statement always terminates program execution. Terminating program execution with this statement closes all open I/O channels, cleans up the program stack, and deactivates all subtask programs of this program. Open windows are **not closed** by this statement.

**Mode 1**—Program execution is terminated. If this program is executing in its compiled form, the return code is set to 0 and control is returned to the calling environment. The calling environment is the state or program running when MultiUser BASIC was entered. This might be the CSI, an EXEC program, WindoWriter, another compiled MultiUser BASIC program (invoked with its [CSI](#) or [SYSTEM](#) statement), a compiled C language program, etc.

**Mode 2**—Identical to [Mode 1](#) except that the *return code* is set to the value of *numeric-expression*.

**Mode 3**—Identical to [Mode 1](#) except that control does not return to the calling environment. Instead, *string-expression* is evaluated and interpreted as a command with command line arguments that is invoked. The calling environment of this program is remembered and when the indicated command exits, that environment is re-entered.

**Notes**              When executing programs with the MultiUser BASIC interpreter, the QUIT statement terminates execution of the program but does not exit from the interpreter environment. This is an intentional difference between a program's compiled and interpreted execution.

The command name specified in [Mode 3](#) with *string-expression* may be an abbreviation (if the account environment indicates that abbreviations are okay) or a synonym (if the account environment indicates that synonyms are okay and specifies a synonym file containing the synonym).

**Restrictions**      Return codes may have values in the range 0–65535.

The command name specified with *string-expression* should be the name of a compiled program or EXEC program that exists. No error is detected or reported by MultiUser BASIC, however, it will be reported by the THEOS CSI.

See also [CSI](#), [END](#), [STOP](#), [SYSTEM](#)

Examples

Main program exit routine.	2000	END.RUN:
	2010	
First, clear the primary window, turn the cursor on, display the window and exit to the LOGOFF command.	2020	WINDOW SELECT 0, UPDATE OFF
	2030	PRINT CRT\$("KON");CLSS;
	2040	WINDOW SELECT 0
	2050	
	2060	QUIT "LOGOFF"
	~	
	~	
This section of code might be used in an error handling routine. If the error code indicated that the error was a command or program not found, then link back to the menu selection program; otherwise, exit with a return code of 253.	280	IF ERR.NBR%=40 ! Command not found?
	290	LINK "MENU"
	300	ELSE QUIT 253
	310	IFEND
	~	

Statements

# RAD Function

RAD returns the number of *radians* represented by a value in degrees.

**RAD( *degree-expression* )**

*Operation*      *degree-expression* is interpreted as a number of degrees. The radian equivalent of those degrees is returned.

**Notes**            There are 360 degrees in a circle, and  $2\pi$  radians in a circle. Therefore,

$$1\text{radian} = \frac{180^\circ}{\pi} = 57.29578^\circ$$

The sign of *degree-expression* is retained as the sign of the function.

**See also**        [DEG](#)

## Examples

*This simple program computes the sine of an angle specified in degrees.*

```
10    OPTION RADIAN, PROMPT " "
20
30    INPUT "Enter angle in degrees: ",MY.ANGLE
40    PRINT
50
```

*Because the current option is RADIANS the angle must be converted to radians before being given to the SIN function.*

```
60    MY.ANGLE.RAD = RAD(MY.ANGLE)      ! Convert to radians
70    MY.ANGLE.MIN = FP(MY.ANGLE)*60      ! Determine minutes
80    MY.ANGLE.SEC = FP(MY.ANGLE.MIN)*60 ! and seconds
90
100    MY.ANGLE$ = STR$(IP(MY.ANGLE))&CHR$(248)&
         STR$(IP(MY.ANGLE.MIN))&"' "&
         STR$(ROUND(MY.ANGLE.SEC,0))&" "" "
110
120    PRINT USING "The sine of 'e (###.#### rad) is #.#####",
         MY.ANGLE$,MY.ANGLE.RAD,SIN(MY.ANGLE.RAD)
```

**Enter angle in degrees: 42.624**

**The sine of 42°37'26" ( 0.7439 rad) is 0.67718**

---

# RANDOMIZE Statement

RANDOMIZE seeds the pseudo-random number generator.

RANDOMIZE

**Operation**      The pseudo-random number generator is reseeded to start with a different sequence of random numbers.

**Notes**            Random numbers are numeric values in the range 0–1 (noninclusive). The pseudo-random number generator is an algorithm that, based upon the last number generated, returns a new, pseudo-random number.

If the RANDOMIZE statement is not used in a program, the RND function will return the same sequence of numbers every time the program is executed. This is useful for developing and testing purposes, but it defeats the purpose of the [RND](#) function.

After a program is tested, the RANDOMIZE statement should be added and executed once, during the program initialization or set-up.

**See also**         [RND](#) function

**Examples**

*This routine simulates the rolling of a pair of dice.*

*RANDOMIZE ensures that the program will not be predictable.*

```
10        RANDOMIZE
20
30        DIE1% = 6*RND+1      ! Roll first dice
40        DIE2% = 6*RND+1      ! Roll second dice
50
60        PRINT "You rolled a";DIE1%;"and a";DIE2%;
70        PRINT "for a total of";DIE1%+DIE2%
```

# READ, READNEXT, READPREV Statements

READ accepts data from DATA statements or an input file. READNEXT reads the next record in an indexed-, keyed- or direct-access file. READPREV reads the previous record from an indexed-access file.

- 1

READ *read-list*
- 2

READ #*channel*: *read-list*
- 3

READ #*channel*, *key*: *read-list*
- 4

READNEXT #*channel*, *key-variable*: *read-list*
- 5

READPREV #*channel*, *key-variable*: *read-list*

<i>read-list</i>	»	<b>IOLIST</b> <i>listname</i> <i>variable-list</i>
<i>variable-list</i>	»	<i>variable-name</i> [, <i>variable-list</i> ] <i>array-reference</i> [, <i>variable-list</i> ]
<i>array-reference</i>	»	<i>array-name</i> ( <i>subscript</i> <sub>1</sub> [, <i>subscript</i> <sub>2</sub> ])

## Operation

**Mode 1**—Data is read or copied from the DATA statements in this program into the variables specified in *read-list*. When there are insufficient data items left to assign to the variables in the list, a trappable error 36 occurs: “Out of data.” Extra data items are not used and are available for subsequent READ or MAT READ statements.

**Mode 2**—Reads a record from a sequential-access disk or tape file. One record from the file is read and each field in the record is assigned to consecutive variables in the *read-list*.

**Mode 3**—Reads a record from a direct-, keyed-, or indexed-access disk file. One record from the file is read and each field in the record is assigned to consecutive variables in the *read-list*.

**Mode 4**—The next record in the direct-, keyed-, or indexed-access file is read with the fields of that record assigned to the consecutive variables in *read-list*. The key of the record is assigned to the *key-variable* variable.

**Mode 5**—The previous record in the indexed-access file is read with the fields of that record assigned to the consecutive variables in *read-list*. The key of the record read is assigned to the *key-variable* variable.



When reading from a disk or tape file and there are more variables in the *read-list* than there are fields in the record, the extra variables are cleared to nulls. If there are fewer variables in the *read-list* than there are fields in the record, the extra fields are ignored.

When no record exists matching the key ([Mode 2](#) and [Mode 3](#)), the **EOF** indicator is set and the variables in the *read-list* are cleared to nulls. If the file is an indexed-access file, this read failure also sets the next and previous record pointers so that a subsequent **READNEXT** or **READPREV** will access the record whose key is just greater or just less than the key that produced the read failure. (See example in **READPREV** statement.)

Records in files cannot be read with these statements if they were created by the **MAT PRINT** or the **PRINT** statement. Attempting to do so will cause the variables in the *read-list* to be cleared to nulls. Use the **MAT INPUT**, **INPUT** or **LINPUT** statements to access records of this type.

The record field types and the variable types should match in type (string, integer or numeric). When there is a mismatch between field type and variable type the following conversions are performed:

Field type	Variable type	Conversion
Nonnumeric string	Integer	Set to zero
Nonnumeric string	Numeric	Set to zero
Numeric string	Integer	String to integer
Numeric string	Numeric	String to float
Integer	String	Integer to string (STR\$)
Integer	Numeric	Integer to float (FLOAT)
Numeric	String	Float to string (STR\$)
Numeric	Integer	Float to integer (FIX)

Reading a record from a file that was opened with **UPDATE** mode causes a record-lock to be placed on the record. Unless the file channel was opened with the **MULTILOCK** option, any other record-lock in the file is removed. The unlock operation is performed whether or not this **READ** is successful.

If an attempt is made to read a record that is locked by another user or task, this program will wait until the record-lock is released by that other program. If **OPTION LOCK** is in effect for this program, a trappable error is generated after the indicated time.

Unless option **MULTILOCK** is in effect for the file channel, record-locks are removed or released by reading another record from the same file channel, writing a record to the same file channel, or deleting this record. Perform-

**DATA &  
RESTORE**

ing the **UNLOCK** statement on this file channel, or closing the file channel always unlocks records in the file.

The *read-list* may be specified as **IOLIST** *listname*. The *listname* must be a previously defined **IOLIST** list name. It specifies the variables to be read and the sequence that those variables are read. Refer to the **IOLIST** statement.

When reading the next record (**Mode 4**), the next record in a direct-access file skips any deleted and never-written records. The next record in a keyed-access file is the next physical record in the file. The next record in an indexed-access file is the logically next record in the file according to the sorting order of the record keys.

When reading the previous record (**Mode 5**), the previous record in an indexed-access file is the record whose key logically precedes the current record pointer key.

Refer to the *MultiUser BASIC Programmer's Guide* for additional information about using files.

The **DATA** statement defines numeric and string literals. These literals or values are assigned to variables with the **READ** and **MAT READ** statements. There may be several **DATA** statements in a program, and they may be distributed throughout the program.

During the compilation of a program, all data literals defined with **DATA** statements are collected and placed in one data pool. The sequence of the literals in this pool is the sequence in which they were defined.

There is an internal data pool pointer used by the program that points to the next data literal available to be read. Initially, this pointer is set to point to the first item in the data pool.

As each item is read from the data pool (using either the **READ** or the **MAT READ** statements) the pointer is incremented to point to the next item .

```

10      READ A$,B$,C$
20
30      DATA "ONE", "TWO", "THREE", "FOUR", "FIVE", 6, 7, 8
40      DATA 9, 10, 11, "TWELVE", "THIRTEEN"
50
60      DAY.NAMES:
70
80      DATA "Monday", "Tuesday", "Wednesday", "Thursday"
90      DATA "Friday", "Saturday", "Sunday"
```

In the above sequence of code, there is a total of 20 items in the data pool. The **READ** statement reads the first three items, assigning the literals to

the variables A\$, B\$, and C\$, respectively. The data pool pointer will be pointing to the fourth item in the pool.

```
100
110      READ D$,E$,F%,G%
```

This READ statement reads the next four items from the data pool, assigning the literals to the variables D\$, E\$, F%, and G%, respectively. The data pool pointer will be pointing to the eighth item in the pool.

```
120
130      DIM DAY.NAMES$(7)
140
150      RESTORE DAY.NAMES
160      MAT READ DAY.NAMES$
```

The [RESTORE](#) statement can be used to change the data pool pointer. In the above example, the [RESTORE](#) statement sets the pointer to the first data item defined after the label DAY.NAMES. The [MAT READ](#) statement then reads the next items from the data pool, starting with the item indicated by the data pool pointer. In this case, the pointer is set to item 14, “Monday”.

### Restrictions

When reading from a file ([Mode 2](#), [Mode 3](#), [Mode 4](#) and [Mode 5](#)), the file channel must be opened with INPUT or UPDATE. Refer to the [OPEN](#) statement for details.

When reading from a direct-access file, the *key* must be a numeric expression. Reading from a keyed- or indexed-access file requires that the *key* be a string expression.

The READNEXT statement ([Mode 4](#)) cannot read from a sequential-access file. The READPREV statement ([Mode 5](#)) can only read from an indexed-access file.

### See also

[DATA](#), [INPUT](#), [IOLIST](#), [MAT READ](#), [OPEN](#), [RESTORE](#)

## Examples

<i>The first READ gets a record from an indexed file. Two variables are concatenated to form the key of the record desired.</i>	600320	READ #1,OPNAME\$&","&PW\$: OPINIT\$,PRIVLEV%,HELPS\$, LOOK.AHEAD\$,WAITTIME%,LOCKTIME%,DEFAULT.SYS\$,MSG%
<i>Records are read from a direct access file. After the records are read, the record lock placed on the last read is removed.</i>	8380 8390 8400 8410 900920	FOR I% = 2 TO 27 <b>READ #16,I%: NAME\$(I%-1)</b> NEXT UNLOCK #16 FOR I% = 1 TO LAST.FIELD%
<i>The last set of READ statements reads items from DATA statements.</i>	900930 900940 900950 900960 900970 900980  900990 901000 901010 ~ 990020  990030  ~	<b>READ LABEL.CODE\$,LABEL\$,LABELX%</b> <b>READ NX%(I%),INY%(I%),INTYPE\$(I%),INL%(I%),INCASE\$(I%)</b> <b>READ INPROMPT\$(I%),INHHELP\$(I%),INSPECIAL\$(I%)</b> IF TRIM\$(LABEL.CODE\$)=" " PRINT AT\$(LABELX%,INY%(I%));LABEL\$; ELSE PRINT AT\$(LABELX%-2,INY%(I%));LABEL.CODE\$; " ";LABEL\$; IFEND LABEL.CODES\$ = LABEL.CODES&LABEL.CODE\$ NEXT  DATA "1","Name",3,15,"A",40,"M","ENTER CUSTOMER NAME", "CUSTNAME","CUSTOMER" DATA "2","Address...",3,15,5,"A",30,"M","ENTER DELIVERY ADDRESS, LINE ONE","ADDR1",""

	~	
<i>This code might be used in a customer record look-up routine. Entry of an up or down arrow indicates a request for the previous or next customer record. Entry of a key terminated by an arrow means that the search starts with the key entered. Whether or not this key exists, performing this read initializes the next record and previous record pointers.</i>	10120 10130 10140 10150 10160 10170 10180 10190 10200 ~	<pre> READ.LOOP% = FALSE%           ! Clear loop repeated flag SELECT CASE INP=10                   ! Down arrow = next record   IF KEY\$ THEN READ #1,KEY\$: X\$   GOSUB READ.NEXT.CUSTOMER CASE INP=11                   ! Up arrow = prev record   IF KEY\$ THEN READ #1,KEY\$: X\$   GOSUB READ.PREV.CUSTOMER </pre>
<i>Read the next record in the file.</i>	12000 12010 12020 12030	<pre> READ.NEXT.CUSTOMER:   READNEXT #1,KEY\$: NAME\$, ADDR\$ </pre>
<i>If end-of-file is encountered then position to start of file and read the first record.</i>	12040	IF EOF(1) AND NOT READ.LOOP% ! Not found, 1st time
<i>The READ.LOOP% flag prevents an endless loop when the file is totally empty.</i>	12050 12060 12070 12080 12090 12100 12110 12000 12010	<pre> READ.LOOP% = TRUE%           ! Only one time READ #1,"": NAME\$           ! Position to start of file GOTO READ.NEXT.CUSTOMER IFEND RETURN READ.PREV.CUSTOMER: </pre>
<i>Read the previous record in the file, similar to read next routine.</i>	12020 12030 12040 12050	<pre>   READPREV #1,KEY\$: NAME\$, ADDR\$ IF EOF(1) AND NOT READ.LOOP% ! Not found, 1st time   READ.LOOP% = TRUE% ! Only one time </pre>
<i>The CHR\$(255) is a character value greater than any key in the file. Using it positions the current record pointer past the last record in the file.</i>	12060 12070 12080 12090 12100	<pre>   READ #1,CHR\$(255): NAME\$ ! Position past end   GOTO READ.PREV.CUSTOMER IFEND RETURN </pre>



# REDIM Statement

The REDIM (redimension) statement defines or redefines array names and sizes.

REDIM *array-list*

*array-list*

»

*array*[, *array-list*]

*array*

»

*array-name*( *subscript* )  
*array-name*( *subscript1*, *subscript2* )

Operation

When *array* is not a currently dimensioned array name, this statement operates exactly like the DIM statement, allocating the array and initializing all of the elements to zero or null strings.

When *array* is the name of an existing array, that array is redimensioned, either adding elements to or deleting elements from that existing array.

Notes

Redimensioning an array to a larger size (increasing the number of total elements) will add elements to the array. These added elements are initialized to zero or null strings as appropriate. Redimensioning an array to a smaller size (decreasing the number of total elements) will delete the extra elements from the array. The values of these deleted elements cannot be regained by a subsequent redimension to the original size.

When an array is redimensioned the position of the elements may be changed because the array is treated as a vector array during the process of redimensioning. For instance, an existing 3×5 array may have the following contents:

0,0	0,1	0,2	0,3	0,4
1,0	1,1	1,2	1,3	1,4
2,0	2,1	2,2	2,3	2,4

If this array is redimensioned to 3×3 it will have the following contents:

0,0	0,1	0,2
0,3	0,4	1,0
1,1	1,2	1,3

Statements

That is because, during the redimension process, it is treated as a vector array:

0,0	0,1	0,2	0,3	0,4	1,0	1,1	1,2	1,3	1,4	2,0	2,1	2,2	2,3	2,4
-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----

The shaded elements are deleted.

The scope of an array is not changed by the REDIM statement.

**Restrictions** The REDIM statement cannot change the number of dimensions of an array. That is, a single-dimension array can only be redimensioned as a single-dimension array; a double-dimension array can only be redimensioned as a double-dimension array.

**See also** [COMMON](#), [DIM](#), [LOCAL](#), [MAT](#), [OPTION](#) BASE, [PUBLIC](#), [SHARED](#), [STATIC](#)

**Example**

```
10 OPTION BASE 1
1000 SHARED HISTORY(1,1)      ! Allocate array
    ...
2000 ! Determine number of years of history actually needed
2010 YEARS% = 25
2020 REDIM HISTORY(YEARS%,12) ! Reallocate to actual
```



---

## REM Statement

The REM statement is a remark or comment added to a program's source.

**REM** *any text*

**!** *any text*

<b>Operation</b>	Everything on the line following the remark keyword is ignored. Control passes to the next line of the program.
<b>Notes</b>	<p>A REM statement is ignored by the compiler and does not make the compiled program larger than it would be without it.</p> <p>A blank line (a line with a line number only) is also treated as a remark and is ignored.</p> <p>The exclamation keyword can be used at the end of a statement without the backslant statement separator. There must be at least one space or tab between the last statement and the exclamation character.</p> <p>The position of the exclamation keyword in the source line is maintained, allowing a remark to be aligned with remarks on other lines.</p>
<b>Restrictions</b>	<p>A REM statement may be added to any other statement in a program except <a href="#">DATA</a> statements.</p> <p>No statement may follow a REM statement on the same line. If present, it is treated as part of the remark.</p>
<b>Examples</b>	<pre>10 REM Program: SAMPLE Sample program for doc purposes 30 REM Version 1.0 01/15/1999 50 REM Programmer: Rebecca Doe 60 70     GOSUB SETUP ! Perform program initialization 80     WINDOW SELECT 1 ! Begin in primary application window 90 100    WHILE NOT END.RUN%      ! Repeat until all tasks done 110        GOSUB GET.KEY ! Get record key 120        IF NOT END.RUN% THEN GOSUB ENTER.DATA 130        WEND              ! Repeat loop 140 150    END.RUN: ! All task done, close files and windows, exit 160 170    GOSUB CLOSE.FILES      ! Close the files opened by prog 180    GOSUB CLOSE.WINDOWS   ! Close windows opened by prog 190 200    QUIT ! Exit program</pre>

# REP\$ Function

REPS returns a string with one field or subfield replaced by a new field or subfield.

REP\$( *string-expression, field-number, subfield-number, new-substring* )

**Operation**      The *new-substring* replaces any existing substring in *string-expression* at *field-number, subfield-number*. The resulting string is returned.

**Notes**            String fields can be formatted with fields and subfields with the [INS\\$](#) and [REPS](#) functions. These functions build string such that the string contains two-level subfields delimited by special characters. The [EXT\\$](#) function extracts these subfields for usage as individual strings. The [REPS](#) and [DEL\\$](#) functions modify the subfields.

A string with subfields is useful when a related group of items are manipulated as a group and rarely as individual components. For example, an employee name and address record might have prior-year earnings fields for total earned, vacation pay, sick pay, federal withholding, state withholding, *etc.* These fields might only be included in a special report. If the fields are grouped into one string using subfields, the normal maintenance and reporting programs would read and write the record using only one variable name for the entire set of fields.

The delimiters used are the special characters CHR\$(221) and CHR\$(222).

**See also**            [DEL\\$](#), [EXT\\$](#), [INS\\$](#)

**Examples**            10 COMPRESS: ! Convert maintenance fields to file fields

```
50      RECS(1) = FLDSS(1)
60      RECS(2) = REPS(RECS(2), 1, 0, FLDSS(2))
70      RECS(2) = REPS(RECS(2), 2, 0, FLDSS(3))
80      RECS(2) = REPS(RECS(2), 3, 0, FLDSS(4))
90      RECS(2) = REPS(RECS(2), 4, 0, FLDSS(5))
100     RECS(3) = REPS(RECS(2), 1, 0, FLDSS(6))
110     RECS(3) = REPS(RECS(2), 2, 0, FLDSS(7))
```

---

## RESET Function

RESET clears the status of an event semaphore and returns its prior status.

RESET( *semaphore* )

<b>Operation</b>	The status of the event <i>semaphore</i> is set to off or false. The prior status of the event is returned.
<b>Notes</b>	<p>Semaphores have states of true (-1) or false (0) corresponding to the <a href="#">SET</a> and RESET functions.</p> <p>The <a href="#">WAIT EVENT</a> statement waits for a semaphore to be set and then resets the semaphore before continuing execution.</p>
<b>Restrictions</b>	Semaphore numbers must be in the range of 0–63.
<b>See also</b>	<a href="#">EVENT</a> , <a href="#">SEMAPHORE</a> , <a href="#">SET</a> functions, <a href="#">WAIT EVENT</a> statement

## Examples

The following is a subtask program.

*Similar to the  
ACTIVATE func-  
tion example, with  
a semaphore added  
to synchronize the  
usage of the display  
between the main  
task and the sub-  
task.*

```
10     TIMER% = SEMAPHORE("MINUTE")
20     ONE.ONLY% = SEMAPHORE("ONE-ONLY")
30
40     TIMER TIMER% SYNC 60
50
60 WAIT.TIMER:
70
80     WAIT EVENT(TIMER%)      ! Wait for timer to go off
90
```

*On line 100, the  
semaphore is set,  
but, if the sema-  
phore was already  
set then the task  
sleeps for a moment  
and then checks the  
prior status again.*

```
100    IF SET(ONE.ONLY%) THEN SLEEP .1 \ GOTO 100
110
120    WINDOW STATUS PRIOR.WIN%
130    WINDOW SELECT 0, UPDATE OFF
```

*This process is  
repeated until  
another task clears  
the semaphore,  
indicating that the  
display is available.*

```
140
150    PRINT AT$(72,1);TIME$(0);
160
170    WINDOW SELECT PRIOR.WIN%
```

*On line 180, this  
task also clears the  
semaphore when it  
is finished with the  
display for awhile.  
The RESET func-  
tion is used, ignor-  
ing the prior status.*

```
180    X = RESET(ONE.ONLY%) ! Clear one-only flag
190
200    TIMER TIMER% SYNC 60
210
220    GOTO WAIT.TIMER%
230
240    END
```

Statements

# RESTORE Statement

RESTORE sets the [DATA](#) statement read pointer.

- 1

RESTORE
- 2

RESTORE *line-reference*

Operation	<b>Mode 1</b> —Reset the <a href="#">DATA</a> read pointer to the top of the program.  <b>Mode 2</b> —Reset the <a href="#">DATA</a> read pointer to <i>line-reference</i> .
Notes	<p>The <a href="#">DATA</a> read pointer is an internal pointer used by MultiUser BASIC when data items are read by the READ and <a href="#">MAT READ</a> statements. Initially, the <a href="#">DATA</a> read pointer is set to the top of the program.</p> <p>Each time a READ or <a href="#">MAT READ</a> statement is performed, the <a href="#">DATA</a> read pointer is adjusted to point to the next <a href="#">DATA</a> statement item available. When the last <a href="#">DATA</a> statement item has been read, further attempts to READ or <a href="#">MAT READ</a> data will cause an “Out of DATA” error.</p> <p>The RESTORE statement can reset the <a href="#">DATA</a> read pointer so that READ and <a href="#">MAT READ</a> statements can read the data items again.</p>
See also	<a href="#">DATA</a> , <a href="#">MAT READ</a> , <a href="#">READ</a>

Examples	100	DIM MONTHS\$(12)
	~	
Select the full month names or the abbreviated month names by choosing a different set of DATA statements.	1000	INPUT "Use full names or abbreviations (F/A): ",OPT\$
	1010	
	1020	SELECT OPT\$
	1030	CASE "F"
	1040	<b>RESTORE MONTH.NAMES</b>
	1050	CASE "A"
	1060	<b>RESTORE MONTH.ABBREVS</b>
	1070	CEND
	1080	MAT READ MONTHS\$
	~	
	990000	MONTH.NAMES:
	990010	
	990020	DATA "January","February","March","April","May","June"
	990030	DATA "July","August","September","October","November"
	990040	DATA "December"
	990050	
	990060	MONTH.ABBREVS:
	990070	
	990080	DATA "JAN","FEB","MAR","APR","MAY","JUN"
	990090	DATA "JUL","AUG","SEP","OCT","NOV","DEC"

---

# RESUME Statement

RESUME marks the end of event handling, error handling, or key trapping routines.

- 1

RESUME
- 2

RESUME 0
- 3

RESUME *line-reference*

**Operation**      **Mode 1**—Control is transferred back to the statement that was executing prior to the occurrence of the error, event, or key press. Specifically, if the event handling routine was invoked due to an [ON EVENT](#), [ON KEY](#), [ON MOUSE](#) or [ON TIMEOUT](#) event, control is transferred back to the statement following the statement that was executing when the event occurred. If the event handling routine was invoked due to an [ON ERROR](#) event, control is transferred back to the statement causing the error.

**Mode 2**—Control is transferred to the system’s standard error processing routine. This mode is only valid for [ON ERROR](#) handling routines. The system’s standard error processing routine displays the error message corresponding to the error number that occurred. Once the message is displayed, compiled programs are exited and interpreted programs enter the command mode with the current line pointer set to the error line.

**Mode 3**—Control is transferred to the *line-reference*.

**Notes**      The RESUME statement is only valid when used as an exit from a routine entered because of an error ([ON ERROR](#) statement), an event ([ON EVENT](#) statement), a key press ([ON KEY](#) statement), a mouse action ([ON MOUSE](#), [ON LEFT](#), [ON CENTER](#) or [ON RIGHT](#) statement) or an input time-out ([ON TIMEOUT](#) statement). The results will be unpredictable if the RESUME statement is executed when one of those routines has not been invoked.

Care should be exercised when using [Mode 3](#). This statement can transfer control anywhere, just like a [GOTO](#). And, just like a [GOTO](#), branching out of a program structure such as a subroutine, function definition, [FOR-NEXT](#), [SELECT-CEND](#), [WHILE-WEND](#) is not recommended. Those types of program structures should only be entered at the top and exited at the bottom.

Do not use [Mode 3](#) of the RESUME statement when the event or error invoking the event handler occurred during execution of a user-defined function or subprogram.

**Restrictions**     *line-reference* must exist in the main program area. That is, the *line-reference* may not be a line or label defined between [DEF FN-FNEND](#) statements nor between [SUB-END SUB](#) statements.

The RESUME statement can only be used in the main program area, not in a function definition or a subprogram definition. The error message “Expecting ... item on the stack, found ... item” is reported when a [RESUME](#) statement is used inside of a function or subprogram definition.

The RESUME statement can also only be used in a program that has at least one of the following statements: [ON ERROR](#), [ON EVENT](#), [ON KEY](#), [ON MOUSE](#) or [ON TIMEOUT](#).

**See also**     [ON ERROR](#), [ON EVENT](#), [ON KEY](#), [ON MOUSE](#), [ON TIMEOUT](#)

**Examples**

*This routine is  
invoked when the  
operator presses*

*[F9] or [Shift]+[F9]*

*The exit from the  
ON KEY trap is  
performed with the  
RESUME state-  
ment.*

*When the operator  
has indicated that a  
program exit is  
desired, the special  
form of the  
RESUME state-  
ment is used to  
branch to routines  
that either exit this  
program or exit the  
application.*

*These exit-to loca-  
tions must be pro-  
grammed to not use  
any information  
that might be on the  
stack and to exit  
quickly, which will  
clean up the stack.*

```

700000  QUIT.KEY:
700010
700020      WINDOW STATUS PRIOR.TO.QUIT%
700030
700040      WINDOW OPEN 8,23,10,35,3; FRAME DOUBLE, RIGHT, COLOR
          WHITE%,RED%; COLOR WHITE%,RED%; SELECT
700050
700060      PRINT AT$(2,2);"Do you really wish to QUIT? ";
700070      REPLY$ = YESNO$
700080
700090      WINDOW SELECT PRIOR.TO.QUIT%, UPDATE ON
700100      WINDOW CLOSE 8, REMOVE
700120
700130      IF REPLY$="N"
700140          RESUME                                ! Return to program
700150      ELSE

700160          IF ON.KEY.TOKEN=F9%
700170              RESUME END.PROGRAM                ! Exit back to menu
700180          ELSE RESUME END.APPLICATION ! Exit out of app
700190              IFEND
700200          IFEND
700210      ~

900000  SETUP:
900010
900020      ESC% = 256
900030      TAB.KEY% = 265
          ~
900070      F1% = 269 \ F2% = 270 \ F3% = 271 \ F4% = 272
900080      F5% = 273 \ F6% = 274 \ F7% = 275 \ F8% = 276
900090      F9% = 277 \ F10% = 278 \ F11% = 279 \ F12% = 280
900100      ALT% = 4000H
900110      CTRL% = 2000H
900120      SHIFT% = 1000H
          ~
901000      ON KEY(F9%) GOTO QUIT.KEY
901010      ON KEY(SHIFT%+F9%) GOTO QUIT.KEY
          ~
902000      ON ERROR GOTO ERROR.TRAP
902010
902020      RETURN
902030
902040      END

```



---

# RETURN Statement

The RETURN statement terminates execution of a subroutine and returns control to the statement following the GOSUB or ON GOSUB statement.

1

RETURN

2

RETURN *line-reference*

Operation	<p><b>Mode 1</b>—Returns control to the statement following the GOSUB or the ON GOSUB statement that invoked this subroutine.</p> <p><b>Mode 2</b>—Retrieves the return-to location placed on the program stack by the GOSUB or ON GOSUB statement that invoked this subroutine. That return-to location is discarded and control is transferred to line-reference.</p>
Notes	<p>A subroutine may have more than one RETURN statement. However, it is good programming practice to limit a subroutine to one RETURN statement and to place that RETURN statement at the physical end of the subroutine. To code multiple exit points from a subroutine with a single RETURN statement use a flag variable or conditional execution to accomplish the same result. (See example below.)</p>
Restrictions	<p><i>line-reference</i> must be defined in the program.</p>
See also	<p>GOSUB, ON GOSUB</p>

## Examples

*A subroutine to  
check an input field  
for a valid dollar  
amount.*

```
23000  VALIDATE.AMOUNT: ! Test dollar input for validity
23010
23020      SELECT
23030          CASE NOT NBR(FIELD$) ! Must be a valid number
23040              VALID% = FALSE%
23050              P2MSG$ = "NUMERIC VALUE REQUIRED"
23060          CASE VAL(FIELD$)<>ROUND(VAL(FIELD$),2)
23070              VALID% = FALSE%
23080              P2MSG$ = "FRACTIONAL PENNIES ARE NOT ALLOWED"
23090          CASE VAL(FIELD$)>=10^(INL%(FLD$)-3)
23100              VALID% = FALSE%
23110              P2MSG$ = "DOLLAR VALUE TOO LARGE"
23120      CEND
23130
```

*Return to caller.*

```
23140      RETURN
23150
```

*A subroutine to  
check an input field  
for valid date.  
Skip tests if empty  
field or validation  
override specified.*

```
24000  VALIDATE.DATE: ! Test date input for validity
24010
24020      IF NOT (FIELD$="" OR INP=TRANSPPOSE%)
24030
24040          IF DTE$(FIELD$)=""
24050              VALID% = FALSE%
24060              P2MSG$ = "INCORRECT DATE FORMAT"
24070          ELSE FIELD$ = DTE$(FIELD$) ! Standard format
24080          IFEND
24090
24100      IFEND
24110
24120      RETURN
24130
```

*Return to caller.*

Statements

# RIGHT\$ Function

RIGHT\$ returns the rightmost portion of a string.

RIGHT\$( *string-expression*, *from* )

**Operation** Starting with character position *from*, the rightmost portion of the *string-expression* is returned.

**See also** [LEFT\\$](#), [MID\\$](#)

## Examples

In this code section, the <i>LEFT\$</i> and <i>RIGHT\$</i> functions are used to replace a section of a string.	900300	LOC% = SCH(1,ERROR.MSGS\$(ERR.NBR%), "%f" )
	900310	IF LOC%
	900320	IF ERR.NBR%=40
	900330	ERROR.MSGS\$(ERR.NBR%) = LEFT\$(ERROR.MSGS\$(ERR.NBR%), LOC%-1)&TO.PROG\$&RIGHT\$(ERROR.MSGS\$(ERR.NBR%), LOC%+2)
Note: The sub-string replacement ability of the <i>LET</i> statement provides a better method of performing this operation.	900340	ELSE ERROR.MSGS\$(ERR.NBR%) =
		LEFT\$(ERROR.MSGS\$(ERR.NBR%), LOC%-1)&
	900350	FILENAME\$&RIGHT\$(ERROR.MSGS\$(ERR.NBR%), LOC%+2)
	900360	IFEND
	~	IFEND

Statements

---

## RND Function

RND returns the next *pseudo-random* number.

### RND

**Operation**      The next number available from the pseudo-random number generator is returned.

**Notes**            Random numbers are numeric values in the range:  $0 < \text{RND} < 1$ .

MultiUser BASIC uses an algorithm to generate random numbers. This algorithm generates numbers that are evenly distributed across all of the possible numbers without numerous repetitions. The next number generated by the algorithm is dependent upon the last number generated. This produces a repeatable series which is useful in testing.

The [RANDOMIZE](#) statement causes the starting number to be randomized based upon external, changeable circumstances like the time of day, date, etc. If the [RANDOMIZE](#) statement is not used then a program will use the same series of random numbers each time it is executed.

To produce a random integer in a given range, use the formula:

```
FIX((upper_limit - lower_limit + 1)*RND + lower_limit)
```

For example, to produce a random number between 1–52 inclusive:

```
FIX((52-1+1)*RND+1)  
FIX(52*RND+1)
```

The FIX function can be omitted if the result is assigned to an integer.

**See also**        [RANDOMIZE](#) statement

## Examples

*This loop shuffles a deck of cards. The **RND** function is used to generate a card number between one and 52.*

```
~
6120 FOR CARD% = 1 TO 52
6130   RND.CARD% = 52*RND+1
6140   WHILE CARDS.USED%(RND.CARD%)
6150     RND.CARD% = 52*RND+1
6160   WEND
6170   CARDS.USED%(RND.CARD%) = 1
6180   DECK%(CARD%) = RND.CARD%
6190 NEXT
~
```

*In the program initialization, the **RANDOMIZE** statement must be used to make each execution of the program different.*

# ROUND Function

ROUND returns the value of an expression rounded to a specified number of decimal positions.

**ROUND( *numeric-expression*, *number-of-places* )**

**Operation**      The value of *numeric-expression* is determined and rounded and truncated according to the value of *number-of-places*.

The value of *number-of-places* may be positive or negative. When it is positive it specifies the number of places to the right of the decimal point; when it is negative it specifies the number of places to the left of the decimal point. For example:

ROUND (12345.6789, 2)      => 12345.68  
ROUND (12345.6789, -2)      => 12300

**Notes**      Rounding up or down is determined by the digit at the rounding position. When the digit is five or greater, round up is performed; when the digit is four or less, round down is performed. In the above examples, the first line rounded up (rounding position is eight), the second line rounded down (rounding position is four).

## Examples

*This routine is used to validate an amount entered into a dollar field.*

*The ROUND function is used to determine if rounding the amount entered to the nearest penny changes the value of the amount.*

```
23000  VALIDATE.AMOUNT :
23010
23020      SELECT
23030          CASE NOT NBR(FIELD$)
23040              VALID% = FALSE%
23050              P2MSG$ = "NUMERIC VALUE REQUIRED"
23060          CASE VAL(FIELD$)<>ROUND(VAL(FIELD$),2)
23070              VALID% = FALSE%
23080              P2MSG$ = "FRACTIONAL PENNIES ARE NOT ALLOWED"
23090          CASE VAL(FIELD$)>=10^(INL%(FLD$)-3)
23100              VALID% = FALSE%
23110              P2MSG$ = "DOLLAR VALUE TOO LARGE"
23120      CEND
23130
23140      RETURN
```

---

# RPAD\$ Function

RPAD\$ returns a fixed length string, space padded on the right.

RPAD\$( *string-expression*, *length* )

**Operations**      *string-expression* is expanded to a length of length by adding spaces to the ending of the string. When string-expression is already long enough, the string is unchanged. The result is returned.

**See also**      [LPAD\\$](#)

**Examples**      The following code section is taken from the DEF.FN.INPUT\$ function definition in the sample programs.

The *RPAD\$* function is used to define the length of the input field for the *LINPUT USING* statement.

```
700090      GET.INFIELD:
700100
700110      PRINT AT$(1,1);
700120      LINPUT USING RPAD$(INFIELD$,L%),INFIELD$
```

Later, the *RTRIM\$* function will be used to remove any trailing spaces that were not needed.

```
700130      INFIELD$ = RTRIM$(INFIELD$)
~
```

Statements

# RPT\$ Function

RPT\$ returns a string composed of repeated copies of a string of characters.

RPT\$( repeat-count, string-expression )

**Operation**      The *string-expression* is duplicated *repeat-count* times. The result is returned.

**See also**        [SPACE\\$](#)

**Examples**        The following subroutine displays a record field on the screen, formatting it according to the field type.

The INL% array defines the display length of the fields.

Dollar fields are formatted with two decimal positions. The width of the integer portion of the amount is formatted with the RPT\$ function.

Note the comma in the decimal format. Assuming that the length is at least five, the comma provides for comma delimiters at the thousands position.

```
100000  DISPLAY.FIELD:
100010
100020      PRINT AT$(INX%(FLD%), INY%(FLD%));
100030      SELECT INTYPE$(FLD%)
100040          CASE "$"
100050              PRINT USING RPT$(INL%(FLD%)-4, "#")&" ,.##",REC(FLD%);

100060      CASE "D"
100070          PRINT RPAD$(DTE$(REC$(FLD%)), INL%(FLD%));
100080      OTHERWISE
100090          PRINT RPAD$(REC$(FLD%), INL%(FLD%));
100100      CEND
100110
100120  RETURN
```



---

# RTRIM\$ Function

RTRIM\$, or right trim, returns a string with all trailing spaces removed.

RTRIM\$( *string-expression* )

**Operation** All trailing spaces are removed from *string-expression* and the result is returned.

**Notes** The functions [LTRIM\\$](#), [TRIM\\$](#) and RTRIM\$ all remove spaces from a string. They differ in where the extra spaces are removed from:

Function	Action
<a href="#">LTRIM\$</a>	Remove all spaces on the left side of the string (leading spaces only).
<a href="#">TRIM\$</a>	Remove all leading and all trailing spaces; reduce all multiple, consecutive, embedded spaces to single spaces.
RTRIM\$	Remove all spaces on the right side of the string (trailing spaces only).

**See also** [LTRIM\\$](#), [TRIM\\$](#)

## Examples

*A field is accepted from the operator.*

1000	LINPUT USING RPAD\$(FIELD\$,40),FIELD\$
------	---

*The RTRIM\$ function is used to remove the unnecessary spaces that might have been left on the input field.*

1010	FIELD\$ = RTRIM\$(FIELD\$)
~	

Statements

---

## RUN Statement

RUN terminates execution of the current program, loads and begins execution of another.

**RUN** *program-name-exp*

**Operation**      The program designated by *program-name-expression* is loaded and execution begins with the first statement in that program.

**Notes**            When executed in the interpreter, the program may have a file-type of BASIC or BAS.

**See also**        [CHAIN](#), [CSI](#), [LINK](#), [SYSTEM](#) statements, RUN command

---

## SCH Function

SCH returns the position of a search string in another string.

SCH( *start-position*, *target-string*, *search-string* )

**Operation**      The *target-string* is searched, starting at character position *start-position*, looking for *search-string*. If a match is found then the starting position of the substring is returned; if a match is not found then a 0 is returned.

Zero is always returned when *start-position* is greater than the length of *target-string*.

**Notes**            When *search-string* is null and *start-position* is less than or equal to the length of *target-string*, then *start-position* is always returned. (If *start-position* is 0 then 1 is returned.)

A zero or negative valued *start-position* is treated as a *start-position* of one.

**See also**         [MATCH](#)

Examples

*This first section of code looks for underscore characters in the program name. Each underscore is changed to the dollar sign character.*

1350	WHILE	SCH(1,TO.PROG\$,"_")
1360		YY% = SCH(1,TO.PROG\$,"_")
1370		TO.PROG\$[YY%:YY%] = "\$"
1380		WEND
1390		FILENAME\$ = TO.PROG\$ \ LINK FILENAME\$
~		

*This line searches for the character string "%f". When found, the substring is replaced with TO.PROG\$.*

900300	LOC% =	SCH(1,ERROR.MSGS\$(ERR.NBR%),"%f")
900310	IF	LOC%
900320		IF ERR.NBR%=40
900330		ERROR.MSGS\$(ERR.NBR%) =
		LEFT\$(ERROR.MSGS\$(ERR.NBR%), LOC% - 1)&
		TO.PROG\$&RIGHT\$(ERROR.MSGS\$(ERR.NBR%),LOC%+2)
~		

*The SCH function is very useful for performing a quick validation of input.*

20090	LINPUT	USING "!",SELECTION\$
20100		
20110	IF	SCH(1,"ABCQR1234567890",SELECTION\$)=0
20120		PRINT CRT\$("BELL"):
20130		GOTO 20090
20140		IFEND

---

## SEC Function

SEC returns the secant of an angle.

**SEC( *numeric-expression* )**

**Operation**      The *numeric-expression* is interpreted as an angle expressed in degrees or radians, depending upon the status of the current **OPTION** **DEGREE** or **OPTION** **RADIAN**. The secant of that angle is computed and returned.

**Notes**            Secants have values that range from  $-\infty$  to  $-1$  and  $+1$  to  $+\infty$ .

**Restrictions**    The value of *numeric-expression* must not be an odd multiple of  $\frac{\pi}{2}$  radians (for example,  $\frac{\pi}{2}$ ,  $\frac{3\pi}{2}$ ,  $\frac{5\pi}{2}$  radians *etc.*, or  $90^\circ$ ,  $270^\circ$ ,  $450^\circ$ , *etc.*) as these values are outside this function's domain and cause SEC to return meaningless values.

**See also**        Other trigonometric functions, **OPTION** **DEGREE** and **RADIAN** statement

### Examples

```
10 OPTION PROMPT "", DEGREE
20
30 INPUT "Enter angle in degrees: ",MY.ANGLE
40 PRINT
50
60 MY.ANGLE.MIN = FP(MY.ANGLE)*60      ! Determine minutes
70 MY.ANGLE.SEC = FP(MY.ANGLE.MIN)*60  ! and seconds
80
90 MY.ANGLE$ = STR$(IP(MY.ANGLE))&CHR$(248)&
      STR$(IP(MY.ANGLE.MIN))&"' "&
      STR$(ROUND(MY.ANGLE.SEC,0))&"'"
100
110 PRINT USING "The secant of 'e is #.#####",MY.ANGLE$,
      SEC(MY.ANGLE)
```

Enter angle in degrees: 4.12

The secant of 4°7'12" is 1.00259

Statements

---

# SECH Function

SECH returns the hyperbolic secant of a number.

SECH( *numeric-expression* )

**Operation**      The hyperbolic secant of *numeric-expression* is computed and returned.

**See also**        Other trigonometric functions

**Examples**

```
10      OPTION PROMPT " "  
20  
30      INPUT "Enter area: ",AREA  
40      PRINT  
50  
60      PRINT "The hyperbolic secant of";AREA;"is";SECH(AREA)
```

Enter area: 3.5  
  
The hyperbolic second of 3.5 is 0.06033974412016

---

# SECOND Function

SECOND returns the number of seconds corresponding to a time-of-day string.

SECOND( *string-expression* )

Operation	The number of seconds represented by the time string in <i>string-expression</i> is returned.
Notes	<p>The <i>string-expression</i> should be formatted in the form of hh:mm:ss. It does not have to be formatted with colons; any delimiter between hours, minutes, and seconds, can be used. However, if no delimiters are used, the value is interpreted as hours (for instance, "121500" is interpreted as one hundred twenty-one thousand, five hundred hours).</p> <p>The leading hours, or hours and minutes, can be omitted, but the delimiters must be specified. For example, fifteen minutes can be specified as ":15", fifteen seconds as "::15".</p> <p>Trailing seconds, or minutes and seconds may be omitted by terminating the string. For example, noon can be specified as "12", ten-thirty as "10:30".</p> <p>The time value is not restricted to 24 hours. For example, SECOND( "60" ) returns 216000.</p>
Restrictions	A numeric value is returned by this function and should not be assigned to an integer as it is possible that the time is greater than 9:06:07am.
See also	<a href="#">MSEC</a> , <a href="#">STRTIME\$</a> , <a href="#">TIME\$</a>

Statements

Examples

This code section formats the current time to the nearest minute, in 12-hour format.

Gets the current time to the nearest second.

Determines whether or not to add "am", "pm", "mid", or "noon" to the time and converts to 12-hour time.

601290

CTIME = **SECOND**(**TIME**\$(0))

601300

SELECT

601310

CASE CTIME<60

601320

C.TIME\$ = " 12 mid"

601330

CASE CTIME<**SECOND**("1") ! Before 01:00 am?

601340

C.TIME\$ = "12"&TIME\$(CTIME)[3:5]&"am"

601350

CASE CTIME<**SECOND**("12") ! Morning?

601360

C.TIME\$ = TIME\$(CTIME)[1:5]&"am"

601370

CASE CTIME<**SECOND**("12:00:59") ! Noon?

601380

C.TIME\$ = "12 noon"

601390

CASE CTIME<**SECOND**("13") ! Before 01:00 pm?

601400

C.TIME\$ = "12"&TIME\$(CTIME)[3:5]&"pm"

601410

OTHERWISE

601420

C.TIME\$ = TIME\$(CTIME-**SECOND**("12"))[1:5]&"pm"

601430

CEND



---

## SELECT Statement

The SELECT statement is used as part of a SELECT-CEND programming structure. SELECT marks the beginning of a series of CASE statements.

- 1 SELECT
  - 2 SELECT *expression*

### Operation

**Mode 1**—Begin a “select one of” program structure. In this mode, each of the CASE statements comprising this structure specifies the complete relational expression. Each CASE statement may test an expression that is completely different from other CASE statements in the structure. For example:

```
1000  SELECT
1010      CASE INP=13
           statements performed when case is true
1020      CASE EOF(4)
           statements performed when case is true
1030      CASE FIELD$="END"
           statements performed when case is true
1040      CASE FIELD$="START"
           statements performed when case is true
1050      OTHERWISE
           statements performed when no cases are true
1060  CEND
```

**Mode 2**—Begin a “select one of” program structure, specifying the left portion of the “is equal to” expression completed in each of the CASE statements. For example:

```
1000  SELECT INP
1010      CASE 13 ! Is INP=13 true?
           statements performed when case is true
1020      CASE 10 ! Is INP=10 true?
           statements performed when case is true
1030      CASE 1 ! Is INP=1 true?
           statements performed when case is true
1040  CEND
```

### Notes

With mode 2, expression is evaluated for each of the CASE statements until a true condition exists. Generally, this is only apparent when the expression performs input, output or changes a value as the YESNO\$ and ERRMSG\$ functions do or as a user defined function might.

When this situation exists, it is best to save the result of the expression in a temporary variable and then use that variable as the *expression* in the SELECT statement.

```

1010  REPLY$ = YESNO$
1020  SELECT REPLY$
1030      CASE "Y"
           statements performed when case is true
1040      CASE "N"
           statements performed when case is true
1050  CEND

```

Every SELECT statement must have a closing **CEND** statement.

Each of the **CASE** statements in this program structure are evaluated in sequence. When the result is false (0), the statements following the **CASE** are bypassed until another **CASE**, **OTHERWISE** or **CEND** statement is encountered.

When the result of a **CASE** statement is true the statements following the **CASE** statement are not bypassed. When the next **CASE** or **OTHERWISE** statement is encountered, control is transferred to the closing **CEND** statement.

When all of the **CASE** statements have evaluated false and an **OTHERWISE** statement is encountered, the statements following that **OTHERWISE** are executed. After executing these statements and another **CASE** or **OTHERWISE** statement is encountered, control is transferred to the closing **CEND** statement.

The **OTHERWISE** statement, when used, should follow all of the **CASE** statements in the program structure. Any **CASE** statements following an **OTHERWISE** will never be evaluated or executed.

The **BREAK** statement can be used in a **CASE** or **OTHERWISE** section to break out of the **SELECT-CEND** structure.

This statement defines the beginning of a control structure. This type of program structure should only be entered at the top (**SELECT** statement) and exited at the bottom (**CEND** statement).

**See also**      **BREAK, CASE, CEND, OTHERWISE**

**Examples**      Refer to the example in the preceding **SECOND** function example.

---

## SEMAPHORE Function

SEMAPHORE returns the number of a semaphore name.

SEMAPHORE( *semaphore-name* )

<b>Operation</b>	The existing semaphore names are searched for one with the name of <i>semaphore-name</i> . When one is found, its semaphore number is returned. When one is not found this <i>semaphore-name</i> is added to the pool and assigned a number. That number is returned.
<b>Notes</b>	<p>The case-mode of <i>semaphore-name</i> is irrelevant because semaphore names are maintained in uppercase.</p> <p>Newly cataloged semaphores are not initialized to the false, or reset state.</p>
<b>Restrictions</b>	<p>A return value of -1 means that the <i>semaphore-name</i> is not an existing semaphore and there is no unused semaphore number available. There is a limit of 64 semaphore numbers for each user session (0-63).</p> <p>Although <i>semaphore-name</i> may be a long string, only the first eight characters are used and are treated as a monospace string.</p>
<b>See also</b>	<a href="#">ACTIVATE</a> , <a href="#">EVENT</a> , <a href="#">RESET</a> , <a href="#">SET</a>

## Examples

The following subroutine accepts a single character from the operator (menu selection). If no input is received from the operator for a specified amount of time, a special value is returned.

	9480	GET.SEL:
	9490	
<i>The time out indicator is cleared.</i>	9500	SEL = 0 \ TIME.OUT% = FALSE% \ Z% = RESET(MENU.TIMER%)
	9510	
<i>The event handler for the timer is defined and the timer is programmed for a period of WAIT-TIME%.</i>	9520	ON EVENT(MENU.TIMER%) GOTO INPUT.TIME.OUT
	9530	TIMER MENU.TIMER% ELAPSED WAITTIME%
	9540	
<i>Processing is suspended until the operator enters a character.</i>	9550	WAIT #0 \ GET SEL%
	9560	
<i>Because of the event timer, the WAIT will terminate because of a time out, in which case the event handler is called and its RESUME statement returns control to the GET statement.</i>	9570	ON EVENT(MENU.TIMER%) GOTO 0 ! Clear event timer
	9580	
	9590	IF TIME.OUT% THEN SEL% = 128
	9600	
	9610	SEL\$ = UCASE\$(CHR\$(SEL%)) \ SEL% = ASC(SEL\$)
	9620	
	9630	RETURN
	9640	
	9650	INPUT.TIME.OUT:
	9660	
	9670	TIME.OUT% = TRUE%
	9680	RESUME
	~	
<i>Catalogue semaphore.</i>	900980	MENU.TIMER% = SEMAPHORE("MENU.TIMER")

---

# SET Function

SET sets an event semaphore and returns the status of the semaphore prior to setting.

SET( semaphore )

- Operation**
- The status of the event *semaphore* is set to “on” or “true.” The prior status of the event is returned.
- Notes**
- Semaphores have states of true (-1) or false (0) corresponding to the SET and [RESET](#) functions.
- Restrictions**
- Semaphore numbers must be in the range of 0–63.
- See also**
- [EVENT](#), [RESET](#), [SEMAPHORE](#) functions, [ON EVENT](#), [WAIT EVENT](#) statements

## Examples

*A semaphore is catalogued and the event handler is specified for that event.*

900810      MINUTE.TIMER% = SEMAPHORE( "MINUTE" )

900820

900830      Z% = RESET(MINUTE.TIMER%)

900840

900850      ON EVENT(MINUTE.TIMER%) GOTO MINUTE.TIME.OUT

900860

*The event is programmed to be set every minute.*

900870      TIMER MINUTE.TIMER% SYNC 60

900880

*The semaphore is set on to cause the event to happen now, even if it is not synchronized to the minute.*

900890      X% = SET(MINUTE.TIMER%)

~

Statements

---

## SET FILL Statement

SET FILL defines the color and fill pattern used by subsequent displays of VDI fill areas.

**SET FILL** [#*channel*:] [**COLOR** *color*] [, **STYLE** *style*]

**Operation**      The fill style and color are defined.

**Notes**            This statement does not fill any areas of the display, nor does it change the display of fill areas previously drawn. It only sets the attributes for fill areas drawn in the future. The [FILL](#) statements are the only means of drawing fill areas.

The COLOR attribute is device dependent due to the nature of the attribute. The three basic display types use the following color codes:

**Monochrome Displays**

Color	Code
Black	0
White	1





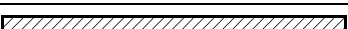
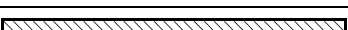

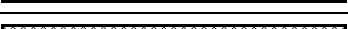
**Color Displays**

Color	Code	Color	Code
Black	0	Red	4
Blue	1	Magenta	5
Green	2	Yellow	6
Cyan	3	White	7

**VGA Color Displays**

Color	Code	Color	Code	Color	Code	Color	Code
Black	0	Red	4	Int Grey	8	Int Red	12
Blue	1	Magenta	5	Int Blue	9	Int Magenta	13
Green	2	Yellow	6	Int Green	10	Int Yellow	14
Cyan	3	White	7	Int Cyan	11	Int White	15

There are at least eight fill **STYLES**:

Style	Pattern	Sample
0	Hollow	
1	Solid	
2	Vertical lines	
3	Horizontal lines	
4	45° diagonals	
5	135° diagonals	
6	Cross hatch	
7	Diagonal cross hatch	

The initial values for fill attributes are: **STYLE** 0 and **COLOR** is the maximum code allowed by the device (1 for monochrome displays, 7 for color displays, 15 for VGA color displays). These attributes are used unless they are changed by a **SET FILL** statement. Fill colors may be specified at the time they are drawn.

When multiple channels are opened to VDI devices, each channel maintains separate fill styles and colors.

**Restrictions** Invalid parameters do not cause errors. However, the specific invalid parameter is treated as a request for the default specification. For example, requesting a **STYLE** 20 causes **STYLE** 0 to be used.

The *channel* number must refer to an open channel. That channel, or the default VDI1 device, must be graphics capable or the trappable error message number 37 “Graphics not available.” is reported.

**See also** [FILL](#), [FILL BAR](#), [FILL CIRCLE](#), [FILL PIE](#)

## Examples

```
10  OPTION BASE 1, DEGREE
20
30  DIM ITEMS(3),ARCS(4)
40  MAT READ ITEMS
50
60  TOTAL = 0
70  FOR I% = 1 TO 3
80      TOTAL = TOTAL+ITEMS(I%)
90  NEXT
100
110 FOR I% = 1 TO 3
120     ARCS(I%+1) = ARCS(I%)+360*ITEMS(I%)/TOTAL
130 NEXT
140
150 SET FILL STYLE 5
160 FILL PIE 16000,160000;6000,ARCS(1),ARCS(2)
170
180 SET FILL STYLE 4
190 FILL PIE 16000,16000;6000,ARCS(2),ARCS(3)
200
210 SET FILL STYLE 3
220 FILL PIE 16000,160000;6000,ARCS(3),ARCS(4)
230
240 WAIT #0 \ GET A$
250
260 DATA 200,120,40
270
280 END
```



## SET LINE Statement

SET LINE defines the color, size, and style of lines drawn with subsequent VDI line drawing statements.

**SET LINE** [#*channel*] [**COLOR** *color*] [, **SIZE** *size*] [, **STYLE** *style*]

**Operation** The attributes for graphics lines are defined.

**Notes** This statement does not draw any lines, or change the display of lines previously drawn. It only sets the attributes for lines drawn in the future. Lines are drawn with the **FILL** and **PLOT** series of statements.

The line color specified here is used by all of the VDI statements that draw lines. The line **STYLE** specified here is used only by the **PLOT** statement.

The **COLOR** attribute is device dependent due to the nature of the attribute. The three basic display types use the following color codes:

**Monochrome Displays**

Color	Code
Black	0
White	1

**Color Displays**








Color	Code	Color	Code
Black	0	Red	4
Blue	1	Magenta	5
Green	2	Yellow	6
Cyan	3	White	7

**VGA Color Displays**

Color	Code	Color	Code	Color	Code	Color	Code
Black	0	Red	4	Int Grey	8	Int Red	12
Blue	1	Magenta	5	Int Blue	9	Int Magenta	13
Green	2	Yellow	6	Int Green	10	Int Yellow	14
Cyan	3	White	7	Int Cyan	11	Int White	15

The number of line sizes available is device dependent. Most bit-mapped devices, such as the main console, have only one size.

There are at least seven line STYLES:

Style	Line	Sample
1	Solid	
2	Short dashes	
3	Dotted	
4	Dash, dot	
5	Long dashes	
6	Dash, dot, dot	
7	Small dots	

The initial values for line attributes are: STYLE 1 and SIZE 1. The initial value for COLOR is the maximum code allowed by the device (1 for monochrome displays, 7 for color displays, 15 for VGA color displays). These attributes will be used unless they are changed by a [SET LINE](#) statement. Line colors may be specified at the time they are plotted.

When multiple channels are opened to VDI devices, each channel maintains separate line styles and colors.

#### Restrictions

Invalid parameters do not cause errors. However, the specific invalid parameter is treated as a request for the default specification. For example, requesting STYLE 20 causes STYLE 1 to be used.

The *channel* number must refer to an open channel. That channel, or the default VDI1 device, must be graphics capable or the trappable error message number 37 “Graphics not available.” is reported.

#### See also

[FILL](#), [FILL BAR](#), [FILL CIRCLE](#), [FILL PIE](#), [PLOT](#), [PLOT ARC](#), [PLOT BAR](#), [PLOT CIRCLE](#), [PLOT PIE](#)

#### Examples

```
1000 SET LINE STYLE 2, COLOR 7
1010
1020 PLOT BAR 4000,4000;20000,10000
```

# SET MARKER Statement

SET MARKER defines the color, size, and style of markers drawn with the VDI [PLOT](#) statement.

SET MARKER [*#channel:*] [**COLOR** *color*] [, **SIZE** *size*] [, **STYLE** *style*]

**Operation**            The attributes for markers, or plotted points, are defined.

**Notes**                This statement does not draw any objects, or change the display of objects previously drawn. It only sets the attributes for markers drawn in the future. Markers are only drawn with [Mode 1](#) of the [PLOT](#) statement (plot a point) and with the [VDI](#) statement primitive, polymark.

The COLOR attribute is device dependent due to the nature of the attribute. The three basic display types use the following color codes:

Monochrome Displays

Color	Code
Black	0
White	1

Color Displays

Color	Code	Color	Code
Black	0	Red	4
Blue	1	Magenta	5
Green	2	Yellow	6
Cyan	3	White	7

VGA Color Displays

Color	Code	Color	Code	Color	Code	Color	Code
Black	0	Red	4	Int Grey	8	Int Red	12
Blue	1	Magenta	5	Int Blue	9	Int Magenta	13
Green	2	Yellow	6	Int Green	10	Int Yellow	14
Cyan	3	White	7	Int Cyan	11	Int White	15

The SIZE for markers ranges from small to large. This SIZE parameter is not coded; that is, the size specified is the actual size of the width of the marker displayed. For example, a SIZE of 800 will be 800 units wide. The range of valid values is 500—2000, inclusive.

Statements

There are at least five marker STYLES:

Style	Marker	Sample
1	Dot	●
2	Plus sign	+
3	Asterisk	✱
4	Circle	○
5	X	✕

The initial values for marker attributes are: STYLE 1 and SIZE 1. The initial value for COLOR is the maximum code allowed by the device (1 for monochrome displays, 7 for color displays, 15 for VGA color displays). These attributes will be used unless they are changed by a SET MARKER statement. Marker colors may be specified at the time they are plotted.

When multiple channels are opened to VDI devices, each channel maintains separate marker colors, styles and sizes.

**Restrictions** Invalid parameters do not cause errors. However, the specific invalid parameter is treated as a request for the default specification. For example, requesting STYLE 55 causes STYLE 1 to be used.

The *channel* number must refer to an open channel. That channel, or the default VDI1 device, must be graphics capable or the trappable error message number 37 “Graphics not available.” is reported.

**See also** [PLOT](#)

**Examples**

```
2000      SET LINE STYLE 1, COLOR 7
2010
2020      PLOT 1000,22000;1000,1000;22000,1000
2030
2040      SET LINE STYLE 2
2050
2060      PLOT 5000,2200;10000,15000;15000,8900;20000,12000
2070
2080      SET MARKER STYLE 4, SIZE 100, COLOR 4
2090
2100      PLOT 5000,2200
2110      PLOT 10000,15000
2120      PLOT 15000,8900
2130      PLOT 20000,12000
~
```

# SET TEXT Statement

SET TEXT defines the attributes associated with alphanumeric text drawn with the VDI [TEXT](#) statement.

SET TEXT [#channel:] [COLOR color] [, SIZE size] [, ANGLE angle] [, PATH path]  
[, FONT font]

**Operation**      The attributes for text display with the [TEXT](#) statement are defined.

**Notes**            This statement does not draw any text, or change the display of text previously drawn. It only sets the attributes for text drawn in the future. Only the [TEXT](#) statement draws text.

The sequence of the attribute specifications is irrelevant.

The COLOR attribute is device dependent due to the nature of the attribute. The three basic display types use the following color codes:

Monochrome Displays

Color	Code
Black	0
White	1

Color Displays

Color	Code	Color	Code
Black	0	Red	4
Blue	1	Magenta	5
Green	2	Yellow	6
Cyan	3	White	7

VGA Color Displays

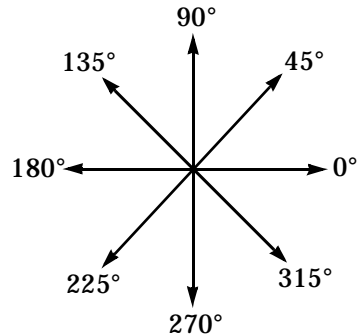
Color	Code	Color	Code	Color	Code	Color	Code
Black	0	Red	4	Int Grey	8	Int Red	12
Blue	1	Magenta	5	Int Blue	9	Int Magenta	13
Green	2	Yellow	6	Int Green	10	Int Yellow	14
Cyan	3	White	7	Int Cyan	11	Int White	15

Statements

The SIZE for text is any one of 4 sizes:

Size	Display
1	Single height, single width
2	Single height, double width
3	Double height, single width
4	Double height, double width

The ANGLE and PATH for text is any one of eight angles:



The value specified for the ANGLE or PATH is specified in degrees or radians, depending upon the current setting of the [OPTION RADIAN](#) or [OPTION DEGREE](#) statement.

Text angles refer to both the direction of the display of each character in a string of characters and to the orientation of each the characters on the page. Text paths refer only to the direction of the display of the text. For example, compare two displays for text with an ANGLE of 45° and then a PATH of 45°

Angle 45°      Path 45°

Text PATHs and ANGLEs may be combined for unusual effects. For example, text displayed with an angle of 90° and a path of 0°

ΩαεζΗφ

The FONT for text is always 1, unless the VDI device is a scaleable font laser printer, which may have more fonts available.

The initial value for text attributes are: SIZE 1, ANGLE 0, PATH 0, and FONT 1. The initial value for COLOR is the maximum code allowed by the

device (1 for monochrome displays, 7 for color displays, 15 for VGA color displays). These attributes will be used unless they are changed by a SET TEXT statement. Text colors may be specified at the time they are drawn.

When multiple channels are opened to VDI devices, each channel maintains separate text colors, styles, sizes, angles and paths.

**Restrictions** Invalid parameters do not cause errors. However, the specific invalid parameter is treated as a request for the default specification. For example, requesting PATH 20 causes PATH 0 to be used.

The *channel* number must refer to an open channel. That channel, or the default VDI1 device, must be graphics capable or the trappable error message number 37 “Graphics not available.” is reported.

**See also** [TEXT](#)

**Examples**

```
2000      OPTION DEGREE
2010
2020      PLOT 5000,26000;5000,5000;26000,5000
2030
2040      SET LINE STYLE 2
2060      PLOT 6000,6200;14000,19000;19000,12900;24000,16000
2070
2080      SET MARKER STYLE 2
2090      FOR I% = 7000 TO 24000 STEP 2000 ! Y-axis hash marks
2100          PLOT 5000,I%
2110          NEXT
2120
2130      FOR I% = 9000 TO 24000 STEP 5000 ! X-axis hash marks
2140          PLOT I%,5000
2150          NEXT
2160
2170      SET TEXT COLOR 7, SIZE 4, ANGLE 90
2180      TEXT 3000,12000;"Y-Axis"
2190      SET TEXT ANGLE 0
2200      TEXT 10000,2000;"X-Axis"
2210
2220      SET MARKER STYLE 4, SIZE 100, COLOR 4
2230
2240      PLOT 9000,6200
2250      PLOT 14000,19000
2260      PLOT 19000,12900
2270      PLOT 24000,16000
```

Statements

---

# SGN Function

SGN returns the sign of the value of an expression.

SGN( *numeric-expression* )

**Operation**      The sign (+1, 0, or -1) of the value of *numeric-expression* is returned.

## Examples

<i>The field P% is</i>	3780	SELECT <b>SGN (P%)</b>
<i>checked to deter-</i>	3790	CASE 1
<i>mine if it is nega-</i>	3800	IF P\$(P%)
<i>tive, positive, or</i>	3810	P.OKAY% = TRUE%
<i>zero.</i>	3820	ELSE P.OKAY% = FALSE%
	3830	IFEND
<i>P% is used to index</i>	3840	CASE -1
<i>into an array. Not</i>	3850	IF P\$(-P%)
<i>only does a nega-</i>	3860	P.OKAY% = FALSE%
<i>tive P% have to be</i>	3870	ELSE P.OKAY% = TRUE%
<i>made positive for</i>	3880	IFEND
<i>indexing, but the</i>	3890	OTHERWISE
<i>logic is also</i>	3900	P.OKAY% = TRUE%
<i>reversed in that sit-</i>	3910	CEND
<i>uation.</i>		



# SHARED Statement

SHARED declares certain variables and arrays used in a subprogram to be *global in scope* with the main program and other program segments.

SHARED    *variable-list*

*variable-list*

»

*variable-name*[, *variable-list*]  
*array*[, *variable-list*]

*array*

»

*array-name*( *subscript* )  
*array-name*( *subscript*<sub>1</sub>, *subscript*<sub>2</sub> )

**Operation**        The variables declared in the *variable-list* are treated as shared variables and arrays.

**Notes**             The purpose of the SHARED statement is to allow a subprogram to use variables and arrays that are defined in the main program, another subprogram or a user-defined function and to allow those other program segments to use the variables and arrays defined in this subprogram.

**Dimensioning**    Arrays dimensioned with this statement may be one-dimensional or two-dimensional.

One-dimensional array:

0	1	2	3	4
---	---	---	---	---

Two-dimensional array:

0,0	0,1	0,2	0,3	0,4
1,0	1,1	1,2	1,3	1,4
2,0	2,1	2,2	2,3	2,4

The number of elements allocated for an array depends upon the [OPTION BASE](#) statement. When [OPTION BASE 0](#) is in effect (default), one is added to each of the dimension sizes specified to account for the zero indices.

For example:

```
OPTION BASE 0
SHARED A(5)           ! 6 elements allocated
SHARED B(12)          ! 13 elements allocated
SHARED C(2,4)         ! 15 elements allocated

OPTION BASE 1
SHARED A(5)           ! 5 elements allocated
SHARED B(12)          ! 12 elements allocated
SHARED C(2,4)         ! 8 elements allocated
```

The number of elements allocated to a SHARED array is retained throughout the entire program. Each program segment declaring the variable-list must dimension the arrays the same as the global declaration and the same as any other SHARED declaration of the array name.

## Defaults

In a user-defined function definition, all variables and arrays referenced, other than the arguments to the function itself, are SHARED by default. The [LOCAL](#) statement must be used to declare that a variable or array is not shared with the main program.

In a **subprogram** definition, all variables and arrays are [LOCAL](#) by default. The SHARED statement must be used to declare that a variable or array is shared with the main program or other program segments.

## Restrictions

The SHARED statement must be specified before any executable statements in the subprogram or user-defined function.

The SHARED statement must appear on a line by itself (it may only be followed by a [REM](#) statement on the same line).

The SHARED statement is only effective when used in a subprogram. When used in the main program or in a user-defined function, the SHARED statement has no effect on variables and is the same as a DIM statement for the arrays specified in the *variable-list*.

The formal arguments to a subprogram cannot be SHARED.

Variables and arrays shared with a subprogram or user-defined function that are used in the main program area must be declared as SHARED variables and arrays in the main program area.

## See also

[COMMON](#), [DIM](#), [LOCAL](#), [PUBLIC](#), [REDIM](#), [STATIC](#)

## Examples

<i>This subprogram</i>	900000	SUB DISPLAY.PRINT.STATUS
<i>uses three types of</i>	900010	
<i>data: local, static,</i>	900020	STATIC LAST.PAGE% ! Last page number displayed
<i>and shared.</i>	900030	STATIC LAST.CUSTOMER\$ ! Last cust code displayed
	900040	<b>SHARED</b> PAGE.NUMBER% ! Current page being printed
	900050	<b>SHARED</b> CUSTOMER.CODE\$ ! Customer being printed
<i>Local storage is</i>	900060	<b>SHARED</b> STATUS.WINDOW% ! Window for status display
<i>used for the</i>	900070	
<i>PRIOR.WINDOW%</i>	900080	WINDOW STATUS PRIOR.WINDOW% ! Save prior window number
<i>variable; static</i>	900090	IF NOT STATUS.WINDOW% ! Status window not defined?
<i>variables are used</i>	900100	WINDOW OPEN ADDROF(STATUS.WINDOW%), 15, 10, 20, 5;
<i>for LAST.PAGE%</i>		FRAME SINGLE, COLOR 7, 1; TITLE " PRINT STATUS ",
<i>and LAST.CUS-</i>		TOP CENTER, COLOR 7, 1; COLOR 7, 1; SELECT
<i>TOMER\$; and</i>	900110	ELSE WINDOW SELECT STATUS.WINDOW% ! Select proper window
<i><b>SHARED</b> vari-</i>	900120	IFEND
<i>ables are used for</i>	900130	
<i>PAGE.NUMBER%,</i>	900140	IF CUSTOMER.CODE\$<>LAST.CUSTOMER\$
<i>CUSTOMER.CODE\$</i>	900150	PRINT AT(15,2);CUSTOMER.CODE\$;
<i>and STATUS.WIN-</i>	900160	CUSTOMER.CODE\$ = LAST.CUSTOMER\$
<i>DOW%.</i>	900170	IFEND
	900180	
<i>These shared vari-</i>	900190	IF PAGE.NUMBER%<>LAST.PAGE%
<i>ables are defined</i>	900200	PRINT AT(15,4);FORMAT\$(PAGE.NUMBER%,"####");
<i>and used in the</i>	900210	LAST.PAGE% = PAGE.NUMBER%
<i>main program.</i>	900220	IFEND
	900230	
	900240	WINDOW SELECT PRIOR.WINDOW% ! Reselect prior window
	900250	
	900260	END SUB

---

## SIN Function

SIN returns the *sine* of an angle.

SIN( *numeric-expression* )

**Operation**      The *numeric-expression* is interpreted as an angle expressed in degrees or radians, depending upon the status of the current **OPTION** DEGREE or **OPTION** RADIAN. The sine of that angle is computed and returned.

**Notes**            Sines have values that range from -1 to +1.

**See also**        Other trigonometric functions

**Examples**

```
10      OPTION PROMPT "", DEGREE
20
30      INPUT "Enter angle in degrees: ",MY.ANGLE
40      PRINT
50
60      MY.ANGLE.MIN = FP(MY.ANGLE)*60      ! Determine minutes
70      MY.ANGLE.SEC = FP(MY.ANGLE.MIN)*60   ! and seconds
80
90      MY.ANGLE$ = STR$(IP(MY.ANGLE))&CHR$(248)&
           STR$(IP(MY.ANGLE.MIN))&"' "&
           STR$(ROUND(MY.ANGLE.SEC,0))&" " " "
100
110     PRINT USING "The sine of 'e is #.#####",MY.ANGLE$,
           SIN(MY.ANGLE)
```

Enter angle in degrees: 76.23

The sine of 76°13'48" is 0.97126

Statements

---

## SINH Function

SINH returns the *hyperbolic sine* of a number.

**SINH**( *numeric-expression* )

**Operation**      The hyperbolic sine of *numeric-expression* is computed and returned.

**See also**      Other trigonometric functions

**Examples**

```
10      OPTION PROMPT ""
20
30      INPUT "Enter area: ",AREA
40      PRINT
50
60      PRINT "The hyperbolic sine of";AREA;"is";SINH(AREA)
```

**Enter area: 2.3**

**The hyperbolic sine of 2.3 is 4.936961805545**

# SLEEP Statement

SLEEP suspends execution of the program for a period of time.

**SLEEP** *seconds*

**Operation** Execution of the program is suspended for the designated amount of time.

**Notes** Although the *seconds* must be a positive value (negative values are merely ignored), it may be a numeric value as small as .001 seconds or as large as 2,147,483.647 seconds (24.85 days)!

While a program is “sleeping,” its execution is suspended until the time expires, unlike a [FOR-NEXT](#) timing loop, which uses all of the processing time available.

When a SLEEP is performed, nothing will awaken the program except a program abort ([Break](#), [Q](#)). Any events or key presses programmed to be trapped with the [ON EVENT](#) or [ON KEY](#) will not take effect until the sleep time has elapsed. For this reason, a SLEEP should not be programmed that will sleep longer than you are willing to accept.

To SLEEP for long periods of time when there are [ON EVENT](#) or [ON KEY](#) traps defined, the recommended method is to use a [FOR-NEXT](#) program structure that repeats a SLEEP statement with a shorter sleep time.

The sleep time specified (*seconds*) is the minimum amount of time that program execution will pause. Because of the multiuser and multitasking design of the THEOS operating environment, it may be several milliseconds after the elapsed time before this program gets activated again.

**Restrictions** The *seconds* value should be a positive value.

**See also** [WAIT](#), [WAIT EVENT](#)

## Examples

Display message	10020	PRINT AT\$(15,24); "CUSTOMER RECORD DELETED";
and allow time to	10030	<b>SLEEP 3</b>
read the message.	10040	PRINT AT\$(15,24); CRT\$( "EOL" );
	~	

---

# SPACE\$ Function

SPACE\$ returns a string of spaces.

SPACE\$( *length* )

**Operation**            A string of *length* spaces is created and returned.

**See also**             [RPT\\$](#)

**Examples**

<i>This first line specifies a LINPUT USING mask of 30 spaces. This allows the operator to enter a maximum of 30 characters in response.</i>	1000 LINPUT USING <b>SPACE\$(30)</b> ,ANSWER\$
<i>The second line displays two fields, separated by 10 spaces.</i>	2000 PRINT AT\$(5,10) ;ANSWER\$; <b>SPACE\$(10)</b> ;MESSAGE\$

Statements

---

## SQR Function

SQR returns the *square root* of an expression.

SQR( *numeric-expression* )

**Operation**      The square root of the value of *numeric-expression* is computed and returned.

**Restrictions**    The square root of negative numbers is invalid and the SQR function of a negative value will return a 0.

**Examples**

```
10 INPUT "Enter area of circle",AREA
20
30 PRINT "Radius of circle with area";AREA;"is";SQR(AREA/PI)
```



# STATIC Statement

The STATIC statement declares that certain variables or arrays are *local in scope* and *static in duration*.

<b>STATIC</b> <i>variable-list</i>	
<hr/>	
<i>variable-list</i>	» <i>variable-name</i> [, <i>variable-list</i> ] <i>array</i> [, <i>variable-list</i> ]
<i>array</i>	» <i>array-name</i> ( <i>subscript</i> ) <i>array-name</i> ( <i>subscript</i> <sub>1</sub> , <i>subscript</i> <sub>2</sub> )

**Operation** The variables declared in the *variable-list* are marked as local variables or local arrays. The contents of the variables or arrays are not initialized.

Within the subprogram or function, references to variable or array names that are specified in the *variable-list* refer to these local variables, not to variables of the same name in the main program storage area.

Upon exit from the function or subprogram the variables are not cleared.

**Notes** The purpose of the STATIC statement is twofold. It allows a subprogram or user-defined function to use variable and array names that are *local in scope* to the subprogram or user-defined function. This means that the subprogram or function can use variable and array names that might conflict with names used in the main program or other program segments without affecting or using the value of the variable or array in the other program segments.

STATIC also declares the variables and arrays to be *static in duration*. That is, static variables and arrays retain their values from one invocation of the function or subprogram to the next.

The STATIC statement is fully effective when used in a user-defined function definition. It can declare variables and arrays to be both local in scope and static in duration.

When used in a subprogram, variables and arrays are local in scope by default. The STATIC statement declares the variables and arrays to also be static in duration.

When used in the main program the `STATIC` statement has no effect on variables and is the same as a `DIM` statement for the arrays specified in the *variable-list*.

**Dimensioning** Arrays may be one-dimensional or two-dimensional.

One-dimensional array: 

0	1	2	3	4
---	---	---	---	---

Two-dimensional array: 

0,0	0,1	0,2	0,3	0,4
1,0	1,1	1,2	1,3	1,4
2,0	2,1	2,2	2,3	2,4

The number of elements allocated for an array depends upon the `OPTION BASE` statement. When `OPTION BASE 0` is in effect (default), one is added to each of the dimension sizes specified to account for the zero indices.

For example:

```
OPTION BASE 0
STATIC A(5)           ! 6 elements allocated
STATIC B(12)          ! 13 elements allocated
STATIC C(2,4)         ! 15 elements allocated

OPTION BASE 1
STATIC A(5)           ! 5 elements allocated
STATIC B(12)          ! 12 elements allocated
STATIC C(2,4)         ! 8 elements allocated
```

**Defaults** Variables or arrays are never static in duration by default. However, variables and arrays are sometimes local in scope by default.

In a **subprogram** definition, all variables and arrays are `LOCAL` by default. The `SHARED` statement must be used to declare that a variable or array is shared with the main program.

In a **user-defined function** definition, all variables and arrays are `SHARED` by default. The `LOCAL` statement must be used to declare that a variable or array is not shared with the main program.

**Restrictions** Static variables must be declared with this statement prior to their usage in a program.

The `STATIC` statement must appear on a line by itself (it may only be followed by a `REM` statement on the same line).

**See also** `COMMON`, `DIM`, `LOCAL`, `PUBLIC`, `REDIM`, `SHARED`

## Examples

*This subprogram is used to display the printing status of a report generation program. It makes use of the properties of a subprogram by isolating its variables from the main program code except for the three **SHARED** variables.*

```
900000 SUB DISPLAY.PRINT.STATUS
900010
900020 STATIC LAST.PAGE%           ! Last page number displayed
900030 STATIC LAST.CUSTOMER$      ! Last cust code displayed
900040 SHARED PAGE.NUMBER%        ! Current page being printed
900050 SHARED CUSTOMER.CODE$     ! Customer being printed
900060 SHARED STATUS.WINDOW%     ! Window for status display
900070
900080 WINDOW STATUS PRIOR.WINDOW% ! Save prior window number
900090 IF NOT STATUS.WINDOW%       ! Status window not defined?
900100     WINDOW OPEN ADDROF(STATUS.WINDOW%), 15, 10, 20, 5;
          FRAME SINGLE, COLOR 7, 1; TITLE " PRINT STATUS ",
          TOP CENTER, COLOR 7, 1; COLOR 7, 1; SELECT
900110 ELSE WINDOW SELECT STATUS.WINDOW% ! Select proper window
900120 IFEND
900130
900140 IF CUSTOMER.CODE$<>LAST.CUSTOMER$
900150     PRINT AT(15,2);CUSTOMER.CODE$;
900160     CUSTOMER.CODE$ = LAST.CUSTOMER$
900170 IFEND
900180
900190 IF PAGE.NUMBER%<>LAST.PAGE%
900200     PRINT AT(15,4);FORMAT$(PAGE.NUMBER%,"###");
900210     LAST.PAGE% = PAGE.NUMBER%
900220 IFEND
900230
900240 WINDOW SELECT PRIOR.WINDOW%   ! Reselect prior window
900250
900260 END SUB
```

*Two variables are declared as **STATIC** because they are work variables local to this subprogram but their values need to be retained from one call to the subprogram to the next.*

---

## STOP Statement

The STOP statement halts execution of the program.

- 1 **STOP**
  - 2 **STOP** *expression*

### Operation

The STOP statement operates differently between the interpreter and compiled forms of a program. It is a statement intended to be used to aid in testing and debugging of programs in the interpreter.

When executing a program with the interpreter, the STOP statement interrupts program execution without ending it. Unlike the [END](#) statement, files and windows are not closed. When a STOP is executed, the command mode of the interpreter is entered.

In a compiled program a STOP statement terminates execution of a program, similar to [Mode 2](#) of the [QUIT](#) statement with the *return code* set to 253. Files are closed, the program stack is cleared, and subtasks are deactivated.

**Mode 1**—Halts execution of the program and displays a stop message. Compiled programs display “Stop” and interpreted programs display “Stop at line nnnnnn.”

**Mode 2**—The *expression* is evaluated, program execution is halted, and a stop message displays that includes the value of the expression. Compiled programs display “Stop” followed by the value of the expression on the next line. Interpreted programs display “Stop *expression* at line nnnnnn.” The *expression* may be a numeric or string expression.

### Notes

Use the [Mode 2](#) form when there are multiple stop points in a program. Specify an expression that indicates the location of the STOP in the program or includes the value of some critical variables in the program.

When running a program with the interpreter, the [CONTINUE](#) command may be used to resume execution with the statement following the STOP statement.

### See also

[END](#), [QUIT](#) statements, [BREAK](#), [CONTINUE](#), [STEP](#) commands

## Examples

*Some **STOP** statements have been added in this code that halt the program after two significant routines have executed. The **STOP** messages are defined to identify the position of the **STOP** in the program.*

*Here, the **STOP** statements are added to identify which **CASE** is being executed.*

```
1000 GOSUB SETUP ! Program initialization
1005 STOP "after SETUP"
1010
1020 WINDOW SELECT 1
1030
1040 WHILE NOT END.RUN%
1050
1060     GOSUB GET.KEY
1070     STOP "after GET.KEY"
1080
1090     IF NOT END.RUN% THEN GOSUB ENTER.DATA
1100     IF NOT END.RUN% THEN GOSUB MODIFICATIONS
1110
1120 WEND
~

10140 SELECT
10150     CASE INP=10           ! Down arrow = next record
10155         STOP "INP = "&STR$(INP)
10160         IF KEY$ THEN READ #1,KEY$: X$
10170         GOSUB READ.NEXT.CUSTOMER
10180     CASE INP=11         ! Up arrow = prev record
10185         STOP "INP = "&STR$(INP)
10190         IF KEY$ THEN READ #1,KEY$: X$
10200         GOSUB READ.PREV.CUSTOMER
10210     CASE (INP=13 OR INP=0) AND KEY$<>" "
10215         STOP "INP = "&STR$(INP)
10220         REC$(1) = KEY$
10230         GOSUB READ.CUSTOMER
10240     OTHERWISE
10245         STOP "INP = "&STR$(INP)
10250         PRINT CRT$("BELL");
10260         VALID% = FALSE%
10270     CEND
10275 STOP "INP = "&STR$(INP)
~
```

# STR\$ Function

STR\$ returns a trimmed string representing the value of a number.

STR\$( *numeric-expression* )

**Operation**      The *numeric-expression* is evaluated and the resulting value is returned as a string. No leading or trailing spaces are added as would be the case when PRINTing the value.

**See also**        VAL

**Examples**        The following code is used to format a date display.

```

~
The day of the week      3070      DAYNAME$ = DAY.NAMES$(1+MOD(DAY(DATE$(0)),7))
name is selected         3080
from an array.

Determine the date       3090      SELECT DATE$(0)[3:3]   ! Look at date delimiter
format.

Use the month num-       3100      CASE "/"              ! Format is mm/dd/yy
ber to select the        3110      C.DATE$ = DAYNAME$&MONTH.NAMES$(
month name; for-          VAL(DATE$(0)[1:2]))&
mat the day number       STR$(VAL(DATE$(0)[4:5]))
with the VAL and         3120      CASE '-'              ! Format is dd-mm-yy
STR$ functions to        3130      C.DATE$ = DAYNAME$&MONTH.NAMES$(
remove any lead-          VAL(DATE$(0)[4:5]))&
ing zero.                STR$(VAL(DATE$(0)[1:2]))
                          3140      CASE "."              ! Format is yy.mm.dd
                          3150      C.DATE$ = DAYNAME$&MONTH.NAMES$(
                              VAL(DATE$(0)[4:5]))&
                              STR$(VAL(DATE$(0)[7:8]))
                          3160      CEND
~
```

Statements

## STRTIME\$ Function

STRTIME\$ returns a formatted date and/or time.

**STRTIME\$( *mask-string*, *day-number*, *second-number* )**

**Operation**      The date and time specified by *day-number* and *second-number* are used to create a formatted date/time string according to the specifications in *mask-string*.

The *mask-string* may contain the following special symbols:

Component	Mask	Meaning	Example output
Date & Time	%C	Date and time in standard format. <sup>1</sup>	"07/04/2000 13:30:45"
	%C	Date and time in standard long format. <sup>2</sup>	"4 Jul 2000 13:30"
Date	%x	Date in standard short format with 4-digit year.	"07/04/2000"
	%D	Date in standard short format with 2-digit year.	"07/04/00"
Year	%y	Year number, 2-digit.	"00"
	%Y	Year number, 4-digit.	"2000"
Month	%b	Abbreviated month name. <sup>2</sup>	"Jul"
	%B	Full month name. <sup>3</sup>	"July"
	%h	Abbreviated month name. <sup>2</sup>	"Jul"
	%m	Month number.	"07"
Day	%a	Abbreviated weekday name. <sup>4</sup>	"Tue"
	%A	Full weekday name. <sup>5</sup>	"Tuesday"
	%d	2-digit day number within month.	"04"
	%e	Day number within month. Single-digit days are preceded with a space.	" 4"
	%j	Julian day number of year.	"186"
	%J	Day number within month. Single-digit days are not preceded with space.	"4"
	%w	Weekday number using Sunday as 1st day of week	"2"

Component	Mask	Meaning	Example output
Week	%U	Week number of year, using Sunday as 1st day of week.	"27"
	%W	Week number of year, using Monday as 1st day of week.	"27"
Time	%r	Time in 12-hour format. <sup>6</sup>	"01:30:45 PM"
	%R	Time in 24-hour format, without seconds.	"13:30"
	%T	Time in 24-hour format, with seconds.	"13:30:45"
	%X	Time in 24-hour format, with seconds.	"13:30:45"
	%p	AM or PM. <sup>6</sup>	"PM"
	%P	a or p. <sup>6</sup>	"p"
	%Z	Timezone name. <sup>7</sup>	"PDT"
Hour	%H	Hour number, 24-hour clock.	"13"
	%I	Hour number, 12-hour clock. (Code is uppercase "i".)	"01"
	%k	Hour number, 24-hour clock. Single-digits are preceded with space character.	"13"
	%l	Hour number, 12-hour clock. Single-digits are not preceded with space character. (Code is lowercase "L".)	"1"
Minute	%M	Minute number.	"30"
Second	%S	Second number.	"45"
Formatting	%n	Newline character.	
	%t	Tab character.	
	%%	Percent character.	"%"

1. The format of the date is dependent upon the current DATEFORM setting.
2. Month abbreviation is read from message #424.
3. Month name is read from message #261, 262 and 263.
4. Weekday abbreviation is read from message #424.
5. Weekday names is read from message #264 and 265.
6. AM/PM text is read from message #424.
7. Timezones are only set by the system configuration settings.



<b>Notes</b>	<p>The <i>day-number</i> is the number of days since January 1, 1980. This should be a floating point value as an integer expression may be too small. The <i>day-number</i> is typically the return value of a <a href="#">DAY</a> function call.</p> <p>The <i>second-number</i> is the number of seconds since midnight and should also be a floating-point value. It is typically the return value from a <a href="#">SECOND</a> function call.</p> <p>The operation of this function is not affected by the DATEIN and DATEOUT settings.</p>
<b>Defaults</b>	A value of 0 for <i>day-number</i> means that today's system date is used. A value of 0 for <i>second-number</i> means that the current system time is used.
<b>See also</b>	<a href="#">DATE\$</a> , <a href="#">DAY</a> , <a href="#">DTE\$</a> , <a href="#">SECOND</a> , <a href="#">TIME\$</a>
<b>Examples</b>	<pre>PRINT STRTIME\$("Today is %A, %B %J, %Y", DAY(0), 0)  JULIAN% = VAL(STRTIME\$("%j", DAY(0), 0)) X\$ = OVR\$(0, 1, 5, "12/25") X = VAL(STRTIME\$("%j", DAY(X\$), 0)) PRINT USING "There are ZZZ days until Christmas", X-JULIAN%</pre>

# SUB Statement

The SUB statement defines the start of a callable subprogram.

1 SUB subprogram-name

2 SUB subprogram-name( argument-list )

argument-list

» variable-name, argument-list

ADDROF( variable-name ), argument-list

DIM array-name( subscript-variable ), argument-list

DIM array-name( subscript-variable<sub>1</sub>, subscript-variable<sub>2</sub> ),

argument-list

**Operation** The lines of code following the SUB statement, up to and including the **END SUB** statement, are identified as a subprogram.

**Mode 1**—The subprogram *subprogram-name* is defined and when accessed, will be passed no arguments.

**Mode 2**—The subprogram *subprogram-name* is defined and will always be accessed with the number and type of arguments identified in *argument-list*. Refer to the **CALL** statement for information about passing arguments to a subprogram.

**Notes** A subprogram is a section of code that is subordinate to the main program logic of a program. All variables used or referenced in a subprogram are local in scope to the subprogram only, unless they are specifically declared as **SHARED** variables or arrays.

Subprograms may only be accessed with the **CALL** statement and they are terminated with the **END SUB** statement, with control returning to the statement following the **CALL** statement. A **BREAK** statement in a subprogram (outside of any **FOR-NEXT**, **SELECT-CEND** and **WHILE-WEND** statement structures) causes control to transfer to the **END SUB** statement of the subprogram.

An **IOLIST** name declared in the main program area may be used inside of a subprogram.

Line labels declared inside of a subprogram are local to the subprogram.

Subprograms may define subroutines that are local to the subprogram. These local subroutines are defined between the SUB and **END SUB** statements. (Be careful to either branch around the subroutine or to use the **BREAK** statement to transfer control to the **END SUB** statement.)

**Defaults** All variables and arrays referenced inside of a subprogram are local to the subprogram unless explicitly declared as **SHARED** variables and arrays.

**Restrictions** A subprogram must start with a SUB statement and end with one and only one **END SUB** statement.

A subprogram may reference other subprograms and it may even reference itself (recursive call) but a subprogram may not be defined within another subprogram (no nested definitions) nor may a user-defined function be defined within a subprogram.

**See also** **BREAK**, **CALL**, **END SUB**, **INCLUDE**, **LOCAL**, **SHARED**, **STATIC**

### Examples

*This subprogram is used to display the printing status of a report generation program. It makes use of the properties of a subprogram by isolating its variables from the main program code except for the three **SHARED** variables.*

```
900000 SUB DISPLAY.PRINT.STATUS
```

```
900010
```

```
    STATIC LAST.PAGE%      ! Last page number displayed
```

```
900020
```

```
    STATIC LAST.CUSTOMER$ ! Last cust code displayed
```

```
900030
```

```
    SHARED PAGE.NUMBER%    ! Current page being printed
```

```
900040
```

```
    SHARED CUSTOMER.CODE$ ! Customer being printed
```

```
900050
```

```
    SHARED STATUS.WINDOW% ! Window for status display
```

```
900060
```

```
900070
```

*In addition to the two **STATIC** variables declared it uses a variable name that might be used in many places in a program: **PRIOR.WINDOW**.*

```
900080
```

```
    WINDOW STATUS.PRIOR.WINDOW% ! Save prior window number
```

```
900090
```

```
    IF NOT STATUS.WINDOW%      ! Status window not defined?
```

```
900100
```

```
        WINDOW OPEN ADDROF(STATUS.WINDOW%), 15, 10, 20, 5;
```

```
        FRAME SINGLE, COLOR 7, 1; TITLE " PRINT STATUS ",
```

```
        TOP CENTER, COLOR 7, 1; COLOR 7, 1; SELECT
```

```
900110
```

```
    ELSE WINDOW SELECT STATUS.WINDOW% ! Select proper window
```

```
900120
```

```
        IFEND
```

```
900130
```

```
900140
```

```
    IF CUSTOMER.CODE$<>LAST.CUSTOMER$ THEN GOSUB DISPLAY.CUST
```

```
900150
```

```
    IF PAGE.NUMBER%<>LAST.PAGE% THEN GOSUB DISPLAY.PAGE
```

```
900160
```

```
900170
```

```
    WINDOW SELECT PRIOR.WINDOW% ! Reselect prior window
```

```
900180
```

<i>The <b>BREAK</b> statement is used to avoid having the logic “fall into” the subroutine code.</i>	900190	BREAK	! Exit subprogram
	900200		
<i>Notice that the two subroutines are defined prior to the <b>END SUB</b> statement. This means that these subroutines are local to the subprogram and can only be used by this subprogram.</i>	900210	DISPLAY.CUST:	! Local subroutine
	900220		
	900230	PRINT AT\$(15,2);CUSTOMER.CODE\$;	
	900240	CUSTOMER.CODE\$ = LAST.CUSTOMER\$	! Save cust displayed
	900250		
	900260	RETURN	
	900270		
	900280	DISPLAY.PAGE:	! Local subroutine
	900290		
	900300	PRINT AT(15,4);FORMAT\$(PAGE.NUMBER%,"####");	
	900310	LAST.PAGE% = PAGE.NUMBER%	! Save page number displayed
	900320		
	900330	RETURN	
	900340		
	900350	END SUB	! End of subprogram

# SWAP Statement

SWAP exchanges the values of two variables.

SWAP *variable<sub>1</sub>, variable<sub>2</sub>*

**Operation**            The value of *variable1* is exchanged with *variable2*.

**Notes**                The SWAP statement performs the same operation as:

```
TEMP$ = A$
A$ = B$
B$ = TEMP$
```

with the TEMP\$ variable being internal to MultiUser BASIC.

**Restrictions**        The two variables must be of the same type, either both strings, both integers or both numeric.

**See also**            [LET](#)

**Examples**

*This first portion just creates an array of values.*

*Sort the array using MAT SORT and make a copy of the original array.*

*Using the sorted indices, SWAP the original array item with the copy array item.*

*Display the original and sorted arrays.*

10        DIM B\$(12),SB%(12),C%(12)

20

30        FOR I% = 1 TO 12

40            B%(I%) = RND\*1000

50            NEXT

60

70        MAT SB% = SORT(B%) ! Sort indices of B%

80

90        MAT C% = B%                ! Make a copy of B%

100

110        FOR I% = 1 TO 12

120            **SWAP B%(SB%(I%)),C%(I%)**

130            NEXT

140

150        MAT PRINT B%;C%;

```
309 15 1 31 36 242 211 939 904 746 136 133
1 15 31 36 133 136 211 242 309 746 904 939
```

Statements

---

## SYS.ENV\$ Function

SYS.ENV\$ returns a system environment value and, optionally, sets environment values.

- 1

**SYS.ENV\$( *parameter* )**
- 2

**SYS.ENV\$( *parameter*, *string-expression* )**

### Operation

**Mode 1**—Used for system environment parameters that do not require a value. The value of the indicated system environment variable (see table below) is returned.

**Mode 2**—Used for system environment parameters when further specifics are needed (*e.g.*, parameter 8 and 9), or to provide the change-to value. The current value of the system environment parameter (see table below) is returned.

Param	Returns	Argument	Comments
35	Maximum user count		Returns the number of licensed consoles for this system.
36	NWM available		A 'Y' or 'N' value is returned indicating the new Window Manager is available or not.
37	Break,P status	New status (optional)	A '0' or '1' value is returned indicating that print echo is enabled ('1') or suppressed ('0').
38	Break,W status	New status (optional)	A '0' or '1' value is returned indicating that page waits are enabled ('1') or suppressed ('0').

## Notes

The HELP command displays a summary of the SYS.ENV\$ functions.

When the second argument is used, it must be a string; when the parameter is a number, it must also be specified with a string:

```
SYS.ENV$(9, "14")
```

A second argument may be specified every time the SYS.ENV\$ function is used, even if the parameter does not require or use it.

## Examples

```
920010    CUR.WAIT$ = SYS.ENV$(38, "1") ! Make sure page wait enabled
920020    CSI "LIST SOME.FILE"
920030    X$ = SYS.ENV$(38, CUR.WAIT$) ! Return to prior status
```

---

## SYSTEM Statement

SYSTEM executes a compiled program or EXEC and returns to the calling program.

**SYSTEM** *command-line-exp*

**Operation** The THEOS Command String Interpreter (CSI) is invoked with the *command-line*. After the command has finished execution control returns to this calling program.

**Notes** This statement is identical to the [CSI](#) statement except that the [CSI](#) statement closes all files and SYSTEM does not close any files.

The *command-line* may specify any compiled program (C language, BASIC language, assembly language) or an EXEC program.

The first character of *command-line* may be the angle bracket ">" which causes the command line to display on the console. Without this angle bracket the command is invoked "silently."

Although the program or EXEC may change directories or accounts, it should not. The current directory and account is not maintained nor restored when the SYSTEM statement is executed. If accounts or directories are changed they should be restored before returning to this program.

Actions	Statements						
	CHAIN	LINK	RUN	CSI	SYSTEM	QUIT	END
Close all files	✓		✓	✓		✓	✓
Clear all non-COMMON data	✓	✓	✓			✓	✓
Clear all COMMON data			✓			✓	✓
Terminate open program structures	✓	✓	✓			✓	✓
Clear the current ON ERROR trap	✓	✓	✓			✓	✓
Clear ON KEY, ON EVENT and ON MOUSE	✓	✓	✓			✓	✓
Begin execution of another program	✓	✓	✓	✓	✓	✓	
Return to calling program				✓	✓		
Reset all OPTIONS						✓	
Close windows							✓
Deactivate subordinate subtasks				✓	✓	✓	✓



**Restrictions**      The `SYSTEM` statement cannot be used in a subtask program. Attempting to do so causes all subtasks to terminate, including this one.

Because your files are not closed by this statement, any record or file locks set by this program are not cleared. It is possible to enter a “deadly embrace” situation where the command invoked needs access to a file locked by this program.

**See also**            [CSI](#)

**Examples**            7070        `SYSTEM "ERASE "&WORKFILE$&" (NOTYPE"`  
                             ~

---

# TAB Function

TAB returns no value, but when output to the console or printer moves the cursor or print head to a column position.

TAB( *column-number* )

**Operation** Spaces are output to advance the current column position to *column-number*.

**Notes** When the current column position is already beyond *column-number*, a new line is started and *column-number*-1 spaces are output.

**Restrictions** TAB can only be used in a [PRINT](#) statement. The syntax analyzer does not allow the TAB function in any statement other than a [PRINT](#).

**See also** [POS](#), [SPACE\\$](#) functions and [PRINT](#) statement

**Examples**

```
~
1010      FOR I%=I% TO PAGE(14) -5
1020          PRINT #14: TAB(3);CUST.ID$(I%);TAB(15); CUSTOMER$(I%)
1030          NEXT
~
```

---

# TAN Function

TAN returns the *tangent* of an angle.

TAN( *numeric-expression* )

**Operation**      The *numeric-expression* is interpreted as an angle expressed in degrees or radians, depending upon the status of the current [OPTION DEGREE](#) or [OPTION RADIAN](#). The tangent of that angle is computed and returned.

**Notes**            Refer to the [COS](#) function for a description of trigonometric relationships.

Tangents have values that range from  $-\infty$  to  $+\infty$ .

**Restrictions**    The value of *numeric-expression* must not be an odd multiple of  $\frac{\pi}{2}$  radians (for example,  $\frac{\pi}{2}$ ,  $\frac{3\pi}{2}$ ,  $\frac{5\pi}{2}$  radians *etc.*, or 90°, 270°, 450°, *etc.*) as these values are outside this functionís domain and cause TAN to return meaningless values.

**See also**        Other trigonometric functions, [OPTION DEGREE](#) and [RADIAN](#) statement

**Examples**

```
1000      OPTION DEGREE, PROMPT ""
1010
1020      INPUT "Enter an angle in degrees:: ",DEGREES
1030
1040      TANGENT = TAN(DEGREES)
1050
1060      PRINT "The tangent of ";STR$(DEGREES);CHR$(248);
            " is";TANGENT
```

Statements

---

## TANH Function

TANH returns the *hyperbolic tangent* of a number.

TANH( *numeric-expression* )

**Operation**      The hyperbolic tangent of *numeric-expression* is computed and returned.

**See also**        Other trigonometric functions

**Examples**

```
10      OPTION PROMPT " "  
20  
30      INPUT "Enter area: ",AREA  
40      PRINT  
50  
60      PRINT "The hyperbolic tangent of";AREA;"is";TANH(AREA)
```

Enter area: 2.3

The hyperbolic tangent of 2.3 is 0.9800963962661

# TEXT Statement

TEXT draws alphanumeric text on a VDI device.

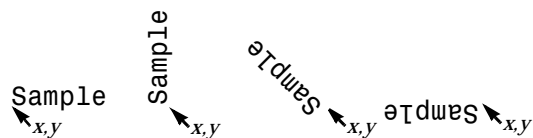
TEXT [#channel:] [color,] x, y, text

**Operation**      Text is drawn on the VDI display, starting at the coordinates *x, y*. The current text size, angle, path, and font are used. The current text color is used unless *color* is specified with this statement.

**Notes**            Refer to the [SET TEXT](#) for a description of the various colors and sizes of text available.

Refer to the MultiUser BASIC Programmer's Guide for a description of VDI devices, statements, and a complete program example.

The coordinates for text refer to the lower left corner of the first character displayed. For example:



Text is monospaced unless the device supports proportional spacing and a proportional spaced font is selected.

The text character background is transparent.

Text displayed beyond the edge of the graphics display page is truncated.

Text drawn at an angle other than horizontal or vertical (i.e., 45°) might appear dim as the same number of pixels for each character are “stretched” to fill a larger area.

**Restrictions**    Invalid parameters do not cause errors. Invalid coordinates are plotted as the maximum value (32767); invalid colors are set to the maximum color code.

The *channel* number must refer to an open channel. That channel, or the default VDI1 device, must be graphics capable or the trappable error message number 37 “Graphics not available.” is reported.

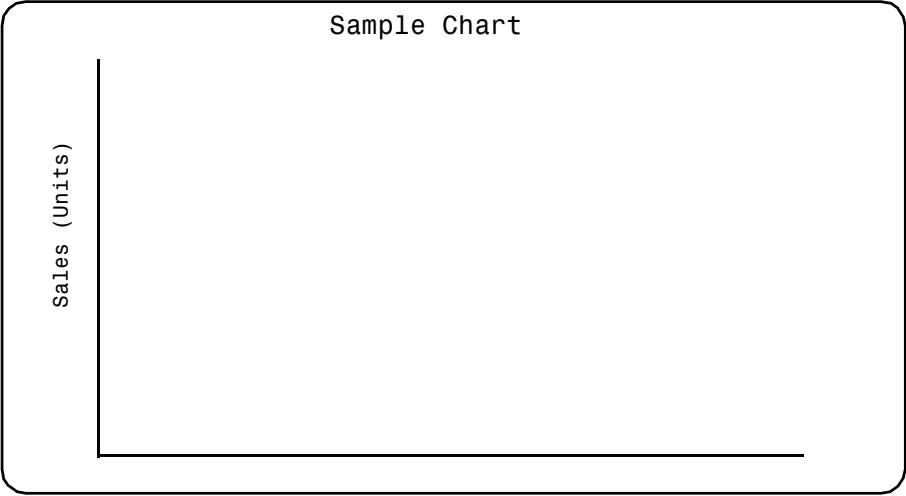
Statements

**Examples**

In the following example code, two items of text are drawn: the chart title and the y-axis legend. Both of these text items are centered in their display areas. In order to center or justify graphics text, the physical size of the text characters must be determined. This is accomplished in the `fn.set.text.size%` function by using the [VDI](#) statement to set and return the size of the text.

	10	OPTION BASE 1, DEGREE
	20	
	30	GOSUB VDI.INIT \ REM Get VDI parameters
	40	
	~	
<i>Draw the chart title text, centered, at the top of the chart. Full page width and height is 32,767.</i>	10000	CHART.TITLE: \ REM Display title, centered
	10010	
	10020	TEXT.WIDTH% = FN.SET.TEXT.SIZE%(4)*LEN("Sample Chart")
	10030	
	10040	TITLE.X% = (32767-TEXT.WIDTH%)/2
	10050	TITLE.Y% = 32767-CHAR.HEIGHT%
	10060	
	10070	TEXT TITLE.X%,TITLE.Y%;"Sample Chart"
	10080	
	10090	RETURN
	10100	
<i>Draw the y axis legend on the left side of the chart, centered. Text is drawn vertically, from bottom to top.</i>	11000	CHART.Y.LEGEND:
	11010	
	11020	TEXT.WIDTH% = FN.SET.TEXT.SIZE%(1)*LEN("Sales (Units)")
	11030	
	11040	SET TEXT ANGLE 90 \ REM Text goes from bottom to top
	11050	
	11060	LEGEND.X% = CHAR.HEIGHT%
	11070	LEGEND.Y% = (32767-TEXT.WIDTH%)/2
	11080	
	11090	TEXT LEGEND.X%,LEGEND.Y%;"Sales (Units)"
	11100	
	11110	RETURN
	~	
<i>This function sets the text size and returns the width of a character in VDI points.</i>	80000	DEF FN.SET.TEXT.SIZE%(S%)
	80010	
	80020	MAT VCONTROL% = (0)
	80030	MAT VIN% = (0)
	80040	VCONTROL%(1) = 12 \ REM Set char size function
	80050	VCONTROL%(2) = 1 \ REM One points in pair
	80060	VPIN%(1,1) = 0
	80070	VPIN%(1,2) = S% \ REM Text size requested
	80080	
	80090	VDI VCONTROL%,VIN%,VPIN%,VOUT%,VPOUT%
	80100	

The V.PIXEL.WIDTH%	80110	CHAR.WIDTH% = VPOUT%(2,1)*V.PIXEL.WIDTH%
and HEIGHT% was	80120	CHAR.HEIGHT% = VPOUT%(2,2)*V.PIXEL.HEIGHT%
computed by the	80130	
VDI.INIT routine. It is	80140	FN.SET.TEXT.SIZE% = CHAR.WIDTH%
the number of VDI	80150	
points in a device	80160	FNEND
pixel.	80170	
	~	
Opens the VDI	90000	VDI.INIT:
device and saves	90010	
some of the key	90020	DIM VCONTROL%(6),VIN%(10),VPIN%(6,2),VOUT%(45),
parameters about		VPOUT%(6,2)
that device.	90030	
	90040	VCONTROL%(1) = 1
	90050	
	90060	VDI VCONTROL%,VIN%,VPIN%,VOUT%,VPOUT%
	90070	
The YADJUST is a	90080	YADJUST = (VOUT%(1)+1)/(VOUT%(2)+1)*VOUT%(4)/VOUT%(5)
value used for	90090	
adjusting the y		
coordinates when		
the device's x and y		
resolutions are not		
the same.		
The V.PIXEL.WIDTH%	90100	V.PIXEL.WIDTH% = 32767/(VOUT%(1)+1)
is the number of	90110	V.PIXEL.HEIGHT% = 32767/(VOUT%(2)+1)
VDI points in a	90120	
device pixel.	90130	RETURN
V.PIXEL.HEIGHT		
% is similar.		



Statements

---

# THEN Statement

THEN is part of a multiple-line [IF-IFEND](#) programming structure. The THEN statement marks the beginning of a series of statements executed when the [IF](#) condition is true.

[THEN] *statement*

**Operation**      The THEN statement marks the beginning of lines that are conditionally executed as part of a multiple-line [IF](#) statement. When the relational-expression of the [IF](#) statement is true, the statements following the [IF](#) statement are executed, up to the matching [ELSE](#) or [IFEND](#) statement; when the expression is false, these statements are skipped and the statements following the [ELSE](#) statement, if present, are executed.

**Notes**            The keyword THEN is not required. In fact, if entered, the MultiUser BASIC editor will remove it. MultiUser BASIC assumes that all statements following a multiple-line [IF](#) statement, up to the matching [ELSE](#) or [IFEND](#) statement, are THEN statements.

**Restrictions**    THEN statements may only be used immediately following an [IF](#) statement.

**See also**         [ELSE](#), [IF](#), [IFEND](#)

## Examples

	~	
	1170	IF TO.PROG\$
Line 1180 is the	1180	<b>LINK TO.PROG\$</b>
THEN statement.	1190	ELSE QUIT
	1200	IFEND



# TIME\$ Function

TIME\$ returns a formatted time-of-day string corresponding to a numeric value of the number of seconds since midnight.

TIME\$( numeric-expression )

**Operation**            The *numeric-expression* is interpreted as the number of seconds since midnight. The corresponding time in HH:MM:SS format is returned. The special value of 0 is used to return the current time of day.

**Restrictions**        When the value of *numeric-expression* is less than 0 or greater than 86,399 (23:59:59) the time string of "00:00:00" is returned.

**See also**             [SECOND](#), [STRTIME\\$](#)

**Examples**            The following section of code is used to determine if the current time of day is during normal work hours. (It might be used to restrict access to game programs in a menu.)

```
~
The current time of 900690 REM offtime is before 8:00, between 12:00 and 13:00,
day, in seconds, is      or after 17:00
computed using          900700
both the SECOND         900710      X = SECOND(TIME$(0))
and the TIME$           900720
functions.
That time is then      900730      IF X<28800.0 OR (X>=43200.0 AND X<=46800.0) OR
compared to the        X>61200.0
times for 8 am,        900740      OFFTIME% = TRUE%
noon, 1 pm, and 5      900750      ELSE OFFTIME% = FALSE%
pm.                   900760      IFEND
The day of the week    900770      IF MOD(DAY(DATE$(0)),7)=0 OR MOD(DAY(DATE$(0)),7)=6
is computed to see     900780      OFFTIME% = TRUE%
if it is a weekend.     900790      IFEND
                      900800
~
```

Statements

---

# TIMER Statement

The `TIMER` statement sets a semaphore flag at a time in the future.

- 1

`TIMER semaphore AT time`
- 2

`TIMER semaphore SYNC interval`
- 3

`TIMER semaphore ELAPSED seconds`

**Operation**      **Mode 1**—Sets *semaphore* on at the designated time of day. If time specifies a time of day that has already passed, the timer is set for tomorrow at that time.

**Mode 2**—Sets *semaphore* on when the system's time of day clock synchronizes with the *interval* specified. For example, an *interval* of 5 means that the timer is set for the next time that is evenly divisible by 5, such as hh:mm:05, hh:mm:10, hh:mm:15, hh:mm:20, etc.

**Mode 3**—Sets *semaphore* on after *seconds* amount of time has elapsed from the current time.

**Notes**              When the timer “goes off,” the timer is cleared. To have it repeat (as is often the case with the `SYNC` form) the timer must be rescheduled.

The *time*, *interval* and *seconds* are all time values that are positive, whole numbers, such as is returned by the [SECOND](#) function.

**See also**          [ON EVENT](#), [WAIT EVENT](#) statements, [SEMAPHORE](#) functions

**Examples**

To see how the following sections of code work, execute the OPERATOR and MENU programs included on the distribution disk.

*The event handler for the timer displays the current time and reschedules the timer.*

```
601000 MINUTE.TIME.OUT:
601010
601020     GOSUB DISPLAY.TIME
601030
601040     TIMER MINUTE.TIMER% SYNC 60 \ REM Reschedule timer
601050
601060     RESUME
~
601200 DISPLAY.TIME: REM Display current time of day on screen
~
```

*During program set-up, the timer event handler and the timer are defined.*

```
900000 SETUP:
~
900580     MINUTE.TIMER% = SEMAPHORE("MINUTE")
~
901290     ON EVENT(MINUTE.TIMER%) GOTO MINUTE.TIME.OUT
901300     TIMER MINUTE.TIMER% SYNC 60 \ REM Once a minute,
        on the minute
~
```

*This routine gets one character from the operator. The operator must respond within a preset amount of time. The **TIMER** statement and the **ON EVENT** statement provide the ability to have timed input.*

```
9480 GET.SEL:
9490
9500     SEL% = 0 \ TIME.OUT% = FALSE% \ REM Init variables
9510
9520     ON EVENT(MENU.TIMER%) GOTO INPUT.TIME.OUT
9530     TIMER MENU.TIMER% ELAPSED WAITTIME%
9540
9550     WAIT #0 \ GET SEL% \ REM Wait and get selection
9560
```

*The **ON EVENT** statement defines where to go when the event occurs; the **TIMER** statement defines when the event occurs.*

```
9570     ON EVENT(MENU.TIMER%) GOTO 0 \ REM Disable timer
9580
9590     IF TIME.OUT% THEN SEL% = 128 \ REM Special code for
        time outs
9600
9610     RETURN
9620
```

*When the timer elapses the **WAIT** is interrupted and **INPUT.TIME.OUT** is invoked. It sets a flag and **RESUMES**. The next statement executed is the **GET** statement which gets nothing.*

```
9630 INPUT.TIME.OUT:
9640
9650     TIME.OUT% = TRUE%
9660
9670     RESUME
~
900980     MENU.TIMER% = SEMAPHORE("TIMER")
~
```

The **ON TIMEOUT** statement provides a more efficient method of performing timed input. Refer to that statement for an example.

---

# TOTAL.WINDOWS Function

TOTAL.WINDOWS returns the maximum number of windows usable on your console.

TOTAL.WINDOWS

**Returns**            The maximum number of windows on the current console.

**Notes**             The maximum number of windows usable on THEOS 3.2 main consoles and Window Manager version 2.1 supported terminals is ten.

$$\text{TOTAL.WINDOWS} = \text{AVAIL.WINDOWS} + \text{USED.WINDOWS}$$

The number of windows for a terminal not supported by Window Manager is one.

**See also**           [AVAIL.WINDOWS](#), [USED.WINDOWS](#)

**Examples**

```
1000      IF TOTAL.WINDOWS<6
1010          PRINT "This program requires Window Manager support"
1020          PRINT "      with at least 6 windows supported."
1030          PRINT
1040          PRINT "Exiting..."
1050          SLEEP 2
1060          QUIT 254
1070      IFEND
```

Statements

# TRIM\$ Function

TRIM\$ returns a string with all leading and trailing spaces removed and all embedded multiple spaces reduced to single spaces.

TRIM\$( *string-expression* )

**Operation** All leading and trailing spaces are removed from *string-expression* and all embedded multiple spaces are reduced to single spaces. The result is returned.

**Notes** The TRIM\$ function is useful for creating a standardized string. For example, to compare an operator's input to a list of choices, the TRIM\$ function removes any extraneous spaces that the operator might have entered. The likelihood of a match is much greater when the TRIM\$ function is used in this situation.

The functions LTRIM\$, TRIM\$, and RTRIM\$ all remove spaces from a string. They differ in where the extra spaces are removed from:

Function	Action
LTRIM\$	Remove all spaces on the left side of the string (leading spaces only).
TRIM\$	Remove all leading and all trailing spaces; reduce all multiple, consecutive, embedded spaces to single spaces.
RTRIM\$	Remove all spaces on the right side of the string (trailing spaces only).

**See also** LTRIM\$, RTRIM\$

## Examples

```
A field entered by 601090      IF TRIM$(FIELD$)
the operator is    601100      FOR SI% = 1 TO 51
tested for a null
string.
When not null, the 601110          IF TRIM$(FIELD$)=STATE.CODES$(SI%)
field is compared 601120          X$ = STATES$(SI%)
to the list of possible 601130          BREAK \ REM Found...exit
state codes in an 601140          IFEND
array.             601150      NEXT
                   601160      IFEND
~
```

Statements

---

# UCASE\$ Function

UCASE\$ returns a string with all alphabetic characters in uppercase.

UCASE\$( *string-expression* )

**Operations**      The *string-expression* is evaluated and all alphabetic characters are replaced with their uppercase equivalent; non-alphabetic characters are not modified. The resulting string is returned.

**See also**          [LCASE\\$](#)

## Examples

<i>A character is accepted from the operator.</i>	200210	WAIT #0 \ GET ANS\$
	200220	
	200230	IF NOT MOD.DONE% AND NOT EVENT.FLAG%
	200240	
<i>Since the LABEL.CODES\$ field contains only uppercase characters, the input character is folded to uppercase before comparing it to the LABEL.CODES\$ field.</i>	200250	IF SCH(1,LABEL.CODES\$,UCASE\$(ANS\$))
	200260	
	~	

Statements

# UNGET Statement

The UNGET statement puts characters onto the console's input stream buffer.

- 1

UNGET *expression-list*
- 2

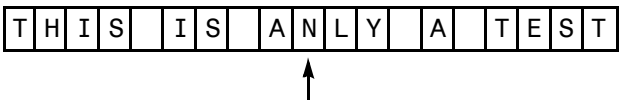
UNGET *#channel: expression-list*

**Operation**      **Mode 1**—One character for each expression in *expression-list* is placed in the console's input buffer.

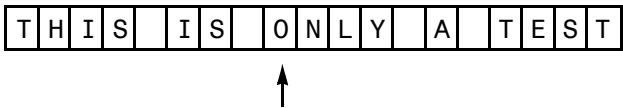
**Mode 2**—One character for each expression in *expression-list* is placed in the file channel's input buffer. This file must have been opened as an INPUT SEQUENTIAL file.

**Notes**              Only one character is guaranteed to be placed in the input stream. Whether more can be placed depends upon the device and file. Frequently, the main console can UNGET as many as seven or eight characters; disk files might be able to UNGET from 1 to 255 characters, depending upon the prior buffer input location; communications devices will probably allow only one UNGET character.

To illustrate the process, assume that the operator has typed the string "THIS IS ONLY A TEST" on the console. The program has accepted nine of these characters via GET statements or maybe a LINPUT USING statement. The console's input buffer and buffer pointer would be:



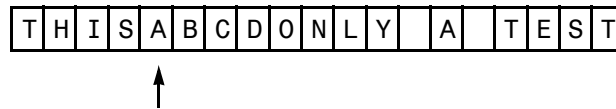
After an UNGET "O"



The buffer has been changed and the buffer pointer moved so that the next input from the console will start with the letter "O".

Statements

If the device can accept multiple UNGETs, as the console can, and an UNGET "A","B","C","D" were performed, the buffer and pointer would look like:



As can be seen, it doesn't matter what has, or has not, been typed. What does matter is what has been accepted by the program. This applies to all sources of input that the UNGET may operate on. When an [INPUT](#) statement is performed, the string "ABCDONLY A TEST" is accepted.

Also, as shown, the characters that are placed with UNGET do not have to be the same characters that were accepted from the input stream earlier.

The data file is not changed by the UNGET statement, only the input buffer.

**Restrictions** Only one character is guaranteed to be placed in the input stream. Until that character is read, it is possible that additional UNGET statements will not change the input buffer.

*channel* must refer to an open file channel, opened for INPUT SEQUENTIAL.

**See also** [GET](#)

### Examples

<i>Accept one character from the console, silently (without display).</i>	2450	GET CHAR%
	2460	
<i>Test the character for special situations.</i>	2470	SELECT CHAR% \ REM Test input character
	2480	CASE 10 \ REM Up arrow
	2490	GOSUB DOWN.ARROW
	2500	CASE 11 \ REM Down arrow
	2510	GOSUB UP.ARROW
	2520	CASE 23 \ REM Top key
	2530	GOSUB TOP
	2540	CASE 25 \ REM Bottom key
	2550	GOSUB BOTTOM
<i>If the character was not special, place it back on the input stream and then perform normal input.</i>	2560	OTHERWISE \ REM None of the above
	2570	UNGET CHAR% \ REM Put char back on input stream
	2580	INPUT REPLY\$ \ REM Get char and string following
	2590	CEND
	~	



---

# UNLOCK Statement

UNLOCK releases the record lock placed on a file record by input or read statements.

UNLOCK #channel

Operation	All records that are locked for the designated file channel are released.
Notes	<p>A file must be opened with access mode UPDATE for any record locks to be used in the file. When the access mode is UPDATE, any <a href="#">INPUT</a>, <a href="#">LINPUT</a>, <a href="#">READ</a>, <a href="#">READNEXT</a>, <a href="#">READPREV</a>, <a href="#">MAT READ</a>, <a href="#">MAT READNEXT</a>, <a href="#">MAT READPREV</a> statement locks the record that is read, preventing other users from accessing the record until this program releases the record lock.</p> <p>Record locks are removed by this statement and by closing the file. If option MULTILOCK is not in effect for the file channel, accessing a different record in the file with a read statement, write statement, or <a href="#">DELETE</a> statement will also unlock records in the file..</p> <p>A file that is opened with the LOCK option places a file lock on the file. This statement does not release file locks. A file must be closed to release a file lock.</p> <p>MultiUser BASIC normally locks one record at a time in a specific file channel. The MULTILOCK option of the <a href="#">OPEN</a> statement allows multiple records in the file to be locked at one time. The UNLOCK statement unlocks all records in the file.</p>
Restrictions	The <i>channel</i> must refer to an open file channel.
See also	<a href="#">INPUT</a> , <a href="#">LINPUT</a> , <a href="#">MAT READ</a> , <a href="#">MAT READNEXT</a> , <a href="#">MAT READPREV</a> , <a href="#">OPEN</a> , <a href="#">READ</a> , <a href="#">READNEXT</a> , <a href="#">READPREV</a>

Statements

Examples

```
6810  MODIFICATIONS:
6820
6830      WAIT #0 \ GET MOD$ \ REM Get modification response
6840
6850      SELECT
6860          CASE MOD$=DELKEY$ OR MOD$=DELCHR$ \ REM Delete
        ~
7120          CASE MOD$=FILEKEY$ OR MOD$=SAVEKEY$ \ REM Save
        ~
7280          CASE MOD$=ESC$ OR MOD$=CHR$(128) \ REM Ignore?
7290          UNLOCK #1 \ UNLOCK #2 \ REM Release record
        ~
```

*The UNLOCK is used when it is determined that the program is finished with the record and is not going to save or delete it.*

*Without the UNLOCK, the program would keep a lock on a record that it no longer needed, preventing other users from accessing it.*

---

## USED.WINDOWS Function

USED.WINDOWS returns the number of currently open windows.

### USED.WINDOWS

**Operation**      The number of windows currently open.

$\text{USED.WINDOWS} = \text{TOTAL.WINDOWS} - \text{AVAIL.WINDOWS}$

**See also**      [AVAIL.WINDOWS](#), [TOTAL.WINDOWS](#)

**Examples**

```
10 ! Program: GLMAINT, linked from MENU program
20
30 ! This program uses many windows...it saves any currently
40 ! open windows before opening it own. Upon completion it will
50 ! restore those windows before returning control back to MENU
60
70     IF USED.WINDOWS! Are there any windows in use?
80         WINDOWS.SAVED% = USED.WINDOWS
90         GOSUB SAVE.OPEN.WINDOWS    ! Save them
100     IFEND
110
~
```

---

## VAL Function

VAL returns the numeric value represented by a string.

VAL( *string-expression* )

**Operations**      The *string-expression* is analyzed for a valid number (see [NBR](#) function). If the string does contain a valid number then its value is returned; otherwise a 0 is returned.

**See also**          [NBR](#)

**Examples**        The following section of code uses the [NBR](#) function and the VAL function to check the input from the operator. The [LINPUT USING](#) statement is used instead of [INPUT](#) to allow for better screen control.

```
1000 GET.NUMBER:
1010
1020     LINPUT "Enter dollar amount: ", USING SPACE$(9),AMT$
1030     IF NBR(AMT$)
1040         AMT = VAL(AMT$)
1050     ELSE GOSUB BAD.ENTRY
1060         IFEND
1070
1080     RETURN
```

---

## VDI Statement

The VDI statement provides a generalized interface to VDI devices.

**VDI** [#*channel*:] *control*, *input*, *points-in*, *output*, *points-out*

**Operation**      The Virtual Device Interface (VDI) driver is accessed at its primitive call level. The five integer arrays specified by *control*, *input*, *points-in*, *output*, *points-out*, are used by the VDI device to control and communicate the results of a graphics operation.

**Notes**            The function of this statement is to provide access to VDI device features that are not supported by the other VDI statements. Generally, the other statements provide a much easier methods of accessing the standard features of VDI devices.

The various arrays must be dimensioned but it doesn't matter whether **OPTION** BASE 0 or **OPTION** BASE 1 is in effect. The VDI driver will access the arrays as scalars. In the following descriptions, the references to the array elements use **OPTION** BASE 1. Merely subtract one from each reference when **OPTION** BASE 0 is used.

**Control array.** The *control* array contains the parameters specifying the function requested and the number of input and output parameters available. It requires six elements.

Element	Function
1	Indicates the function or graphics command.
2	Specifies the number of <i>points-in</i> value pairs available.
3	Set by VDI; indicates the number of <i>points-out</i> value pairs returned.
4	Specifies the number of <i>input</i> parameters available.
5	Set by VDI; indicates the number of <i>output</i> parameters returned.
6	Indicates a subfunction to be performed.

**Input array.** The *input* array contains data items being given to the VDI driver. For example, the text characters to draw, the angles to use for arcs, *etc.*

**Points-in array.** The *points-in* array contains x and y coordinate pairs used in the operation of the function. For example, the points of a line, the center of a circle, *etc.* It is best to allocate the *points-in* as a two-dimensional array, such as PI%(6,2).

**Output array.** The *output* array contains values set by the VDI and returned to the program. For example, when a line style is set, the *output* array will contain the actual style set.

**Points-out array.** The *points-out* array contains x and y coordinate pairs set by the VDI and returned to the program. It is best to allocate the *points-out* as a two-dimensional array, such as PO%(6,2).

### Caution

The *output* and *points-out* arrays must be dimensioned with sufficient elements for the command being performed. The VDI statement accesses the VDI device driver as a primitive. Very little, if any, validation is performed.

When a command description states that six *points-out* data pairs are returned, then that many items are returned, whether or not the array is allocated large enough to accept them. The program's data integrity is lost when the return arrays are not allocated with sufficient space.

### VDI Open

Probably the most common usage of the VDI statement is to open a VDI device. Although the OPEN statement opens VDI devices, this form of the open allows the program to specify the initial parameters and, more importantly, to receive the device's specifications.

Control array

Element	Value	Meaning
1	1	Device open.
2	0	No <i>points-in</i> elements used.
3	6	6 <i>points-out</i> pairs returned.
4	10	<i>input</i> elements required.
5	45	<i>output</i> elements returned.
6		Not used.

The initial parameters used by the device for lines, markers, text and fill areas can be set when the device is opened.

#### Input array

Element	Meaning
1	Not used.
2	Initial line style.
3	Initial line color.
4	Initial marker style.
5	Initial marker color.
6	Initial text font.
7	Initial text color.
8	Initial fill style.
9	Not used.
10	Initial fill color.

After the device is opened, the *output* and *points-out* arrays are filled in by the device driver. These two arrays define the capabilities of the device.

#### Output array

Element	Meaning
1	Maximum horizontal address of device in device units. (1 less than horizontal resolution).
2	Maximum vertical address of device in device units. (1 less than vertical resolution).
3	Device scaling flag: 0 = device can precisely scale; 1 = device cannot.
4	Width of device pixel in microns.
5	Height of device pixel in microns.
6	Number of character sizes supported.
7	Number of line styles supported.
8	Number of line widths supported.
9	Number of marker styles supported.

## Output array

Element	Meaning
10	Number of marker sizes supported.
11	Number of character fonts supported.
12	Number of fill styles supported.
13	Not used.
14	Maximum number of colors displayable at one time.
15	Number of drawing primitives supported.
16-24	<p>List of drawing primitives supported, terminated by a -1. The codes for drawing primitives are:</p> <p>1 = bar 2 = arc 3 = pie slice 4 = circle</p>
25-35	<p>List of attributes associated with each drawing primitive, terminated by a -1. The attribute codes are:</p> <p>0 = polyline 1 = polymarker 2 = text 3 = fill area</p>
36	Color capability flag (0 = no, 1 = yes).
37	Text rotation capability flag (0 = no, 1 = yes).
38	Fill area capability flag (0 = no, 1 = yes).
39	Read cell array capability flag (0 = no, 1 = yes).
40	<p>Number of colors available.</p> <p>0 = Infinite 2 = Monochrome 3—32767 = Actual number of colors</p>
41	Number of input devices supported, such as joysticks, track balls, mice, touch screens, etc.
42	Number of valuator devices supported.
43	Number of choice devices supported.
44	Number of string devices supported.



### Output array

Element	Meaning
45	Type of device:  0 = Output only 1 = Input only 2 = Input and output capable

### Points-out array

Element		Meaning
1	1	Set to zero.
	2	Minimum character height, in device units.
2	1	Set to zero.
	2	Maximum character height in device units.
3	1	Minimum line width in device units.
	2	Set to zero.
4	1	Maximum line width in device units.
	2	Set to zero.
5	1	Set to zero.
	2	Minimum marker height in device units.

With the information available in these two output arrays, subsequent operations performed on the device can be optimized. The most common optimization is the adjustment of x and y coordinates to accommodate the nonuniform resolution of a device.

For example, most graphics screens have different horizontal and vertical resolutions. When it is desired to draw a square on the screen this difference in resolution must be taken into account or the “square” will be a rectangle.

```

10000 COMPUTE.YADJUST:
10010
10020     DIM VCONTROL%(6),VIN%(10),VPIN%(1,2),VOUT%(45),VPOUT%(6,2)
10030
10040     MAT VCONTROL% = (0)
10050     VCONTROL%(1) = 1 \ REM Open command
10060
10070     VDI VCONTROL%,VIN%,VPIN%,VOUT%,VPOUT%
10080
10090     YADJUST% = ((VOUT%(1)+1)/(VOUT%(2)+1))*VOUT%(4)/VOUT%(5)
10100
10110     RETURN

```

## VDI Close

Close the VDI. After execution, no other VDI statement function will operate, except VDI Open.

Control array

Element	Value	Meaning
1	2	Device close.
2	0	No <i>points-in</i> used.
3	0	No <i>points-out</i> used.
4	0	No <i>input</i> used.
5	0	No <i>output</i> used.
6		Not used.

## Polyline

Draws a series of connected line segments using the current line style, width, and color.

Control array

Element	Value	Meaning
1	6	Polyline.
2	n	Number of <i>points-in</i> pairs used. The <i>points-in</i> array must have $2n$ elements defined for the sets of x,y vertices of the line segments. $1 < n \leq 32767$ .
3	0	No <i>points-out</i> used.
4	0	No <i>input</i> used.
5	0	No <i>output</i> used.
6		Not used.

## Control array

Element	Value	Meaning
1	7	Polymarker.
2	$n$	Number of <i>points-in</i> pairs used. The <i>points-in</i> array must have $2n$ elements defined for the sets of x,y vertices of the markers. $1 < n \leq 32767$ .
3	0	No <i>points-out</i> used.
4	0	No <i>input</i> used.
5	0	No <i>output</i> used.
6		Not used.

The *polyline* and *polymarker* functions can be used, one after the other, to draw a complicated line chart. For example:

```

1000 MAT READ VIN% \ REM Get array of points
1010
1020 MAT VCONTROL% = (0)
1030 VCONTROL%(1) = 6 \ REM Command is polyline
1040 VCONTROL%(2) = 65 \ REM Number of points to connect
1050
1060 VDI VCONTROL%,VIN%,VPIN%,VOUT%,VPOUT%
1070
1080     VCONTROL%(1) = 7 \ REM Command is polymarker
1090 VDI VCONTROL%,VIN%,VPIN%,VOUT%,VPOUT%

```

The above code would draw a line chart connecting 65 points with a line in the current line style and color. It then places a marker on each of the 65 points.

## Text

Draw text characters using the current text rotation, angle, path, font, size, and color. Because of the necessity of converting a text string to individual character values required by this statement, the TEXT statement is much easier to use.

### Control array

Element	Value	Meaning
1	8	Text.
2	1	Number of <i>points-in</i> pairs used. The <i>points-in</i> array contains the x,y coordinate of the lower left corner of the first character displayed.
3	0	No <i>points-out</i> used.
4	n	Number of <i>input</i> used. The <i>input</i> array contains the individual characters to display. $1 < n \leq 32767$ .
5	0	No <i>output</i> used.
6		Not used.

## Polygon Fill

Draws an irregularly shaped polygon and fills the interior. The polygon is drawn using the a line style of 0, and the current line width, and color; the interior is filled using the current fill style and color.

### Control array

Element	Value	Meaning
1	9	Polygon fill.
2	n	Number of <i>points-in</i> pairs used. The <i>points-in</i> array must have $2n$ elements defined for the sets of x,y vertices of the polygon. $3 < n \leq 32767$ .
3	0	No <i>points-out</i> used.
4	0	No <i>input</i> used.
5	0	No <i>output</i> used.
6		Not used.

The polygon fill command can draw rectangles. However, the following command, or even the [FILL BAR](#) statement, provides a much more efficient method of drawing and filling rectangles.

Whether or not the polygon is actually filled depends upon the capabilities of the device. Most bit-mapped graphics devices (including the console monitor) do not support polygon fill. Only the polygon shape will be drawn without filling the interior.

## Draw

Draw shapes using the device's drawing primitive capabilities.

**Draw bar.** Draws and fills a rectangular region using the line style 0, the current line width and color, and the current fill style and color.

### Control array

Element	Value	Meaning
1	11	Draw.
2	2	Number of <i>points-in</i> pairs used. The <i>points-in</i> array:  1,1 Lower left corner x value. 1,2 Lower left corner y value. 2,1 Upper right corner x value. 2,2 Upper right corner y value.
3	0	No <i>points-out</i> used.
4	0	No <i>input</i> used.
5	0	No <i>output</i> used.
6	1	Draw bar.

**Draw arc.** Draws an arc of a circle, counterclockwise. The arc drawn uses the line style 0 and the current line width, and color.

### Control array

Element	Value	Meaning
1	11	Draw.
2	4	Number of <i>points-in</i> pairs used. The <i>points-in</i> array:  1,1 x coordinate of arc center. 1,2 y coordinate of arc center. 4,1 radius of arc.
3	0	No <i>points-out</i> used.
4	2	2 <i>input</i> elements used for the starting and ending angles, in tenths of degrees.
5	0	No <i>output</i> used.
6	2	Draw arc.

**Draw pie.** Draws a pie-shaped region of a circle, counterclockwise. The pie-shape drawn uses the line style 0, the current line width and color, and the current fill style and color.

#### Control array

Element	Value	Meaning
1	11	Draw.
2	4	Number of <i>points-in</i> pairs used. The <i>points-in</i> array: 1,1 x coordinate of arc center. 1,2 y coordinate of arc center. 4,1 radius of arc.
3	0	No <i>points-out</i> used.
4	2	2 <i>input</i> elements used for the starting and ending angles, in tenths of degrees.
5	0	No <i>output</i> used.
6	3	Draw pie.

**Draw circle.** Draws a circle using the line style 0, the current line width and color, and the current fill style and color.

#### Control array

Element	Value	Meaning
1	11	Draw.
2	3	Number of <i>points-in</i> pairs used. The <i>points-in</i> array: 1,1 x coordinate of arc center. 1,2 y coordinate of arc center. 3,1 radius of arc.
3	0	No <i>points-out</i> used.
4	0	No <i>input</i> used.
5	0	No <i>output</i> used.
6	4	Draw circle.

The various text, line, marker, and fill attributes can be set with the following functions of the VDI statement:

#### Set character size.

##### Control array

Element	Value	Meaning
1	12	Set character size.
2	1	One <i>points-in</i> pairs used:  1,1 Zero 1,2 Requested character size index
3	2	Number <i>points-out</i> returned:  1,1 Actual character width in device units. 1,2 Actual character height in device units. 2,1 Character cell width in device units.. 2,2 Character cell height in device units.
4	0	No <i>input</i> used.
5	0	No <i>output</i> used.
6		Not used.

The advantage of this primitive is that it returns the actual size of the characters that will be used. With this information a program can position the text properly, taking into account the length and height of the text as drawn.

## Set text rotation.

Control array

Element	Value	Meaning
1	13	Set text angle or path.
2	0	No <i>points-in</i> used.
3	0	No <i>points-out</i> returned.
4	1 or 2	Number of <i>input</i> used. The value '1' sets text angles, '2' sets text paths.  1 Text angle in tenths of degrees. Set to -1 for setting path angles. 2 Path angle in tenths of degrees.
5	1	Number <i>output</i> returned. The text angle or path actually used is returned.
6		Not used.

The text and path angles cannot be set at the same time. Two calls to this command are required: one for the text angle, another for the text path.

## Set line style.

Control array

Element	Value	Meaning
1	15	Set line style.
2	0	No <i>points-in</i> used.
3	0	No <i>points-out</i> returned.
4	1	Number of <i>input</i> used. Requested line style.
5	1	Number of <i>output</i> returned. Line style actually selected.
6		Not used.



## Set line width.

### Control array

Element	Value	Meaning
1	16	Set line width.
2	1	Number of <i>points-in</i> pairs used.  1,1 Requested line width, in device units. 1,2 Set to zero.
3	1	Number of <i>points-out</i> returned.  1,1 Line width actually selected, in device units. 1,2 Set to zero.
4	0	No <i>input</i> used.
5	0	No <i>output</i> returned.
6		Not used.

## Set line color.

### Control array

Element	Value	Meaning
1	17	Set line color.
2	0	No <i>points-in</i> used.
3	0	No <i>points-out</i> returned.
4	1	Number of <i>input</i> used. Requested line color.
5	1	Number of <i>output</i> returned. Line color actually selected.
6		Not used.

## Set marker style.

### Control array

Element	Value	Meaning
1	18	Set marker style.
2	0	No <i>points-in</i> used.
3	0	No <i>points-out</i> returned.
4	1	Number of <i>input</i> used. Requested marker style.
5	1	Number of <i>output</i> returned. Marker style actually selected.
6		Not used.

## Set marker size.

### Control array

Element	Value	Meaning
1	19	Set marker size.
2	1	Number of <i>points-in</i> pairs used.  1,1 Set to zero. 1,2 Requested marker size, in device units.
3	1	Number of <i>points-out</i> returned.  1,1 Set to zero. 1,2 Marker size actually selected, in device units.
4	0	No <i>input</i> used.
5	0	No <i>output</i> returned.
6		Not used.

## Set marker color.

### Control array

Element	Value	Meaning
1	20	Set marker color.
2	0	No <i>points-in</i> used.
3	0	No <i>points-out</i> returned.
4	1	Number of <i>input</i> used. Requested marker color.
5	1	Number of <i>output</i> returned. Marker color actually selected.
6		Not used.

## Set text font.

### Control array

Element	Value	Meaning
1	21	Set text font.
2	0	No <i>points-in</i> used.
3	0	No <i>points-out</i> returned.
4	1	Number of <i>input</i> used. Requested text font.
5	1	Number of <i>output</i> returned. Text font actually selected.
6		Not used.

## Set text color.

### Control array

Element	Value	Meaning
1	22	Set text color.
2	0	No <i>points-in</i> used.
3	0	No <i>points-out</i> returned.
4	1	Number of <i>input</i> used. Requested text color.
5	1	Number of <i>output</i> returned. Text color actually selected.
6		Not used.

## Set fill style.

### Control array

Element	Value	Meaning
1	23	Set fill style.
2	0	No <i>points-in</i> used.
3	0	No <i>points-out</i> returned.
4	1	Number of <i>input</i> used. Requested fill style.
5	1	Number of <i>output</i> returned. Fill style actually selected.
6		Not used.

## Set fill color.

### Control array

Element	Value	Meaning
1	25	Set fill color.
2	0	No <i>points-in</i> used.
3	0	No <i>points-out</i> returned.
4	1	Number of <i>input</i> used. Requested fill color.
5	1	Number of <i>output</i> returned. Fill color actually selected.
6		Not used.

See also

Other VDI statements

## Examples

	90000	VDI.INIT:
	90010	
	90020	DIM VCONTROL%(6),VIN%(10),VPIN%(6,2),VOUT%(45),VPOUT%(6,2)
	90030	
	90040	VCONTROL%(1) = 1
	90050	
<i>The VDI open command initializes the graphics device and returns parameters describing the capabilities of the device.</i>	90060	<b>VDI VCONTROL%,VIN%,VPIN%,VOUT%,VPOUT%</b>
	90070	
<i>If the device's horizontal and vertical resolutions are different, the YADJUST value can be used to adjust the y coordinate for displays.</i>	90080	YADJUST = (VOUT%(1)+1)/(VOUT%(2)+1)*VOUT%(4)/VOUT%(5)
	90090	
	90100	V.PIXEL.WIDTH% = 32767/(VOUT%(1)+1)
	90110	V.PIXEL.HEIGHT% = 32767/(VOUT%(2)+1)
	90120	
	90130	RETURN
<i>This function sets the text size and computes the actual size of the characters in VDI points.</i>	80000	DEF FN.SET.TEXT.SIZE%(S%)
	80010	
	80020	MAT VCONTROL% = (0)
	80030	MAT VIN% = (0)
	80040	VCONTROL%(1) = 12 \ REM Set char size function
	80050	VCONTROL%(2) = 1 \ REM One points in pair
	80060	VPIN%(1,1) = 0
	80070	VPIN%(1,2) = S% \ REM Text size requested
	80080	
	80090	<b>VDI VCONTROL%,VIN%,VPIN%,VOUT%,VPOUT%</b>
	80100	
	80110	CHAR.WIDTH% = VPOUT%(2,1)*V.PIXEL.WIDTH%
	80120	CHAR.HEIGHT% = VPOUT%(2,2)*V.PIXEL.HEIGHT%
	80130	
	80140	FN.SET.TEXT.SIZE% = CHAR.WIDTH%
	80150	
<i>The VPIXEL.WIDTH% and HEIGHT% are computed in the VDI.INIT routine, above.</i>	80160	FNEND

---

## WAIT Statement

The WAIT statement performs a console page wait or waits for a byte or character to be available on an input channel.

- 1 **WAIT**
  - 2 **WAIT** *#channel*

### Operation

**Mode 1**—A “page wait” is performed on the console: The cursor is positioned to the bottom left corner of the screen or window; a caret is displayed (^) and the cursor is positioned on top of it. Unless the page wait feature is suspended ([Break](#), [W](#)), processing is suspended until the operator enters a non-alphanumeric key (a control character, enter key, arrow key, etc.). After entry, the caret is erased and processing resumes.

**Mode 2**—Processing is suspended until a character is available for input on the designated input *channel*.

### Notes

Entry of a key that has an [ON KEY](#) trap set for it, will “wake up” a [Mode 1](#) WAIT or a [Mode 2](#) WAIT when the channel is the console. However, there will be no character available as the [ON KEY](#) trap used it. (Unless the [ON KEY](#) event handler used the [UNGET](#) to place the character back onto the input stream.) If a WAIT #0 is followed by a [GET](#) statement, the program should test to make sure that a character was actually accepted. Refer to the example, below.



If a WAIT #0 executed while an [ON TIMEOUT](#) statement is in effect, and no key is entered before the time period elapses, the WAIT statement is exited and the time-out event handler is invoked.

### Restrictions

The *channel* must refer to an open file channel, opened with INPUT SEQUENTIAL, or UPDATE SEQUENTIAL. (Channel number 0 is always open to the console).

### See also

[GET](#), [ON KEY](#), [OPEN](#), [WAIT EVENT](#)

## Examples

```
809000  RELEASE.HELP.PAGE:
809010
809020      PRINT AT$(WIDTH%,DEPTH%); \ REM Position to bottom right
           corner
809030
809040      REPLY% = 0
809050      WHILE REPLY%=0
809060          WAIT #0 \ GET REPLY%
809070      WEND
809080

Even though the 809090      L% = 0
WAIT is per-    809100      PRINT CLS$;
formed, because of 809110
ON KEY trapping, 809120      RETURN
it is possible for the
GET to receive no
characters. Thus,
the WHILE loop.
```

---

## WAIT EVENT Statement

WAIT EVENT suspends program execution until a semaphore is set.

WAIT EVENT ( *semaphore* )

**Operation**      Program execution is suspended until *semaphore* is on.

**Notes**            While a program is paused by this statement, it takes virtually no system resources, unlike a [WHILE-WEND](#) loop testing the status of the semaphore, which uses all of the cpu time available.

When a WAIT EVENT is performed, nothing will awaken the program except the setting of semaphore, a program abort ([Break](#), [Q](#)), an [ON KEY](#) event, an [ON EVENT](#) event (for another semaphore), or, in the interpreter only, a program cancel ([Break](#), [C](#)).

When the *semaphore* is already on, or is set on, the *semaphore* is reset and execution resumes in this program with the statement following the WAIT EVENT.

As stated, the WAIT EVENT statement waits for a semaphore to be set on. There is no efficient method of waiting for a semaphore to be set off. For example:

```
WHILE EVENT(SEMAPHORE%) \ WEND            ! Loop until semaphore off
```

is very inefficient as it will use as much of the CPU time resource as possible. Try to design the semaphore meaning and usage so that the WAIT EVENT statement can be used.

When two separate tasks are waiting for the same semaphore (with either the WAIT EVENT or the [ON EVENT](#) statements), only one task will be awakened when the semaphore is set. The process of awakening the task clears the semaphore, and the other task will not be awakened unless the semaphore is set again.

**Restrictions**      There are 64 semaphores available to a user and the *semaphore* must have a value in the range of 0–63. Specifying a *semaphore* value outside of this range causes the WAIT EVENT to be ignored.

**See also**            [ON EVENT](#), [TIMER](#) statements, [SEMAPHORE](#) function



Examples

<i>This might be the main task of a multiple task application that communicates between tasks via the PUT and GET COMMON statements.</i>	10 20 30 40 50 60 70 80 90	COMMON INFORMATION\$  COMMON.DATA% = SEMAPHORE ( "COMMON-DATA" ) COMMON.DATA.CHANGED% = SEMAPHORE ( "DATA-CHANGED" )  SUB.TASK% = ACTIVATE ( "TASK1" , -1,0)  X% = SET (COMMON.DATA%) \ REM Okay to change common
--	--	--

<i>The WAIT EVENT statement waits for the subtask to indicate that it has sent an update to the common data.</i>	100 110 ~	WAIT EVENT (COMMON.DATA.CHANGED%)
<i>The COM-MON.DATA semaphore is used to ensure that only one task at a time sends data to the parent's common data.</i>	150 160	X% = SET (COMMON.DATA%) \ REM Okay to change common

---

## WEND Statement

The WEND statement marks the end of a [WHILE](#)-WEND program structure.

### WEND

#### Operation

The WEND statement is part of a program structure that starts with the [WHILE](#) statement and ends with the WEND statement. These two statements are separated by zero or more statements that comprise the statements of the loop.

Control is transferred to the current [WHILE](#)-WEND structure's [WHILE](#) statement, causing that [WHILE](#) statement's control expression to be reevaluated and the loop repeated, depending upon the results of the test.

Refer to the [WHILE](#) statement description for additional information about [WHILE](#)-WEND program structures, restrictions, and an example.

#### See also

[WHILE](#)

---

## WHILE Statement

The WHILE statement marks the start of, and the conditions for, a WHILE-WEND programming structure.

**WHILE** *expression*

**Operation** The WHILE statement is part of a program structure loop that starts with the WHILE statement and ends with the WEND statement. These two statements are separated by zero or more statements that comprise the statements of the loop.

The *expression* is tested for a true (non-zero) or false (0) value. A false value transfers control to the statement following the WEND statement. A true value causes the statements of the loop to be executed.

**Notes** WHILE-WEND structures may be nested to any depth.

The CONTINUE and BREAK statements may be used inside of a WHILE-WEND program structure. A CONTINUE will transfer control to the WEND statement; BREAK transfers control to the statement following the WEND.

**Restrictions** Every WHILE statement must have a WEND statement; every WEND statement must have a WHILE statement. The WEND statement must physically follow the WHILE statement in the program.

**Special Note:** This statement defines the beginning of a program structure. This type of program structure should only be entered at the top (WHILE statement). If control is transferred into this structure via a RETURN line-ref, RESUME line-ref, or a GOTO line-ref, an error will occur when the WEND statement is encountered.

It is acceptable practice to branch out of a WHILE-WEND structure with the GOTO statement, although the BREAK statement provides a better method.

**See also** BREAK, CONTINUE, WEND

## Examples

*This loop reads and displays records from a file.*

800250  
800260  
800270  
800280  
800290  
800300  
800310  
800320  
800330  
800340

*Note that the control condition (status of read) is set prior to entering the loop and it is the last operation prior to repeating the loop.*

```
LINPUT #38: HELP.TEXT$  
  
L% = 0 \ REM Number of lines displayed in window  
  
WHILE NOT EOF(38) \ REM Repeat until end of help text  
  IF L%>=DEPTH% THEN GOSUB RELEASE.HELP.PAGE  
  L% = L%+1 \ REM Count lines displayed in window  
  PRINT AT$(2,L%);HELP.TEXT$;  
  LINPUT #38: HELP.TEXT$ \ REM Get next help line  
WEND
```

## WINDOW CHOICE Statement

WINDOW CHOICE opens a window and displays a choice list. The operator is allowed to select one of the choices and program execution continues.

- 1 **WINDOW CHOICE** *col, row, count, item-array, index-variable* [ *options* ]
- 2 **WINDOW CHOICE KEEP** *window, col, row, count, item-array, index-variable* [ *options* ]
- 3 **WINDOW CHOICE KEEP ADDROF**( *num-var* ), *col, row, count, item-array, index-variable* [ *options* ]

---

*options*

- » [; **TITLE** *title-exp*  
    [, **TOP** | **BOTTOM**]  
    [, **LEFT** | **CENTER** | **RIGHT**]  
    [, **NORMAL** | **REVERSE** | **BLINK** | **UNDERLINE** | **HALF**]  
    [, **COLOR** *fg* [, *bg*] ] ]
- » [; **INVERT ON** | **OFF**]
- » [; **COLOR** *fg* [, *bg* [, *rv-fg*, *rv-bg*] ] ]
- » [; **HEIGHT** *height*]
- » [; **HELP**]
- » [; **HELPLINE** *help-array*  
    [, **INVERT ON** | **OFF** ]  
    [, **COLOR** *fg* [, *bg*] ]
- » [; **HOT** *hot-key-array*]
- » [; **ON.KEY.VALID**]
- » [; **USING** *start-item*]
- » [; **WIDTH** *width*]

### Operation

**Mode 1**—A new window is created for the choice window. The window number used is the first unused window number available. Upon exit from the WINDOW CHOICE statement, the prior window is selected (with the same update status as before execution of the WINDOW CHOICE statement), the choice window is removed from the display and it is closed.

**Mode 2**—Similar to [Mode 1](#) except that the window number used for the choice window is indicated by *window*. If this window is an open window, it is reopened as the choice window. Upon exit from the WINDOW CHOICE

statement, the prior window is selected (with the same update status as before execution of the WINDOW CHOICE statement), but the choice window is not removed nor is it closed.

**Mode 3**—Similar to **Mode 2** except the window number is the first unused window number available. This window number is assigned to the variable *num-var*. Upon exit from the WINDOW CHOICE statement, the prior window is selected but the choice window remains open.

## Notes

This statement uses a window to display a list of choices. This choice window is always outlined with a double line frame and a drop shadow on the right. The color of the frame is the same as the normal video color of the interior of the window.

The *item-array* is a string array with each element representing one item to be displayed. This array may be single or double dimensioned but it is always treated as a one-dimensional array and, when OPTION BASE 0 is in effect, subscript zero is always ignored. For example, an array dimensioned as (3,2) contains the following elements:

	0	1	2	3
0	"0,0"	"0,1"	"0,2"	"0,3"
1	"1,0"	"1,1"	"1,2"	"1,3"
2	"2,0"	"2,1"	"2,2"	"2,3"

This array, when used by the WINDOW CHOICE statement, is treated as:

1	2	3	4	5	6	7	8	9	10	11
"0,1"	"0,2"	"0,3"	"1,0"	"1,1"	"1,2"	"1,3"	"2,0"	"2,1"	"2,2"	"2,3"

A blank or null item in the *item-array* is displayed as a horizontal line and cannot be selected. These horizontal lines are useful for separating the choice list into sections.

The position of the window is specified by the parameters *col* and *row*, which define the upper left corner of the interior of the window.





The width of the choice window is the longer of the longest item in *item-array* and the length of the title text, unless the WIDTH option is used. The height of the choice window is the smaller of the number of items in *count* and the remaining number of lines on the screen (*row*-screen-height-2), unless the HEIGHT option is used. If the choice window height is less than *count* a scroll bar is used on the right side of the window.






The *count* value indicates the number of items in the choice list. This value must be equal to, or less than the allocated size of *item-array*. Only *count* items of *item-array* are displayed in the choice window. This *count* value includes any blank items in the *item-array*. If there is insufficient space available on the screen for the choice window to display all *count* items of *item-array*, the choice window will be sized to the full depth of the screen and the items are displayed in “pages.”

The items from *item-array* are displayed, one item per line, with as many items displayed at one time as will fit in the choice window. A selected item is indicated by displaying the item in reverse video. This reverse video item display is called the **highlight bar**.

When there are more items in the choice list than can be displayed in the choice window, a **scroll bar** is displayed on the right frame of the window. This scroll bar indicates the position of the highlight bar in relation to the entire choice list, not just the portion of the choice list currently displayed in the window.

While the choice window is displayed, the operator may use any key listed in the following table for choice positioning and selection. There may be other keys available, depending upon the option HOT, described later.

Key	Meaning
	Moves the highlight bar to the previous item in the list. When the highlight bar is at the top of the choice window, the items in the display are scrolled down one line and the prior item is displayed at the top. When the highlight bar is at the top of the choice list, no action is taken. Blank items (shown as horizontal bars) are skipped.
	Moves the highlight bar to the next item in the list. When the highlight bar is at the bottom of the choice window, the items in the display are scrolled up one line and the next item is displayed at the bottom. When the highlight bar is at the bottom of the choice list, no action is taken. Blank items (shown as horizontal bars) are skipped.
	Positions the highlight bar to the first item in the list. If this first item is not currently displayed in the window, the contents of the choice window are repainted, with the first item at the top of the window.
	Positions the highlight bar to the last item in the list. If this last item is not currently displayed in the window, the contents of the choice window are repainted, with the last item at the bottom of the window.

	The previous page or window full of choices is displayed and the top item in that display is highlighted. If the current display is the first page of the choice list, the top item is highlighted.
	The next page or window full of choices is displayed. If the current display already includes the last item of the choice list, the bottom item is highlighted.
	Selects the currently highlighted item. The <i>index-variable</i> is set to the value of the index of the selected item and the WINDOW CHOICE statement is terminated.
	The <i>index-variable</i> is set to zero and the WINDOW CHOICE statement is terminated.
	This key is ignored unless the HELP option is specified. When HELP is specified and this key is pressed, the <i>index-variable</i> is set to the negative value of the selected item's index and the WINDOW CHOICE statement is terminated.

When the HOT option is not used (see options, below), items may be selected by pressing the first nonspace character of the item. A case insensitive search is performed. The highlight bar is positioned to the next item that starts with the pressed character. If that item is not currently displayed, the contents of the window is repainted with the selected item at the top. If there are no matching items below the current item, a “wrap” is performed and the highlight bar is positioned to the first item in the list that starts with the pressed character. When there are no items in the entire list that start with the pressed character, no action is taken.

The KEEP phrase in this statement does cause the choice window to remain on the screen. However, since the prior window is selected upon exit from this statement it is possible that that window had UPDATE status specified and it overlays this choice window. In that situation the prior window is displayed on top of this choice window. To avoid this situation, make sure that the prior window either does not overlay this window or that it does not have UPDATE status set.



## Options

There are many options that may be specified when using the WINDOW CHOICE statement. These options can be specified in any sequence. When conflicting options are specified, the last option specified takes precedence. For example TITLE TOP BOTTOM is treated as if the specification were TITLE BOTTOM.

**COLOR** Defines the colors used in the interior of the window.

Color	Code	Color	Code
Black	0	Red	4
Blue	1	Magenta	5
Green	2	Yellow	6
Cyan	3	White	7

Omitting the background color specification causes black to be used for the window background. Omitting the reverse video color specifications causes the reverse of the normal video colors to be used.

Color specifications may be used even when the display is monochrome. For monochrome displays, color specifications are ignored. Use the INVERT option for monochrome displays.

**TITLE** Specifies the text and attributes of the window frame title. When this option is not used, the choice window is displayed without a title.

**TITLE ... TOP or BOTTOM** Specifies the location of the title display: either on the top line of the frame or the bottom line.

**TITLE ... LEFT, CENTER, or RIGHT** Defines the justification of the title text on its display line: left justified, center justified or right justified.

**TITLE ... NORMAL, REVERSE, BLINK, UNDERLINE, and HALF** Defines the video attributes associated with the displayed title text. Multiple attributes may be specified. When there is a conflict with attributes specified (i.e, UNDERLINE, NORMAL) the last one specified takes precedence.

**TITLE ... COLOR** Specifies the color of the title text. Omitting the COLOR specifications for the TITLE causes the default colors of the window to be used. Specifying a foreground color without the background color causes black to be used for the background color.

- INVERT ON or OFF** Causes normal and reverse video attributes to be swapped on monochrome displays. This is the default for odd numbered windows. With [Mode 1](#) and [Mode 3](#), the window number is not known beforehand, therefore, it is best to define the invert status of all choice windows.
- HEIGHT** Specifies the height, in lines, of the choice window created. When this option is not used the height of the window is the smaller of the number of items in the choice list and the number of lines remaining on the screen (screen depth minus starting row).
- HELP** Indicates that the `[F1]` key is active and, when pressed, causes the negative value of the index of the highlighted item to be returned. For example, when the highlight bar is positioned to the third item and `[F1]` is pressed, a -3 is returned as the value of the *index-variable*.
- HELPLINE** Causes an additional one line window to be opened in conjunction with the choice window. This one line window is frameless and is positioned at the bottom of the screen. It uses the full width of the screen, starting at column one.
- The *help-array* must be a string array dimensioned with at least *count* items. As the highlight bar is moved from item to item, this bottom window displays the contents of the matching *help-array* item. A *help-array* item that is a null string removes the one-line help window.
- When the WINDOW CHOICE statement is terminated, the help window is always closed, no matter which mode of the statement is used.
- HELPLINE ... COLOR** Defines the color for the one line help window. Omitting this specification causes the default colors to be used. Since it is unknown which window number will be used for the helpline window, and since the default colors for a window are dependent upon its window number, it is best to always specify the color of the helpline window.
- HELPLINE ... INVERT ON or OFF** Defines the invert status of the one line help window. The default invert status for odd numbered windows is ON. However, since the number of this window cannot be controlled by the program, always specify the invert status desired.

**HOT** Specifies that the items may be positioned to, and selected, with single character input (hot keys). The hot key character for each item is one of the characters in the item text. The position of the character for each item is specified in the numeric *hot-key-array*.

For example, if item five is “Save” and the hot key character is the first letter ‘S,’ then the *hot-key-array* item five will contain the value 1.

Hot key characters are highlighted with the underline attribute (on color terminals, the underline attribute is defined in the class-code and is frequently displayed in a different color than normal video.)

When a text character is entered, the list of hot key characters is searched, from the top, looking for the first hot key character that matches. The search is case insensitive.

**ON.KEY.VALID** Indicates that any current ON KEY traps will remain enabled. Entry of one of those trapped keys will cause the WINDOW CHOICE statement to terminate (with the current highlighted item selected) and the ON KEY trap will be invoked.

If there is a conflict between an ON KEY trap and a key that WINDOW CHOICE normally uses, the ON KEY trap takes precedence.

**USING** Indicates the index of the item in *item-array* that is used for the initial position of the highlight bar. Omitting this option causes the first item in the list to be highlighted.

**WIDTH** Specifies the width, in columns, of the choice window created. When this option is not used the width of the window is the larger of the length of the longest item and the length of the title text.

To have complete control of the color for titles, text and helpline windows, specify each component’s color in the WINDOW CHOICE statement. The default colors used for the various components of a window vary according to the window number, session number, the colors specified in the SETUP CRT environment and the Window Manager setup, making it difficult to predict what the default colors are for any WINDOW CHOICE or WINDOW OPEN statement.

For the same reason, when the program is used on a monochrome display, use the INVERT status for the choice window and the HELPLINE window.



The mouse device may be used to select an item in a WINDOW CHOICE list. Position the mouse cursor is the desired item and click the left mouse button. That item is selected just as if the highlight bar were moved to the item and the **Enter** key was pressed.

Clicking the mouse outside of the WINDOW CHOICE window acts as if the **Esc** key were pressed.

All ON MOUSE routines are disabled while the WINDOW CHOICE statement is executing.



If an ON TIMEOUT statement is in effect and no item is selected for the time period specified, the WINDOW CHOICE statement is exited with a selection of zero and the time-out event handler is invoked.

## Restrictions

In [Mode 2](#), *window* must be an integer value between one and the maximum number of windows.

The choice window must fit on the screen. That is, the *col* and *row* values must be greater than one (to allow the left and top frame to be displayed). The *row* value must be small enough to allow at least three items in the choice list to be displayed, along with the bottom frame and the drop shadow. The *col* value must not be so large that the choice window, its frame and shadow, can not fit on the screen. When the window cannot fit on the screen, a trappable error 60 is reported.

The status line, or 25th line on consoles with 25 lines, cannot be used for any part of the choice window, including the frame and shadow.

The arrays *item-array*, *help-array*, and *hot-key-array* must be dimensioned with at least *count* items. If any of these arrays is allocated with fewer elements, a trappable error 66 is reported: "Array too small for count in WINDOW CHOICE statement."

During execution of the WINDOW CHOICE statement, all ON KEYS are suppressed unless the [ON.KEY.VALID](#) option is used. Any keys previously programmed with [ON KEY](#) traps will return their normal value during a WINDOW CHOICE. Upon termination of the WINDOW CHOICE statement, [ON KEY](#) trapping is re-enabled.

When [ON.KEY.VALID](#) is not used the [ON.KEY.TOKEN](#) value will be cleared by this statement.

This statement cannot be used when i/o redirection is used. Because it is an editing-type operation, attempting to use this statement when stdin or stdout have been redirected causes error 56 to occur.

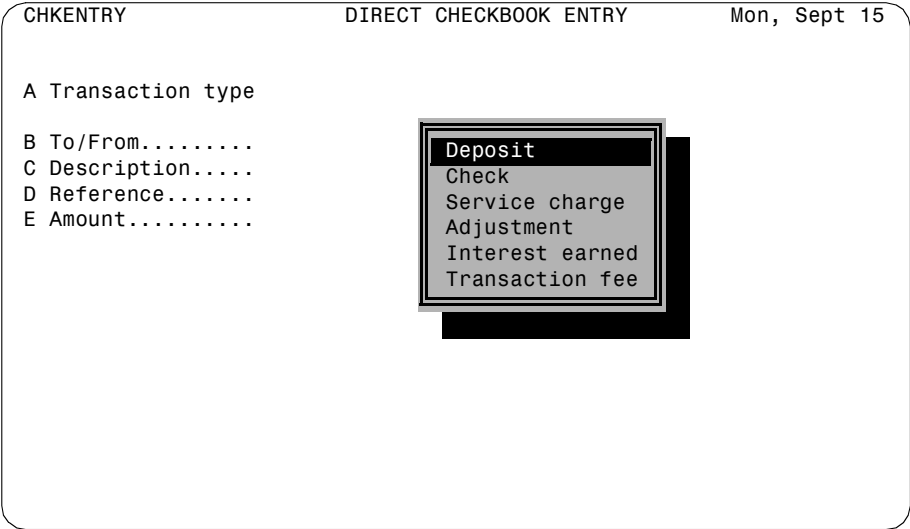
See also [WINDOW OPEN](#), [WINDOW CLOSE](#)

Examples

*Display a choice window starting at 40,6. There are six choices available in the array TYPES\$. Allow the operator to choose one of the choices and return the choice index in the variable SELECTED%.*

```
~
1000 GET.DATA:
1010
1020      WINDOW CHOICE 40,6,6,TYPES$,SELECTED%; COLOR 7,1;INVERT ON
1030
1040      IF SELECTED% \ REM Item selected?
1050          TRAN.TYPES$ = TRIM$(TYPES$(SELECTED%))
1060      ELSE GOSUB REQUIRED.FIELD \ REM Display error message
1070          GOTO GET.DATA \ REM Redo
1080          IFEND
1090
1100      PRINT AT$(20,5);TRAN.TYPES$;
~
```

*After operator selects a choice, the choice window is closed and removed. Display the selected choice on the screen as a field.*



Statements

Refer to the illustration on the following page.

Statements

<i>This routine is</i>	10	OPTION BASE 1, PROMPT " "
<i>invoked when the</i>	20	
<i>"Configuration and</i>	~	
<i>Setup" menu item is</i>	1000	MENU.SELECT.OPTION:
<i>selected.</i>	1010	
<i>Open a choice win-</i>	1020	<b>WINDOW CHOICE 60,8,12,MENU.OPTIONS\$,OPT%; INVERT ON;</b>
<i>dow on the right</i>		<b>COLOR 7,3; HELPLINE MENU.OPTIONS.HELP\$, INVERT ON;</b>
<i>side of the screen.</i>		<b>COLOR 7,2; HOT MENU.OPTIONS.HOT%</b>
<i>Enable the helpline</i>	1030	
<i>and the hot key</i>		
<i>options.</i>		
<i>After choice made</i>	1040	SELECT OPT%
<i>and window</i>	1050	CASE 1
<i>removed, deter-</i>	1060	GOSUB SETUP
<i>mine the action to</i>	1070	CASE 3
<i>take for the choice</i>	1080	GOSUB MENU.MAINTENANCE
<i>selected.</i>	~	
	1210	CEND
	1220	
	1230	RETURN
	~	
	9100	DIM MENU.OPTIONS\$(12)
	9110	DIM MENU.OPTIONS.HELP\$(12)
	9120	DIM MENU.OPTIONS.HOT\$(12)
	9130	
	9140	MAT READ MENU.OPTIONS\$
	9150	MAT READ MENU.OPTIONS.HELP\$
	9160	MAT READ MENU.OPTIONS.HOT%
	~	
<i>Notice that four of</i>	99010	DATA " Setup "
<i>the data items are</i>	99020	DATA " ", " Menus "
<i>null strings. These</i>	99030	DATA " ", " Printers "
<i>null items are non-</i>	99040	DATA " Forms "
<i>choices that dis-</i>	99050	DATA " Reports "
<i>play as horizontal</i>	99060	DATA " ", " Modem "
<i>lines (see illustra-</i>	99070	DATA " ", " Configuration "
<i>tion on next page).</i>	99080	DATA " Terminal Setup "
	99090	
<i>Line 99100, the hel-</i>	99100	DATA "Define company name and address, setup codes, etc."
<i>pline text array</i>	99110	DATA " "
<i>matches the choice</i>	99120	DATA "Maintain menu choices and actions"
<i>array on a one-to-</i>	99130	DATA " "
<i>one basis.</i>	99140	DATA "Define printer attributes"
	99150	DATA "Define report form attributes"
	99160	DATA "Define report fonts, title, etc."
	99170	DATA " "
	99180	DATA "Define modem attributes"
	99190	DATA " "
	99200	DATA "Define options for operator"
	99210	DATA "Define console terminal display attributes"
	99220	

Line 99230, the hot key array identifies the character position in the choice array. Like the helpline array, filler data items are used for the null choice items.

99230 DATA 2,0,2,0,2,2,2,0,4,0,2,2

MENU

MAIN MENU

Mon, Sept 15

Accounts Receivable

Accounts Payable

Payroll

Configuration and Setup

Exit

Setup

Menus

Printers

Forms

Reports

Modem

Configuration

Terminal Setup

Statements

---

## WINDOW CLEAR Statement

WINDOW CLEAR fills the interior of a window with spaces or a user-specified character code.

- 1 **WINDOW CLEAR** *window*
  - 2 **WINDOW CLEAR** *window, char-code*

**Operation**      **Mode 1**—The specified window is cleared to spaces.

**Mode 2**—The specified window is cleared to *char-code*.

**Notes**            *window* does not have to be the active window. However, if *window* is not the active window, is HIDDEN or has UPDATE OFF display status then the effect of the WINDOW CLEAR is not seen until the window is refreshed or selected with UPDATE ON status (refresh will not display a HIDDEN window).

*char-code* is specified as an integer constant or variable and is interpreted as a raw character code value. That is, it is not translated by the operating system or the console class code, it is merely output to the console as a character value similar to a [PUT](#) statement.

Since *char-code* is treated as a raw character code the console class code should be tested to determine if it will be able to display the character you want. For instance, if the console uses the IBM PC character set, you can clear a window to a “stipple pattern” or half intensity spaces by using a *char-code* value of 176, 177 or 178:

```
CC% = VAL(SYS.ENV$(7,"CON"))
IF CC%=22 OR CC%=53 OR CC%=59 OR CC%=60 OR
  (CC%>=90 AND CC%<=99) OR (CC%>=170 AND CC%<=199) OR
  (CC%>=210 AND CC%<=219)
  ! Console is PC Term compatible
  PC.TERM% = -1
  WINDOW CLEAR 4, 176           ! Clear to stipple pattern
ELSE WINDOW CLEAR 4           ! Clear to spaces
  PC.TERM% = 0                 ! Save as false indicator
IFEND
```

**Restrictions**    *window* must refer to an open window.

This statement cannot be used when i/o redirection is used. Because it is an editing-type operation, attempting to use this statement when stdin or stdout have been redirected causes error 56 to occur.



**See also**[CLSS](#) function**Examples**

```
11000 ! Clear background window (0)
11010
11010     IF PC.TERM%
11020         WINDOW CLEAR 0, 176           ! Clear to stipple pattern
11030     ELSE WINDOW CLEAR 0, 32         ! Clear to spaces
11040         IFEND

...
11060     WINDOW CLEAR STATUS.WIN%       ! Clear status window
...
```

---

# WINDOW CLIP Statement

WINDOW CLIP controls whether or not a particular window clips or truncates all characters displayed beyond the right edge of the window.

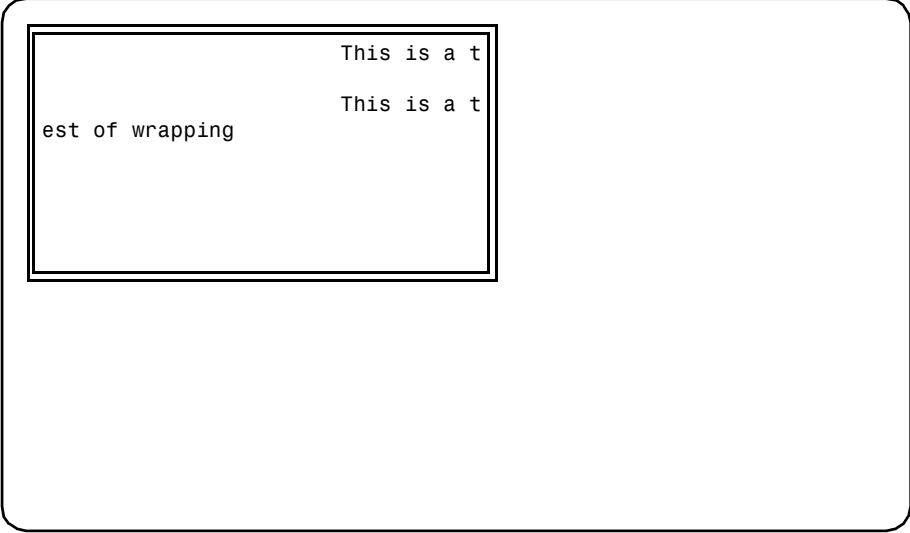
WINDOW CLIP *window*, ON | OFF

Operation	The clipping attribute for the window is set, either on or off, depending upon the option specified.
Notes	<p>A WINDOW CLIP ON means that clipping will occur in the window. Clipping means that characters displayed beyond the right edge of the window are lost (not displayed). A WINDOW CLIP OFF means that the characters are not lost—they are displayed at the beginning of the next line.</p> <p>Setting WINDOW CLIP OFF does not allow you to position beyond the right margin of a window. It only takes affect when you are properly positioned in a window and then display more text than will fit on the line.</p> <p>Changing the clipping attribute for a window does not change the text already displayed in the window.</p>
Restrictions	<p>The <i>window</i> must be an open window. It does not, however, have to be the active window.</p> <p>Window 0 always has CLIP OFF and it cannot be changed with this statement.</p>
See also	<a href="#">WINDOW OPEN</a>

## Examples

This example illustrates the effects of clipping by opening a window with an interior width of 40 columns. Clipping is set on for that window, and text is displayed that falls off the right edge of the window. Then, with clipping set off, more text is displayed near the right side of the window, showing how the text wraps from the end of that line to the start of the next.

```
100      WINDOW OPEN 1,2,2,40,12; FRAME DOUBLE; SELECT
110      WINDOW CLIP 1, ON
120
1010     PRINT AT$(30,1);"This is a test of clipping"
1020
1030     WINDOW CLIP 1, OFF
1040     PRINT AT$(30,3);"This is a test of wrapping"
```



This is a t  
est of wrapping      This is a t

Statements

---

## WINDOW CLOSE Statement

WINDOW CLOSE closes a window or all windows.

- 1 **WINDOW CLOSE** *window*
  - 2 **WINDOW CLOSE ALL**

**Operation**      **Mode 1**—The *window* is closed and memory allocated for the window is released. The window is always removed from the screen when it is closed.

**Mode 2**—All windows are closed and removed from the display. Only window 0 remains on the screen.

**Notes**            Closing the active window causes an underlying window to become the active window. Refer to the discussion of the “[Window Refresh Sequence](#)” on page 614.

**Restrictions**    The *window* must be open.

You cannot close window number 0.

**See also**        [END](#), [KILL](#), [QUIT](#), [STOP](#), [WINDOW OPEN](#), [WINDOW REMOVE](#)

**Examples**        The following two sections of code show typical uses of the WINDOW CLOSE statement. The first section might be used just prior to exiting a portion of a program that has defined and used some windows. It closes five windows used by the program.

```
1140      FOR I% = 8 TO 4 STEP -1
1150          WINDOW CLOSE I%
1160      NEXT
~
```

The second section of code might be used inside a routine that had opened a new window. This code first selects the window that was in use prior to using this window (see [WINDOW STATUS](#) statement) and then closes the window.

```
25220      WINDOW SELECT PRIOR.WIN%, UPDATE ON
25230      WINDOW CLOSE CUR.WIN%
~
```

---

# WINDOW COPY Statement

WINDOW COPY copies a text region from one window and places it in another window.

**WINDOW COPY** *source-window, source-col, source-row, width, depth,*  
*dest-window, dest-col, dest-row*

<b>Operation</b>	<p>A rectangular block of text in the <i>source-window</i> is copied to the <i>dest-window</i>.</p> <p>The upper left corner of the block of text in <i>source-window</i> is identified by <i>source-col</i> and <i>source-row</i>. The block extends to the right for <i>width</i> columns and down for <i>depth</i> rows.</p> <p>A copy of the block of text is made and overlays <i>dest-window</i> starting at <i>dest-col</i> and <i>dest-row</i>.</p>
<b>Notes</b>	<p>The <i>source-window</i> and <i>dest-window</i> may be the same window number.</p> <p>The text copied includes its color and video attributes.</p> <p>Although the <i>source-window</i> is not changed by this statement, if either <i>source-window</i> or <i>dest-window</i> is the active window with update mode on, the active window is refreshed to the screen.</p> <p>The <i>source-col</i>, <i>source-row</i>, <i>dest-col</i>, and <i>dest-row</i> all refer to their respective window origins.</p>
<b>Restrictions</b>	<p>Both the <i>source-window</i> and the <i>dest-window</i> must refer to open windows.</p> <p>The block of text must be contained within the source window and must fit in the destination window.</p>
<b>See also</b>	<p><a href="#">WINDOW GET</a>, <a href="#">WINDOW GET TITLE</a>, <a href="#">WINDOW OPEN</a>, <a href="#">WINDOW TAKE</a></p>

Statements

## Examples

In this example, two windows are opened on the screen. The first one is filled in with some customer name and address information (lines 10 - 70). Later on, the second window is selected and the name and address information from the first window is copied to it (lines 1000 - 1010).

The example illustrates a technique for duplicating some information from one window to another. Although this example has both windows visible for illustration purposes, it is more likely that the second window would overlap and cover the first window.

```
10 WINDOW OPEN 1,2,2,78,20; FRAME SINGLE, COLOR 7,0; COLOR 7,0,0,7
20 WINDOW OPEN 2,5,10,50,10; FRAME DOUBLE, COLOR 7,1; COLOR 7,1,1,7
30
40 WINDOW SELECT 1
50 PRINT AT$(2,2);"Customer: THEOS Software Corporation";
60 PRINT AT$(12,3);"1777 Botelho Drive";
70 PRINT AT$(12,4);"Walnut Creek, CA 94596-5022";
80 PRINT AT$(45,2);"Date: ";DATE$(0);
90 PRINT AT$(45,4);"Balance: ":FORMAT$(BAL,"###,###.##");
~
1000 WINDOW SELECT 2
1010 WINDOW COPY 1,2,2,40,3,2,2,2
~
```

# WINDOW EDIT Statement

The WINDOW EDIT statement allows the operator to edit an array of text strings in a window.

WINDOW EDIT

*window, count, string-array* [ *options* ]

*options*

»

[, QUIT ASK ON | OFF]

»

[, WRAP ON | OFF]

»

[, CASE MIXED | UPPER]

»

[, EDIT INSERT | REPLACE]

**Operations** The first *count* elements of *string-array* are displayed in *window* and the user may edit these strings in a manner similar to a general-purpose text editor.

*string-array* is always treated as a single dimension array. The current **OPTION** BASE is used.

When *window* has a frame and *count* is greater than the window height, a scroll bar is displayed on the right side of the frame to indicate the relative position of the cursor in *string-array*.

When *window* has a frame but does not have a title, the edit status for wrap mode, input case mode and insert/replace mode is displayed in the lower right corner of the frame.

The current CLIP setting for *window* determines how the strings display. If CLIP ON is set the strings are displayed one element per line, truncated to the window width. If CLIP OFF is set the strings are displayed in “wrap” mode.

Fourscore and seven years ago our fathers brought forth on  
this continent a new nation conceived in liberty and dedi-  
cated to the propostion that all men are created equal. Now  
we are engaged in a great civil war testing whether that  
nation, or any nation so conceived and so dedicated, can long  
endure. We are met on a great battlefield of that war. We  
have come to dedicate a portion of that field as a final  
resting-place for those who here gave their lives that that  
nation might live. It is altogether fitting and proper that

WRAP MIXEDCASE INSERT

Statements

In wrap mode as much of a string is displayed on a line as can fit. Words in the string that cannot fit on the line are moved to the next line, which is also wrapped if necessary. This initial display is slightly different from the “wrap paragraph” editing command that can be used by the operator. With the wrap paragraph command the current line and following lines are wrapped, fitting as many words on a line as can fit in the window width. When there is space available at the end of the line, words from the line following are “pulled up” to fill out the line. This process continues until a paragraph break is encountered. A paragraph break is a blank text line.

WINDOW EDIT always restores the active window upon exit. If *window* is selected prior to executing WINDOW EDIT, it will be the selected window after the statement is finished.

## Notes

While editing the text strings, the user may use any of the following keys to move around the text, exit or change the editing options.

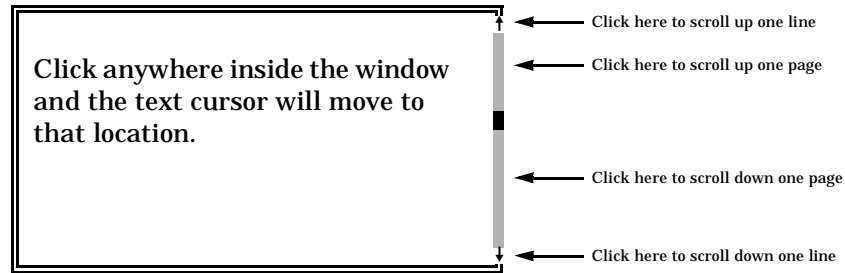
Edit / Position Key	Control Key	Action
<b>Esc</b>		Exit <u>without</u> saving.
<b>Quit</b>	<b>Ctrl</b> + <b>Q</b>	Exit <u>without</u> saving.
<b>File</b>	<b>Ctrl</b> + <b>F</b>	Save and then exit.
<b>Save</b>	<b>Ctrl</b> + <b>S</b>	Save without exiting.
<b>Again</b>	<b>Ctrl</b> + <b>A</b>	Undo all changes since start or last save. This command reformats the <i>string-array</i> according to the current wrap mode.
<b>↑</b>	<b>Ctrl</b> + <b>K</b>	Move up one line, scrolling display if necessary.
<b>↓</b>	<b>Ctrl</b> + <b>J</b>	Move down one line, scrolling display if necessary.
<b>←</b>	<b>Ctrl</b> + <b>H</b>	Move left one character. If at start of current line, move to end of prior line.
<b>→</b>	<b>Ctrl</b> + <b>L</b>	Move right one character. If at right edge of window, move to start of next line.
<b>BegLine</b>	<b>Ctrl</b> + <b>G</b>	Move to start of current line.
<b>Home</b>	<b>Ctrl</b> + <b>T</b>	Move to start of current line. <b>Home</b> , <b>Home</b> moves to start of top line in window. <b>Home</b> , <b>Home</b> , <b>Home</b> moves to start of first line of the text array.
<b>EndLine</b>	<b>Ctrl</b> + <b>E</b>	Move to end of current line.
<b>End</b>	<b>Ctrl</b> + <b>Y</b>	Move to end of current line. <b>End</b> , <b>End</b> moves to end of last line in window. <b>End</b> , <b>End</b> , <b>End</b> moves to end of last line of the text array.



<b>InsLine</b>	<b>Ctrl + V</b>	Insert a blank line above the current line. Move to the start of the inserted line.
<b>Delline</b>	<b>Ctrl + ^</b>	Delete line that cursor is on, pulling up remainder of array.
<b>Enter ↵</b>	<b>Ctrl + M</b>	Always inserts a new line and positions to the start of the new line. When the cursor is not at the end of a line of text, the text is split at the current location with the text to the right of the cursor forming the new line.
<b>← Backspace</b>		Deletes the character to the left of the cursor. If wrap mode is enabled, the paragraph is rewrapped.
<b>Delete</b>	<b>Ctrl + Z</b>	Deletes the character under the cursor. If wrap mode is enabled, the paragraph is rewrapped.
<b>PageDown</b>	<b>Ctrl + P</b>	Display next page of text array.
<b>PageUp</b>	<b>Ctrl + B</b>	Display prior page of text array.
<b>Case</b>	<b>Ctrl + C</b>	Toggle input case mode.
<b>Tab ⇥</b>	<b>Ctrl + I</b>	Insert spaces to advance cursor to next eight column tab stop.
<b>Erase</b>	<b>Ctrl + N</b>	Erase to end of current line.
<b>Insert</b>	<b>Ctrl + R</b>	Toggle insert/replace mode. For most terminals the cursor shape changes to a blinking underline when in replace mode and a blinking block when in insert mode.
<b>WordFwd</b>	<b>Ctrl + X</b>	Advance to the start of the next "word."
<b>WordBack</b>	<b>Ctrl + U</b>	Backup to the start of the word. If at the start of a word then backup to the start of the prior word.
<b>Find</b>	<b>Ctrl + D</b>	Wrap the current paragraph.
<b>SchFwd</b>	<b>Ctrl + W</b>	Toggle wrap/nowrap mode.
<b>Transpose</b>	<b>Ctrl + O</b>	Transpose or swap the two characters under the cursor.



A mouse connected to your terminal can be used to position the cursor to a particular character location. It may also be used to scroll the display up or down.



With wrap mode disabled, inserting characters or tabs pushes characters to the right, possibly beyond the right edge of the *window*. When this occurs those characters are lost. With wrap mode enabled the words pushed off the right side are wrapped to the next line.

### Options

If *window* is specified with CLIP ON the initial status of word wrap is set to off and the individual elements of *string-array* are truncated to the window width.

When *window* has CLIP OFF status the initial status of word wrap is set to on and any elements of *string-array* that are longer than the window width are wrapped to multiple lines. (This operation is also performed when undo command is used.)

When *window* has a frame and the window height is less than *count* a scroll bar is shown on the right side of the window frame.

When *window* has a frame, does not have a title and the window width is greater than 30 columns, the edit option status is shown in the lower right corner of the frame. These status words are: WRAP or NOWRAP, MIXED-CASE or UPPERCASE and INSERT or REPLACE. When the window width is less than 30 columns but more than nine columns, the edit option status is shown with single letters only: W or N, M or C and I or R.

### Defaults

The initial status of the word wrap feature is set according to the window clip status, as described above; the initial status of the insert/replace feature is set to INSERT; the initial status of the input case mode is set to MIXED.

### Restrictions

The *window* must be open and *count* cannot be larger than the dimensioned size of *string-array*.

## Examples

*The window is opened but not selected. **WINDOW EDIT** will select it when it uses it and it will select the prior window upon exit.*

*In this instance, no frame is specified for the window as it is not needed.*

```
PRINT AT$(5,5);"Ship to:";
WINDOW OPEN ADDROF(EDITWIN%),15,5,40,4; FRAME; INVERT ON; CLIP ON
```

```
WINDOW EDIT EDITWIN%,4,ADDR$
```

```
FOR I% = 1 TO 4                                ! Redisplay the array
  PRINT AT$(15,4+I%);RPAD$(ADDR$(I%),40);
NEXT
```

*After the **WINDOW EDIT** finishes, the code redisplay the edited array.*

```
Shipt To:  THEOS Software Corporation
           1801 Oakland Blvd., Suite 315
           Walnut Creek, CA 94596-7000
```

## WINDOW FRAME Statement

WINDOW FRAME changes a window's frame and shadow.

- 1 **WINDOW FRAME** *window*
- 2 **WINDOW FRAME** *window, frame-style* [, *shadow-style*] [, **COLOR**, *fg* [, *bg*]]

---

*frame-style*           »   **NONE**  
                              **SINGLE**  
                              **DOUBLE**  
                              **RAISED**  
                              **SUNKEN**

*shadow-style*       »   **LEFT**  
                              **RIGHT**

**Operation**       **Mode 1**—The *window*'s frame and shadow are removed.

**Mode 2**—The *window*'s frame and shadow are set according to the options used:

Frame-style	Meaning
NONE	Causes the window to have no frame. (It may have a shadow if specified.)
DOUBLE	Sets the frame style to double line.
RAISED	Sets the frame style to single line with a “raised” edge effect.
SINGLE	Sets the frame style to single line.
SUNKEN	Sets the frame style to single line with a “depressed” or “sunken” edge effect.

Shadow-style	Meaning
LEFT	Use a drop shadow on the left side of the window.
RIGHT	Use a drop shadow on the right side of the window.

**Notes**           The *fg* and *bg* color codes for the frame are:

Color	Code	Color	Code
Black	0	Red	4
Blue	1	Magenta	5
Green	2	Yellow	6
Cyan	3	White	7

When the *fg* and *bg* colors are not specified the frame colors revert to the default colors for this window number.

The various window frame styles display as:



SINGLE



DOUBLE



RAISED



SUNKEN



NONE

A RAISED or SUNKEN frame assumes that the “sun” is in the upper left corner of the window. To be consistent, a shadow-style should be RIGHT.

Statements

Specifying a frame style without specifying the shadow position indicates that the window will have a frame without a shadow. For example: WINDOW FRAME SINGLE, COLOR 7,0. To specify a shadow but no frame use a frame-style of NONE.

The FRAME option of the WINDOW OPEN statement provides an easier method for defining the frame of a window. The WINDOW FRAME statement is normally used to change the existing frame style, color, or shadow location of an open window.

Changing the frame style of an active window with UPDATE ON causes that window's frame to be displayed immediately.

Changing the shadow of an active window causes all of the windows that are currently displayed on the screen with UPDATE ON mode, to be refreshed in refresh sequence with this active window refreshed last.

The color and attributes of a shadow is reduced intensity white text on a black background.

Removing the frame of a window removes any associated title.

**Restrictions**      The *window* must refer to an open window.

Although window number 0 is always open it cannot have a frame or shadow, and is invalid for this statement.

A window cannot have a frame if the frame does not physically fit on the screen. A window cannot have a shadow unless the shadow can fit on the screen. The shadow uses two columns on the right (or left) and one row on the bottom.

**See also**            [COLOR](#), [WINDOW OPEN](#)

**Examples**           This example opens a small window, defines the frame style and shadow location, and then waits for input. (Omitting line 1000 causes the window to be displayed and immediately erased.)

```
10 WINDOW OPEN 1,5,5,10,10; COLOR 7,4
20 WINDOW FRAME 1, SINGLE, RIGHT, COLOR 7,4
30 WINDOW SELECT 1
1000 LINPUT USING "!",A$
```

---

## WINDOW GET Statement

WINDOW GET copies text from a window into a program variable.

**WINDOW GET** *window, string-variable-name, length*

**Operation**      The character contents of *window*, starting at the window's current cursor location, is transferred to *str-var*.

**Notes**            The number of characters transferred is the smaller of:

- ▶ The value of *length*
- ▶ The number of columns remaining in *window* to the right of the cursor

**Restrictions**    The *window* must be an open window. It does not, however, have to be the active window.

Only text is transferred with this statement, not color or video attributes.

Only text on the current cursor location line is transferred.

**See also**        [WINDOW OPEN](#), [WINDOW GET TITLE](#)

**Examples**        The following example creates and displays a screen heading window (lines 30-70). A second window is created (line 90). Later in the program (possibly a subroutine), the active window number is saved and the program name field is retrieved from the heading window (lines 1000-1030). After the program name is retrieved, the current window is re-selected (line 1050) and the name is displayed in that window.

Although the WINDOW GET does not require the window to be selected, this code does because it needs to make sure the cursor for that window is positioned at the proper location. Notice that line 1010 performs a select without specifying UPDATE ON. This causes the window to be selected without changing the screen display.

```
10    USER$ = SYS.ENV$(1)
20
30    WINDOW OPEN 1,1,1,80,1; COLOR 7,1,1,7; SELECT
40    PRINT AT(1,1);
50    PRINT USING " '1111111 ' "&RPT$(51,"c")&
      " 'rrrrrrr '1111111","PROGRAM",
      "TEST WINDOWS PROGRAM",USER$,DATE$(0);

60
70    WINDOW REFRESH 1
80
90    WINDOW OPEN 2,5,10,40,3; FRAME SINGLE, COLOR 7,4;
      COLOR 7,4; SELECT
100   PRINT CRT("KOFF");
~
1000  WINDOW STATUS CUR.WIN%
1010  WINDOW SELECT 1
1020  PRINT AT$(2,1);
1030  WINDOW GET 1,PROGRAM.NAME$,8
1040
1050  WINDOW SELECT CUR.WIN%, UPDATE ON
1060  PRINT AT$(2,2);"You are running program: ";
      PROGRAM.NAME$;
1070
1080  WAIT #0 \ GET A$
```



---

# WINDOW GET TITLE Statement

WINDOW GET TITLE copies a window's title to a program variable.

WINDOW GET TITLE *window, string-variable-name*

Operations	<p>The title of <i>window</i> is copied to <i>str-variable</i>.</p> <p>The entire title is copied. If the window does not have a title, then a null string is copied.</p>
Notes	<p>Only the title text is copied, not its color or video attributes. The color, attributes and position of a window title can be determined with the information provided by the <a href="#">WINDOW STATUS</a> statement.</p>
Restrictions	<p>The <i>window</i> must be an open window. It does not, however, have to be the active window.</p>
See also	<p><a href="#">WINDOW GET</a>, <a href="#">WINDOW OPEN</a>, <a href="#">WINDOW TITLE</a>, <a href="#">WINDOW STATUS</a></p>
Examples	<p>In this example section of code, a subroutine gets the current window number (line 8020), the title of that window (line 8030), opens a new window for its own usage (line 8050), and then examines the title of the previous window (line 8070). If the previous window's title indicates that it is a menu window, that window is removed from the display (line 8080); otherwise it is left on the display, underneath this new window.</p>

```
8000 DATE.SELECT:
8010
8020     WINDOW STATUS PRIOR.WINDOW%
8030     WINDOW GET TITLE PRIOR.WINDOW%, PRIOR.TITLE$
8040
8050     WINDOW OPEN 4,32,19,17,1; FRAME SINGLE, RIGHT,
           COLOR YELLOW%,CYAN%;COLOR WHITE%,CYAN%; SELECT UPDATE ON
8060
8070     IF SCH(1,PRIOR.TITLE$,"MENU")
8080         WINDOW REMOVE PRIOR.WINDOW%
8090     IFEND
8100
8110     PRINT AT$(2,1);"Date: ";
8120     X$ = DATE$(0)
8130     PRINT AT$(8,1)
8140     LINPUT USING X$,NEWDATE$
~
```

Statements

# WINDOW INVERT Statement

WINDOW INVERT changes a window's video inversion on monochrome displays.

WINDOW INVERT *window* ON | OFF

Operation	<p>If the display is color-capable, no action is taken.</p> <p>On monochrome displays, the <i>invert</i> status is set.</p> <p><b>INVERT ON</b> Indicates that normal and reverse video is inverted. That is, normal text is displayed with black characters on a white background; reverse video text is displayed white on black.</p> <p><b>INVERT OFF</b> Indicates that normal and reverse video is not inverted.</p>
Defaults	Using this statement without specifying ON or OFF sets the <i>window</i> to INVERT ON.
Notes	<p>The invert status for windows can be set with the <a href="#">WINDOW OPEN</a> statement.</p> <p>When the invert status is not specified during the <a href="#">WINDOW OPEN</a>, and the display is monochrome, the invert status is set for odd numbered windows and reset for even numbered windows. For more information refer to the <a href="#">WINDOW OPEN</a> statement description.</p>
Restrictions	Although this statement is only effective on monochrome displays, no error is reported when used on a color display...the statement has no effect.
See also	<a href="#">WINDOW OPEN</a>

Examples	<pre>1000 WINDOW OPEN 8,30,11,22,3; FRAME; INVERT ON; SELECT 1010 PRINT CRT\$("KOFF");AT\$(2,2);"CHANGE DISKETTE NOW"; 1020 GET ANS% \ REM Accept any char when ready 1030 WHILE ANS%=0 1040     SLEEP 1 \ REM Wait a bit 1050     WINDOW STATUS 8,STATS 1060     IF STATS(23)=0 \ REM Check current invert status 1070         WINDOW INVERT 8, ON 1080     ELSE WINDOW INVERT 8, OFF 1090     IFEND 1100 GET ANS% \ REM Accept any char when ready 1110 WEND</pre>
----------	--

---

# WINDOW LOCATE Statement

WINDOW LOCATE returns a window's cursor location to program variables.

WINDOW LOCATE    *window, col-var-name, row-var-name*

**Operation**            The current cursor location in *window* is returned in the two variables *col-var-name* and *row-var-name* .

**Notes**                The values returned into *col-var-name* and *row-var-name* are relative to the window origin, not the screen.

                          The screen display is not affected by this statement.

**Restrictions**        The *window* must be open. It does not have to be the active window.

**See also**            [WINDOW OPEN](#)

**Examples**            The following section of code could be used at the start of a routine that opens a new window, stacked on top of the current window.

                          The current window number is found with the [WINDOW STATUS](#) statement at line 6020. The cursor location in that window is determined with the WINDOW LOCATE statement at line 6030. It would be the responsibility of the calling routine to position the cursor at the proper location.

                          A new window is opened with line 6050 so that the current line of the prior window is still visible. The new window is drawn immediately below and to the right of that line.

```
6000 NEW.WINDOW:
6010
6020     WINDOW STATUS PRIOR.WIN%
6030     WINDOW LOCATE PRIOR.WIN%,FROM.X%,FROM.Y%
6040
6050     WINDOW OPEN PRIOR.WIN%+1,FROM.X%+2,FROM.Y%+2,40,5
6060
~
```

Statements

---

## WINDOW MOVE Statement

WINDOW MOVE changes the displayed position of a window.

**WINDOW MOVE** *window, col, row*

<b>Operation</b>	The <i>window</i> is moved. The new upper left corner is at <i>col, row</i> .
<b>Notes</b>	<p>If <i>window</i> is the active window with UPDATE ON, then the screen is refreshed to reflect the new location of this window.</p> <p>When <i>window</i> is not the active window, the screen is not updated.</p>
<b>Restrictions</b>	<p>The <i>window</i> must be open. It does not have to be the active window.</p> <p>Window number 0 cannot be moved.</p> <p>The new location of the window, including any frame and drop shadow, must fit on the screen.</p>
<b>See also</b>	<a href="#">WINDOW OPEN</a> , <a href="#">WINDOW RESTORE</a>

**Examples**      The following section of a program shows the WINDOW MOVE statement being used to position a “help display” window to the appropriate quadrant of the screen.

```
100      WINDOW OPEN 9,1,1,38,6;COLOR WHITE%,GREEN%
~
1000  HELP.DISPLAY:
1010
1020      SELECT POSITION%
1030          CASE 1
1040              WINDOW MOVE 9,1,1 \ REM Upper left quadrant
1050          CASE 2
1060              WINDOW MOVE 9,41,1 \ REM Upper right quadrant
1070          CASE 3
1080              WINDOW MOVE 9,41,12 \ REM Lower right quadrant
1090          CASE 4
1100              WINDOW MOVE 9,1,12 \ REM Lower left quadrant
1110          OTHERWISE
1120              WINDOW MOVE 9,1,1 \ REM Upper left quadrant
1130          CEND
1140
1150      WINDOW SELECT 9, UPDATE 0
```

### WINDOW OPEN Statement

**WINDOW OPEN** opens and defines a window. Optionally, this statement also defines all of the initial attributes for the window.

```
1 WINDOW OPEN window, col, row, width, depth[options]
```

2 **WINDOW OPEN ADDROF**( *num-var* ), *col*, *row*, *width*, *depth* [ *options* ]

```

options
»   [; FRAME
        [, NONE | SINGLE | DOUBLE | RAISED | SUNKEN ]
        [, LEFT | RIGHT]
        [, COLOR fg[, bg]]

»   [; TITLE title-exp
        [, TOP | BOTTOM]
        [, LEFT | CENTER | RIGHT]
        [, NORMAL | REVERSE | BLINK | UNDERLINE | HALF]
        [, COLOR fg[, bg]]

»   [; INVERT ON | OFF]

»   [; COLOR fg[, bg[, rv-fg[, rv-bg]]]]

»   [; CLIP ON | OFF]

»   [; SELECT [ UPDATE ON | UPDATE OFF ]
        [ HIDDEN | TOP ]]

```

**Operation**      **Mode 1**—A window is opened, or reopened, as window number *window*.

**Mode 2**—A new window is opened. The window number of this new window is the first available, unused window number. The window number value is assigned to *num-var*.

The window is rectangular with the upper left corner of the interior of the window at *col*, *row*. The width (number of columns) and depth (number of rows) of the window is specified by *width* and *depth* respectively.

**Options**

There are many options that may be specified when using [Mode 2](#) of the WINDOW OPEN statement. These options may be specified in any sequence. When conflicting options are specified the last option specified takes precedence. For example TITLE TOP BOTTOM is treated as if the specification were TITLE BOTTOM.

<b>CLIP ON</b>	Text flowing beyond the right edge of the window will be clipped or truncated. This is the default.
<b>CLIP OFF</b>	Text flowing beyond the right edge of the window is wrapped to the next line.
<b>COLOR</b>	Defines the colors used in the interior of the window.
<b>FRAME</b>	Specifies the frame style. Using the FRAME option by itself causes the window to have no frame, shadow, or title.  Refer to the <a href="#">WINDOW FRAME</a> statement for examples of the various frame styles.
<b>... NONE</b>	Causes the window to have no frame. (It may have a shadow if specified.)
<b>... DOUBLE</b>	Sets the frame style to double line.
<b>... RAISED</b>	Sets the frame style to single line with a “raised” edge effect.
<b>... SINGLE</b>	Sets the frame style to single line.
<b>... SUNKEN</b>	Sets the frame style to single line with a “depressed” or “sunken” edge effect.
<b>... COLOR</b>	Followed by one or two color codes, sets the colors used for the frame.
<b>... LEFT</b>	Use a drop shadow on the left side of the window.
<b>... RIGHT</b>	Use a drop shadow on the right side of the window.
<b>INVERT</b>	Specifies the “colors” to use when the console is monochrome.
<b>... OFF</b>	The window uses white text on a black background. This is the default for <u>even</u> numbered windows.
<b>... ON</b>	The window uses black text on a white background. This is the default for <u>odd</u> numbered windows.

**SELECT** Specifies whether this is the active window and whether or not changes to the contents of the window are updated on the screen. Specifying **SELECT** by itself implies **SELECT UPDATE ON**.

Refer to the [WINDOW SELECT](#) statement for examples of the effects of various selection modes.

**... UPDATE OFF** The window is active but changes are not displayed at this time.

**... UPDATE ON** The window is active, the display is refreshed if necessary and subsequent changes are immediately displayed on the screen.

**... HIDDEN** The window is active with **UPDATE OFF** status and the window is not visible on the screen.

**... TOP** The window is active and appears on the screen on top of all other windows. This mode can be used in combination with **UPDATE OFF** or **UPDATE ON** but not with **HIDDEN**.

**TITLE** Specifies the text and attributes of the window title. Since the title appears in the frame area of a window, specifying a title implies a **FRAME** specification whether it is explicitly stated or not. (When the **FRAME** is not specified and **TITLE** is, a default frame of **FRAME DOUBLE** is used.

A **TITLE** specification with no other title attributes specified defaults to a **TITLE text**, **TOP**, **CENTER**, **NORMAL** specification.

**... LEFT** The title text appears on the left side of the frame.

**... CENTER** The title text appears in the center of the frame.

**... RIGHT** The title text appears on the right side of the frame.

**... BOTTOM** The title text appears on the bottom frame.

**... TOP** The title text appears on the top frame.

**... NORMAL** Display the title text with no special video attributes.

**... REVERSE** Display the title text in reverse video.

**... BLINK** Display a blinking title.

- ... **UNDERLINE**    The title text is underlined.
- ... **HALF**        The title is displayed in reduced intensity.
- ... **COLOR**       Specifies the foreground and background colors for the title text. Omitting the COLOR specifications for the TITLE causes the default colors of the frame to be used (not the specified colors of the frame).

**Notes**            The *col* and *row* parameters refer to the upper left corner of the interior of the window. That is, these coordinates exclude the frame. The *width* and *depth* parameters also refer to the interior of the window.

Window number 0 is always open as a full-screen window with no frame.

To open a window with a frame but no shadow, specify the FRAME option and SINGLE, DOUBLE, RAISED or SUNKEN but do not include the LEFT or RIGHT parameters. To open a window with a shadow but no frame, use a FRAME option of NONE and a LEFT or RIGHT specification for the shadow.

To have complete control of the color for frames, titles and text, specify each component's color in the WINDOW OPEN statement. The default colors used for the various components of a window vary according to the window number, session number, the colors specified in the SETUP CRT environment and the Window Manager setup, making it difficult to predict what those colors will be for any given WINDOW OPEN statement.

A window can be re-opened with this statement. If *window* refers to an open window, that window is first closed and then the open is performed. This is the only way a window's size can be changed. Note however, that re-opening a window clears its contents.

**Defaults**        Unless options are used to specify the attributes of the window, the default attributes of the window are: FRAME DOUBLE; CLIP ON; SELECT UPDATE ON. If the window and frame do not fit on the screen, the frame is suppressed. There is no shadow or title text. The default color for the frame and interior is the same and is determined by the Window Manager based upon the current color definitions for your terminal, the session number, and the number of the window being opened.



**Special Note**

Because the WINDOW OPEN statement has multiple clauses, it is possible to have multiple errors in the specifications. For example, you might specify a full-screen window and specify that it has a frame with drop shadow and title:

```
WINDOW OPEN 1,1,1,80,24; FRAME SINGLE, LEFT; TITLE " SAMPLE "
```

Here, the window can fit on the screen but not with a frame or with a drop shadow. The actions taken depend upon the [ON ERROR](#) statement. When there is no error trap defined at the time the statement is executed, the error is detected (error code 55, "Invalid window size specified") and the program is terminated.

When [ON ERROR](#) trapping is in effect the operation is a little different due to the operation of error trapping. Without error trapping MultiUser BASIC traps and reports the error when it occurs. With error trapping MultiUser BASIC detects the error and sets an error indicator. The error indicator is examined only after statements are fully executed. For most statements this difference is totally transparent.

Since the WINDOW OPEN statement is really a compound statement the error is detected and the indicator is set when the FRAME clause is processed. But, since the statement is not finished executing, the trap is not invoked. Instead, the window is set to no frame (or shadow) and the next clause is processed. The TITLE clause is now an error because the window has no frame. The title is not set nor is the error code changed from the original error code. Since the TITLE clause is the last part of the WINDOW OPEN statement the window is opened and processing continues. (The window would not be opened if the interior of the window could not fit on the screen.)

At this point the error is trapped by your routine and the ERR function returns the code for the first error detected during the processing of the WINDOW OPEN statement.

**Restrictions**

*window* must be a value between 1 and the total number of windows available (see [TOTAL.WINDOWS](#) function). The window must be able to physically fit on the screen.

The status line or 25th line on consoles with 25 lines, cannot be used for any part of a window, including the frame and shadow.

You must have sufficient memory resources available for the system to create this window in memory.

This statement cannot be used when i/o redirection is used. Attempting to use this statement when stdin or stdout have been redirected causes error 56 to occur.

## See also

[AVAIL.WINDOWS](#), [TOTAL.WINDOWS](#), [USED.WINDOWS](#), [WINDOW CHOICE](#), [WINDOW FRAME](#), [WINDOW INVERT](#), [WINDOW MOVE](#), [WINDOW SELECT](#), [WINDOW STATUS](#), [WINDOW TITLE](#)

## Examples

In this example, taken from a program's initialization routine, four windows are opened. Window numbers 1, 2, and 3 are defined with colors but no frame, title, or shadow (lines 9110–9130). Since they are not specified as `SELECT UPDATE ON` and no [WINDOW REFRESH](#) is performed, these windows are not displayed on the screen at this time.

Window number 4 (line 9140) is defined to be a window in the middle of the screen with a double line frame and drop shadow to the right. No title is given to the window. It is also not selected so it is not displayed at this time.

All four windows have the default clipping attribute of `CLIP ON`.

The screen title window (window number 2) is selected (line 9160) causing it to display on the screen.

```
~
9100     PRINT CLS$;CRT$("KOFF");           ! Init window 0
9110     WINDOW OPEN 1,1,2,80,21; COLOR 7,1,1,7
9120     WINDOW OPEN 2,1,1,80,1; COLOR 7,1,1,7
9130     WINDOW OPEN 3,1,23,80,2; COLOR 7,2,2,7
9140     WINDOW OPEN 4,18,11,42,3; FRAME DOUBLE, RIGHT, COLOR 7,1;
        COLOR 7,1,1,7
9150
9160     WINDOW SELECT 2, UPDATE ON
9170     PRINT AT$(2,1);FROM.PROG$;
~
```

This next example shows a user-defined function whose purpose is to display help message text.

The window number of the currently active window is saved and a new window is opened to display the help text. This new help window has a single-line raised frame with a drop shadow to the right; a title that uses the default `TOP, CENTER` position, and is the active window. All components of the window are colored with a white foreground on a green background.

After the help text is displayed and the operator releases the display, the prior window is re-selected (line 800300). The help window is not closed by this function. Instead, since it was opened with the `TOP` option, it remains on the screen fully visible even though the prior window is selected. This allows the help text to be viewable during the data input for the field that the help describes. The window number of the help display is returned as

the value of the function, allowing the calling program to close (and remove) it when it is appropriate.

<i>This first function</i>	800000	DEF FN.HELP%(HELP.FILE\$,FX%,FY%,W%,D%)
<i>opens a "help" win-</i>	800010	
<i>dow with a single</i>		
<i>line raised-effect</i>		
<i>frame and a drop</i>		
<i>shadow on the</i>		
<i>right. The window</i>		
<i>includes a title and</i>		
<i>the frame, title and</i>		
<i>window interior are</i>		
<i>all white text on a</i>		
<i>green background.</i>		
 <i>The HELP.WIN% is</i>	800020	LOCAL HELP.WIN%, PRIOR.TO.HELP%
<i>returned so the</i>	800030	
<i>calling program</i>	800040	HELP.WIN% = 9
<i>can close it when it</i>	800050	WINDOW STATUS PRIOR.TO.HELP%
<i>wants to.</i>	800060	OPEN #30: HELP.FILE\$, INPUT SEQUENTIAL
	800070	
	800080	<b>WINDOW OPEN HELP.WIN%,FX%,FY%,W%,D%;</b>
		<b>FRAME RAISED, RIGHT, COLOR WHITE%,GREEN%;</b>
		<b>TITLE " HELP ", COLOR WHITE%,GREEN%;</b>
		<b>COLOR WHITE%,GREEN%; SELECT UPDATE ON, TOP</b>
	800090	
	~	
	800290	CLOSE #30
	800300	WINDOW SELECT PRIOR.TO.HELP%, UPDATE ON
	800310	
	800320	FN.HELP% = HELP.WIN%! Return window # used
	800330	
	800340	FNEND

Statements

*This function displays a message in a temporary window. Because it is temporary, the program doesn't need to use a specific window number. It uses the **ADDROW OPEN** feature of **WINDOW OPEN** to request that the next available window number be assigned.*

*The window is centered on the screen with yellow text on a blue background.*

```

801000 DEF FN.MESSAGE$(MESSAGE.TEXT$)
801010     LOCAL PRIOR.WIN%           ! Temp variables
801020     LOCAL MSG.WIN%
801030     LOCAL WIDTH%
801040     LOCAL REPLY$
801050
801060     WIDTH% = MAX(2+LEN(MESSAGE.TEXT$),27)! Room for prompt
801070
801080     WINDOW STATUS PRIOR.WIN% ! Save so we can reselect
801090
801100     WINDOW OPEN ADDROW(MSG.WIN%),
            (LINE(0)-WIDTH%)/2,(PAGE(0)-5)/2+2,WIDTH%,3;
            FRAME SINGLE, RIGHT, COLOR WHITE%, YELLOW%;
            TITLE " PRESS ANY KEY TO CONTINUE ", BOTTOM, CENTER,
            COLOR YELLOW%, BLUE%;
            COLOR YELLOW%, BLUE%; SELECT UPDATE ON
801110
801120     PRINT AT$(1,2);CRT$("KOFF");MESSAGE.TEXT$;! Display
801130
801140     WAIT #0: \ GET #0: REPLY$ ! Get reply for release
801150
801160     WINDOW SELECT PRIOR.WIN% ! Select the prior window
801170
801180     WINDOW CLOSE MSG.WIN% ! Close the window
801190
801200     LET FN.MESSAGE$ = REPLY$ ! Return their response
801210
801220     FNEEND

```

---

# WINDOW REFRESH Statement

WINDOW REFRESH displays a window without making it active.

WINDOW REFRESH    *window*

**Operation**            The indicated *window* is refreshed on the screen by displaying it on top of all other displayed windows.

**Notes**                    This statement does not SELECT *window*. If the active window partially overlays this window, then that portion of the window is not refreshed.

                              The *window* is placed at the top of the window refresh sequence, unless the active window has UPDATE ON, in which case, *window* is placed at the top of the window refresh sequence, but below the active window. Refer to the discussion of the “[Window Refresh Sequence](#)” on page 614.

                              All other displayed windows that are at least partially visible, are also refreshed.

**Restrictions**            The *window* must be an open window.

                              A HIDDEN window cannot be refreshed.

**See also**                [WINDOW OPEN](#), [WINDOW SELECT](#)

**Examples**                This example is taken from a calendar and appointment scheduling program. Four subroutines are called (lines 10320–10350) that output various information on the screen. The program is designed such that the windows are opened with UPDATE OFF. The information is displayed in the windows but not displayed on the screen. This was done to avoid a “jerky” look to the screen as each of the components was displayed on the screen.

                              The WINDOW REFRESH statement is used to display all of the collected information at once (lines 10370–10380).

```
~
10320        GOSUB SHOW.DAY
10330        GOSUB SHOW.NOTES
10340        GOSUB SHOW.ANNIV
10350        GOSUB SHOW.DAY.SCHEDULE
10360
10370        WINDOW REFRESH CAL.WIN%
10380        WINDOW REFRESH APPOINT.WIN%
```

---

# WINDOW REMOVE Statement

WINDOW REMOVE removes or erases a window from the display.

WINDOW REMOVE    *window*

Operation	The indicated <i>window</i> is removed from the screen display. All regions of the screen underlying this window are updated to reflect this removed window.
Notes	<p>This statement is the complement of the <a href="#">WINDOW REFRESH</a> statement. It does <u>not</u> CLOSE the window, it merely removes it from the display and places <i>window</i> at the bottom of the window refresh sequence using a special code indicating that it is not displayed on the screen. The contents of the <i>window</i> are not affected.</p> <p>Removed windows are not redisplayed on the screen unless they are selected (see <a href="#">WINDOW SELECT</a> statement) or refreshed (see <a href="#">WINDOW REFRESH</a> statement). Refer to the discussion of the “<a href="#">Window Refresh Sequence</a>” on page 614.</p> <p>Removing the active window causes a window beneath this window in the refresh sequence to be selected as the active window.</p>
Restrictions	The <i>window</i> must be open.
See also	<a href="#">WINDOW CLOSE</a> , <a href="#">WINDOW OPEN</a> , <a href="#">WINDOW REFRESH</a>

## Examples

This example shows a “quit key” event trap. For an explanation refer to the ON KEY statement.

Of interest here is line 700100. After the operator’s response, the program needs to remove the event window. It first selects another window and then removes this window. If it had not selected another window prior to removing this one (line 700090), window 0 might have been selected.

```
700000 QUIT.KEY:
700010
700020     WINDOW STATUS PRIOR.TO.QUIT%
700030
700040     WINDOW SELECT MESSAGE.WIN%, UPDATE ON
700050
700060     PRINT AT$(2,2);"Do you really wish to QUIT? ";
700070     REPLY$ = YESNO$
700080
700090     WINDOW SELECT PRIOR.TO.QUIT%, UPDATE ON
700100     WINDOW REMOVE MESSAGE.WIN%
700110
700120     IF REPLY$="N" RESUME ELSE RESUME END.RUN
```

---

## WINDOW RESTORE Statement

WINDOW RESTORE sets a window to the attributes and contents of a previously saved window.

- 1 **WINDOW RESTORE** *window*, *file-spec*
  - 2 **WINDOW RESTORE** *window*, **USING** *string*

**Operation**      **Mode 1**—Open or reopen *window* and set its attributes and contents from the window saved in the file *file-spec*.

**Mode 2**—Open or reopen *window* and set its attributes and contents from the window saved in the variable *string*.

**Notes**            The restored *window* has all of the attributes of the saved window including:

- ▶ Origin column and row
- ▶ Width and depth
- ▶ Cursor location and style
- ▶ Frame style, attributes and color
- ▶ Shadow position
- ▶ Title position, alignment, attributes and color
- ▶ Clip status
- ▶ Invert status for monochrome displays
- ▶ Display type (hidden, top or neither)
- ▶ Interior contents including text, attributes and color

*window* does not have to be an open window. If *window* is open this statement reopens it with the attributes and contents specified in the saved window.

The *window* is restored but not displayed with this statement. You must [WINDOW REFRESH](#) or [WINDOW SELECT](#) the window to display it on the screen.



A window created and saved on a color display and later restored to a monochrome display will be restored with no color and the INVERT status is restored even though it was not effective on the color display.

This statement cannot be used when i/o redirection is used. Attempting to use this statement when stdin or stdout have been redirected causes error 56 to occur.

#### See also

[WINDOW OPEN](#), [WINDOW SAVE](#), [WINDOW STATUS](#)

#### Examples

The previously saved contents of a program copyright display window is restored. After restoring that window, it is refreshed to the screen. Refer to the [WINDOW SAVE](#) statement description for an example showing the creation of the "COPYRITE.DISPLAY" file.

```
10      WINDOW RESTORE 9, "COPYRITE.DISPLAY"
20      WINDOW REFRESH 9      ! Display copyright message
30
40      GOSUB PROGRAM.SETUP
50
60      WINDOW CLOSE 9
~
```

The following code section restores a window that was previously saved in a string variable (see [WINDOW SAVE](#) statement example). Note that the variable WINDOW1\$ is used both as a storage area and as a switch indicating that it is used.

```
2000      IF WINDOW1$
2010          WINDOW RESTORE 1, USING WINDOW1$      ! Restore window 1
2020          WINDOW REFRESH 1                      ! Display it
2030          WINDOW1$ = ""                          ! Clear save string
2040          IFEND
```

Statements

---

# WINDOW SAVE Statement

WINDOW SAVE saves all of a window's attributes and contents in a file or to a string.

- 1

WINDOW SAVE *window*, *file-spec*
- 2

WINDOW SAVE *window*, USING *string*

**Operation**      **Mode 1**—The *window* attributes and contents are written to the file *file-spec*. Any previous contents of the file are overwritten by this statement.

**Mode 2**—The *window* attributes and contents are saved in the variable *string*.

**Notes**            The complete specifications for *window* are written to the file *file-spec* or to *string*, including:

- ▶ Origin column and row
- ▶ Width and depth
- ▶ Cursor location and style
- ▶ Frame style, attributes and color
- ▶ Shadow position
- ▶ Title position, alignment, attributes and color
- ▶ Clip status
- ▶ Invert status for monochrome displays
- ▶ Display type (hidden, top or neither)
- ▶ Interior contents including text, attributes and color

The [WINDOW RESTORE](#) statement is used to retrieve this saved information.

**Restrictions**    The *window* must be open. It does not have to be the active window.

**See also**        [WINDOW OPEN](#), [WINDOW RESTORE](#), [WINDOW STATUS](#)

## Examples

This example creates and saves a program copyright message window.

```
10      WINDOW OPEN 1,19,10,40,7; COLOR 7,1;
        FRAME SINGLE, RIGHT,COLOR 7,1
20
30      PRINT AT$(11,2);"The EXPERT System";
40      PRINT AT$(8,4);"Version 1.3a 19 July 1994";
50      PRINT AT$(5,6);"Copyright 1992 by ABC Software";
60
70      WINDOW SAVE 1,"COPYRITE.DISPLAY"
80
90      WINDOW CLOSE 1
```

This second example shows a program that needs to open another window when all of the available windows are already in use.

```
840      IF AVAIL.WINDOWS = 0           ! No windows available?
850          WINDOW SAVE 1, USING WINDOW1$      ! Save window 1
860          WINDOW CLOSE 1               ! Close window 1
870          IFEND
880
890      ! Open new window, display contents, let operator see it,
900      ! then close the window.
910
920      ...

2000     IF WINDOW1$
2010         WINDOW RESTORE 1, USING WINDOW1$      ! Restore window 1
2020         WINDOW REFRESH 1                      ! Display it
2030         WINDOW1$ = ""                        ! Clear save string
2040         IFEND
```

## WINDOW SELECT Statement

WINDOW SELECT activates a window for subsequent input and output.

- 1 **WINDOW SELECT** *window*
- 2 **WINDOW SELECT** *window, update-mode* [ , *display-type*]

---

<i>update-mode</i>	»	<b>UPDATE OFF</b> <b>UPDATE ON</b>
<i>display-type</i>	»	<b>HIDDEN</b> <b>TOP</b>

**Operation**      **Mode 1**—The open window number *window* is selected as the active window. All subsequent input from and output to the console is displayed inside the boundaries of this window. The *update-mode* is set to UPDATE ON and it is not HIDDEN or TOP.

**Mode 2**—The open *window* is selected as the active window and its display status is set according to the *update-mode* and *display-type* specification. All subsequent input from and output to the console is within the boundaries of this *window*.

**Notes**      The meanings of the *update-mode* and *display-type* specification are:

Option	Meaning
UPDATE OFF	The window is active but changes are not displayed at this time.
UPDATE ON	The window is active, the display is refreshed if necessary and subsequent changes are immediately displayed on the screen.
... HIDDEN	The window is active with UPDATE OFF status and the window is not visible on the screen.
... TOP	The window is active and appears on the screen on top of all other windows. This mode can be used in combination with UPDATE OFF.

When a window is selected, the previous state of the window takes effect for the following attributes: cursor location, cursor on/off, video attributes, normal and reverse video text colors.

The first time a window is selected it has a cursor location of 1,1.

Window number 0 is always open and can be selected with this statement but it cannot be specified as TOP.

**HIDDEN mode** The HIDDEN attribute of a window causes that window to be hidden from view at all times. Specifically, a hidden window is not placed in the “refresh sequence” (see discussion, below) and is not displayed on the screen.

The visible portions of a window previously selected with UPDATE OFF but not HIDDEN status remain displayed when another window is selected. No portion of a hidden window is ever displayed unless it is specifically refreshed or is selected without the HIDDEN status.

**TOP mode** Selecting a window with UPDATE ON but not HIDDEN or TOP causes that window to be displayed on the screen on top of all other windows except those marked as TOP. Selecting a window with the TOP attribute causes that window to be displayed on the screen on top of all other windows, including other windows marked as TOP.

**Window 0** Window 0 is different than other windows in several ways. It is a window that:

- ▶ Is always open and cannot be closed
- ▶ Has no frame, title or shadow
- ▶ Covers the entire screen (except the status line, if any)
- ▶ Has text clipping on and cannot be changed
- ▶ When selected covers all other windows including those marked as TOP
- ▶ When window 0 is selected and then a window other than window 0 is selected and there is at least one window marked as TOP, window 0 becomes HIDDEN.

Window 0 can be selected with the HIDDEN attribute to keep it from appearing even as a background to other windows. (When the interpreter selects window 0 to display messages or to allow you to use the command mode, it is always selected without the HIDDEN attribute.)

## ■ Window Refresh Sequence

Window Manager maintains a list of all of the open windows in their display sequence (the “refresh stack”). Thus, if the following windows are open and selected with:

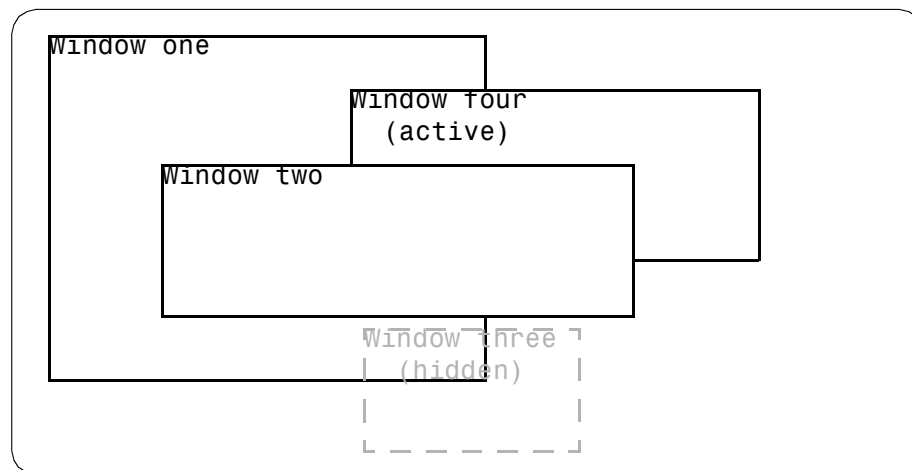
```
WINDOW SELECT 1
WINDOW SELECT 2, UPDATE OFF, TOP
WINDOW SELECT 3, HIDDEN
WINDOW SELECT 4, UPDATE ON
```

the refresh stack contains:

Refresh Sequence	Window #
3	2 (top)
2	4 (active)
1	1
0	0
-1	3 (hidden)

Window number three is not truly in the refresh sequence because it is HIDDEN. It is in the refresh stack but it is marked with a special code (-1) indicating that it is not displayed.

The display might appear as:



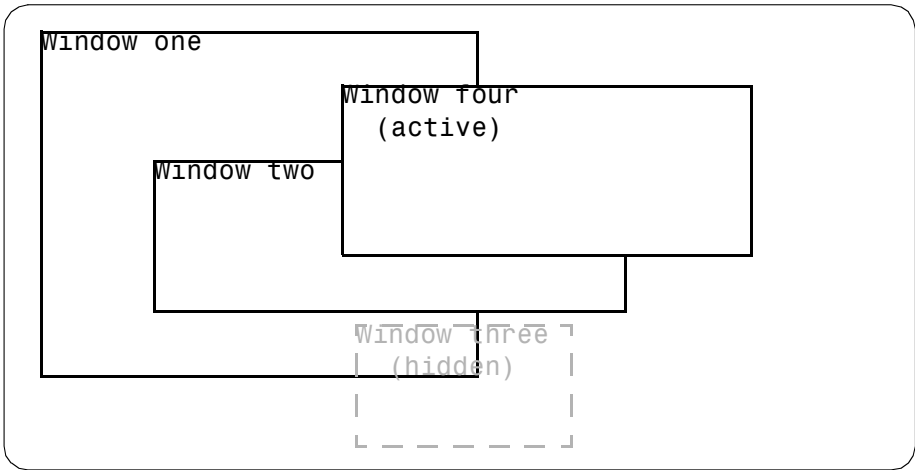
Even though window four is the active window and it covers part of the area used by window two, window two is still displayed because it was last selected with the TOP attribute. To make the text in window four that is behind window two visible, window four will have to be selected with the

TOP attribute, window two will have to either be selected again without the TOP attribute (and then reselect window four) or window two will have to be removed or closed.

If window two is reselected without the TOP attribute and then window four is selected the refresh stack changes to:

Before:	Refresh Sequence	Window #	After:	Refresh Sequence	Window #
	3	2 (top)		3	4 (active)
	2	4 (active)		2	2
	1	1		1	1
	0	0		0	0
	-1	3 (hidden)		-1	3 (hidden)

and window two is now partially overlaid by window four.



Statements

Removing a window from the display removes the window number from the refresh sequence just as if it had never been selected for display. For instance, if a `WINDOW REMOVE 2` is performed the refresh stack contains:

Before:	Refresh Sequence	Window #	After:	Refresh Sequence	Window #
	3	4 (active)		3	
	2	2		2	4 (active)
	1	1		1	1
	0	0		0	0

Refresh Sequence	Window #
-1	3 (hidden)

Refresh Sequence	Window #
-1	3 (hidden)

No portion of window two will be visible on the screen but it will reappear if it is selected or refreshed. To remove a window from the display and to keep it removed until the program specifically requests its display, select it with the HIDDEN attribute.

Removing or closing the active window causes the window number below the removed or closed window to become the active window. For instance, using the original window display and refresh sequence shown on page 614, if window four is closed then window one becomes the active window:

Before:	<b>Refresh Sequence</b>	<b>Window #</b>	After:	<b>Refresh Sequence</b>	<b>Window #</b>
	3	2 (top)		3	
	2	4 (active)		2	2 (top)
	1	1		1	1 (active)
	0	0		0	0
	-1	3 (hidden)		-1	3 (hidden)

- Defaults

When no *update-mode* is specified ([Mode 1](#)) the window is selected with UPDATE ON status, is not HIDDEN and does not have the TOP attribute set.
- Restrictions

*window* must be an open window.
- See also

[WINDOW CHOICE](#), [WINDOW OPEN](#), [WINDOW REFRESH](#)
- Examples

In this example, several windows are opened (lines 9110–9140) and then one of them is selected as the active window (line 9160).

```
~
9100 PRINT CLS$;CRT$("KOFF"); \ REM Init window 0
9110 WINDOW OPEN 1,2,80,21; COLOR 7,1,1,7
9120 WINDOW OPEN 2,1,1,80,1; COLOR 7,1,1,7
9130 WINDOW OPEN 3,1,23,80,2; COLOR 7,2,2,7
9140 WINDOW OPEN 4,18,11,42,3; FRAME DOUBLE, RIGHT,
      COLOR 7,1; COLOR 7,1,1,7
9150
9160 WINDOW SELECT 2
9170 PRINT AT$(2,1);PROG.NAMES$;
~
```



# WINDOW STATUS Statement

WINDOW STATUS returns the number of the active window or the complete status of any window.

- 1

WINDOW STATUS    *numeric-variable*
- 2

WINDOW STATUS    *window, numeric-array*

**Operation**            **Mode 1**—The value of *numeric-variable* is set to the window number of the active window.

**Mode 2**—The first 23 elements of *numeric-array* are filled with the attributes of the window.

**Notes**                    After [Mode 2](#) of this statement is executed the contents of *numeric-array* will be:

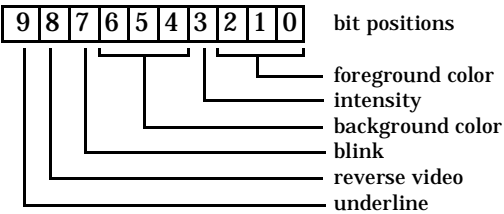
Index	Description	Codes used
1	Window status:	0 = Not open
		1 = Open, not active
		2 = Open, active
2	Window origin, column	
3	Window origin, row	
4	Window width	
5	Window depth	
6	Cursor column location	
7	Cursor row location	
8	Cursor type:	0 = Not displayed
		1 = Blinking underline
		2 = Blinking block
		3 = Steady underline
		4 = Steady block
<sup>1</sup> These codes are bit-mapped. That is, each code represents a single bit turned on or off. The codes for a field can be combined.		
<sup>2</sup> The update code is bit-mapped.		

Statements

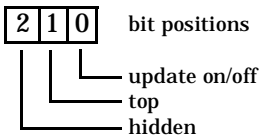
Index	Description	Codes used
9	Frame type:	0 = None
		1 = Single line
		2 = Double line
		3 = Raised, single line
		4 = Sunken, single line
10	Shadow type:	0 = None
		1 = Right side
		2 = Left side
11	Title type:	0 = None
		1 = Top frame line
		2 = Bottom frame line
12	Title alignment:	0 = Center justified
		1 = Left justified
		2 = Right justified
13	Title attributes <sup>1</sup> :	Bit-mapped value indicating foreground and background colors and attributes.
14	Frame attributes <sup>1</sup> :	Bit-mapped value indicating foreground and background colors and attributes.
15	Edge clipping:	0 = Off
		1 = On
16	Update mode <sup>2</sup> :	Bit-mapped code indicating update status and top or hidden status.
17	Interior attributes <sup>1</sup> :	Bit-mapped value indicating foreground and background colors and attributes.
18	Refresh sequence:	0 is lowest
		-1 = currently removed from the screen or never displayed
19	Interior fg color	Current normal video foreground color
<sup>1</sup> These codes are bit-mapped. That is, each code represents a single bit turned on or off. The codes for a field can be combined. <sup>2</sup> The update code is bit-mapped.		

Index	Description	Codes used
20	Interior bg color	Current normal video back-ground color
21	Interior rvfg color	Current reverse video fore-ground color
22	Interior rvbg color	Current reverse video back-ground color
23	Invert status	0 = Off
		1 = On
<div><div>1</div><div>These codes are bit-mapped. That is, each code represents a single bit turned on or off. The codes for a field can be combined.</div></div>		
<div><div>2</div><div>The update code is bit-mapped.</div></div>		

Attribute codes:



Update codes:



**Restrictions**      *numeric-array* must be a numeric array with at least 23 elements.

**See also**            [WINDOW OPEN](#), [WINDOW SELECT](#)

## Examples

This first example shows the usage of [Mode 1](#) of the statement. Line 20860 gets the window number of the current window before this routine opens a new window. This current window number is used later on to re-select that window after the routine is finished with its window.

```
20840  FIND.DATE:
20850
20860      WINDOW STATUS PRIOR.WIN%
20870
20880      WINDOW OPEN DATE.WIN%,32,19,17,1;

                FRAME SINGLE, RIGHT, COLOR YELLOW%,CYAN%;

                COLOR WHITE%,CYAN%; SELECT UPDATE ON
20890
~
21040      WINDOW SELECT PRIOR.WIN%
21050
21060      RETURN
```

This next example shows a usage of [Mode 2](#) of the statement. First, the current window number is determined with line 8020. Then, that window number is used in line 8030 to get the complete specifications of the current window.

At line 8050 the current video attribute for the window is checked to determine if it is reverse video. If so, normal video is set. Similarly, the code could have checked the other attribute possibilities, or current colors, and changed them if desired.

```
8000      DIM STAT%(23)
8010
8020      WINDOW STATUS CUR.WIN%
8030      WINDOW STATUS CUR.WIN%,STAT%
8040
8050      IF STAT%(17) AND 256 THEN PRINT CRT("RVOFF");
8060
~
```

This example shows the WINDOW STATUS statement being used to determine which windows are still open.

```
2000  EXIT:
2010
2020      DIM STATS%(23)
2030
2040      WINDOW SELECT 0
2050
2060      FOR I% = 10 TO 1 STEP -1
2070          WINDOW STATUS I%,STATS%
2080          IF STATS%(1) THEN WINDOW CLOSE I%
2090          NEXT
~
```

---

# WINDOW TAKE Statement

WINDOW TAKE moves a region of text from one window to another.

**WINDOW TAKE** *source-window, source-col, source-row, width, height,*  
*dest-window, dest-col, dest-row*

<b>Operation</b>	<p>The rectangular block of text in the <i>source-window</i> is moved from that window to the <i>dest-window</i>.</p> <p>The upper left corner of the block of text in <i>source-window</i> is identified by <i>source-col</i> and <i>source-row</i>. The block extends to the right for <i>width</i> columns and down for <i>depth</i> rows.</p> <p>The block of text is moved to and overlays <i>dest-window</i> starting at <i>dest-col</i> and <i>dest-row</i>. The area in <i>source-window</i> where the text was is blanked to spaces in the current background color attributes of the window.</p>
<b>Notes</b>	<p>To copy the block of text without removing it from the <i>source-window</i>, use the <a href="#">WINDOW COPY</a> statement.</p> <p>The <i>source-window</i> and <i>dest-window</i> may be the same window number.</p> <p>The text moved includes its color and video attributes.</p> <p>If either <i>source-window</i> or <i>dest-window</i> is the active window, and it has UPDATE ON, the screen is refreshed.</p>
<b>Restrictions</b>	<p>Both the <i>source-window</i> and the <i>dest-window</i> must refer to open windows.</p> <p>The block of text must be contained within the source window and must fit in the destination window.</p>
<b>See also</b>	<a href="#">WINDOW COPY</a> , <a href="#">WINDOW GET</a>
<b>Examples</b>	<p>An example usage of the WINDOW TAKE is shown on the following page, moving displayed text within the same window.</p>

Statements

If a window currently has:

INVOICE		INVOICE MAINTENANCE		05/17/1999	
Customer: AAAA Manufacturing 2637 West Hollywood Blvd Nirvana, CA 99000-4321			Invoice 90001  Date 05/12/1999		
DATE	DESCRIPTION	QTY	AMOUNT	TOTAL	
=====					
04/15/1999	ABC 6000 P-III 666, TOWER	1	\$1,250.00	\$1,250.00	
	8MB 70ns MEMORY	16	\$150.00	\$2,400.00	
	KEYBOARD EXTENSION CABLE	1	\$9.00	\$9.00	
	MONITOR EXTENSION CABLE	1	\$15.00	\$15.00	
	MOUSE EXTENSION CABLE	1	\$9.00	\$9.00	
	16GB MAXTOR SCSI DISK DRIVE	2	\$400.00	\$800.00	
	25" VGA MONITOR	1	\$2,200.00	\$2,200.00	
			TOTAL:	\$6,683.00	

Then the following statement:

WINDOW TAKE 1,1,11,80,8,1,1,14

produces:

INVOICE		INVOICE MAINTENANCE		05/17/1999	
Customer: AAAA Manufacturing 2637 West Hollywood Blvd Nirvana, CA 99000-4321			Invoice 90001  Date 05/12/1999		
DATE	DESCRIPTION	QTY	AMOUNT	TOTAL	
=====					
04/15/1999	ABC 6000 P-III 666, TOWER	1	\$1,250.00	\$1,250.00	
	8MB 70ns MEMORY	16	\$150.00	\$2,400.00	
	KEYBOARD EXTENSION CABLE	1	\$9.00	\$9.00	
	MONITOR EXTENSION CABLE	1	\$15.00	\$15.00	
	MOUSE EXTENSION CABLE	1	\$9.00	\$9.00	
	16GB MAXTOR SCSI DISK DRIVE	2	\$400.00	\$800.00	
	25" VGA MONITOR	1	\$2,200.00	\$2,200.00	
			TOTAL:	\$6,683.00	

As can be seen, two blank lines have been inserted, but not quite. Notice that the "totals line" at the bottom remained in the same position. If a line insert had been performed, the totals line would have scrolled off the window. Basically, the WINDOW TAKE statement moved a section of the middle of the window down two lines without affecting any other portion of the screen.

---

# WINDOW TITLE Statement

WINDOW TITLE sets or changes the displayed title of a window.

1

WINDOW TITLE *window*

2

WINDOW TITLE *window*, *title-exp* [, TOP | BOTTOM]  
[, LEFT | CENTER | RIGHT]  
[, NORMAL | REVERSE | BLINK | UNDERLINE | HALF]...  
[, COLOR *fg*, [*bg*]]

Operation	<p><b>Mode 1</b>—Specifies that the <i>window</i> has no displayed title.</p> <p><b>Mode 2</b>—Specifies that the <i>window</i> has a title and may specify the display parameters of that title.</p>
Notes	<p>TOP and BOTTOM refer to the location of the title display: either on the top frame line or the bottom frame line.</p> <p>LEFT, CENTER and RIGHT refer to the justification of the title text on its display line: left justified, center justified or right justified.</p> <p>NORMAL, REVERSE, BLINK, UNDERLINE and HALF refer to the video attributes associated with the displayed title text. Multiple attributes may be specified. When there is a conflict with attributes specified (i.e, UNDERLINE, NORMAL) the last one specified takes precedence.</p> <p>The COLOR option specifies the color of the title text. Omitting the bg parameter causes the background color to be black.</p> <p>If <i>window</i> is the active window with UPDATE ON, the new title is displayed immediately.</p> <p>Multinational and line drawing characters may be included in the title text.</p>
Defaults	<p>When <a href="#">Mode 2</a> is used (title text specified), the title text will have the following attributes, unless overridden by specific options described above: TOP, CENTER justified, with NORMAL video attributes, with the same COLOR as the default window frame colors.</p>

Statements

**Restrictions**      The *window* must refer to an open window with a frame.

The title text must fit on the frame (be no longer than the window's width).

Since window number 0 has no frame, it can have no title.

A window may have only one title at a time. Specifying a second title for a window causes the second title to replace the previous title.

The window title text cannot include video attributes (blink, underline, etc.). Use the video attribute options of the WINDOW TITLE statement to set title attributes.

**See also**            [WINDOW OPEN](#)

**Examples**          This example is similar to an earlier one used for illustrating the [WINDOW OPEN](#) statement. Here, instead of opening a new window for the help display, a previously opened window is modified by changing the title (line 800060) and selecting it. After the help text is displayed, the window is not closed, just removed from the display. It is available the next time a help display is needed.

```
800000      DEF FN.HELP$(HELP.FILE$,FX%,FY%,W%,D%)
800010
800020          HELP.WIN% = 9
800030          WINDOW STATUS PRIOR.TO.HELP%
800040          OPEN #30: HELP.FILE$, INPUT SEQUENTIAL
800050
800060          WINDOW TITLE HELP.WIN%, HELP.FILE$, TOP, CENTER,
                COLOR WHITE%,GREEN%
800070          WINDOW SELECT HELP.WIN%, UPDATE ON
~
800280          PRINT CLS$;
800290          CLOSE #30
800300          WINDOW SELECT PRIOR.TO.HELP%, UPDATE ON
800310          WINDOW REMOVE HELP.WIN%
800320          HELP.WIN% = 0
800330
800340          FNEND
```



# WRITE Statement

WRITE outputs formatted fields to a disk file.

1   **WRITE**   *#channel: write-list*

2   **WRITE**   *#channel, key: write-list*

*write-list*               »   **IOLIST** *listname*

*expression-list*       »   *expression-list*

*expression-list*       »   *expression[, expression-list]*

**Operation**       **Mode 1**—Writes the expression values in *write-list* to a sequential access disk or tape file record. The record is written at the current end of file.

**Mode 2**—Writes the expression values in *write-list* to a direct-, keyed- or indexed-access disk file. The record is written to the record position indicated by *key*.

**Notes**            This statement formats a record into fields. This formatted record can be read with the various forms of the READ statement and should not be read with the [INPUT](#) or [LINPUT](#) statement.

The formatted fields have the general form: *code data*. The code may be one of five different values, indicating the type of field:

Code	Meaning	Length
1	Integer value	3
2	Numeric value	9
4	String value < 255 chars	2+string len
5	String value > 254 chars	5+string len
0	End of record	1

Unless option MULTILOCK is in effect for the file channel, writing a record to a file releases all record locks on the file channel.

The *write-list* may be specified as **IOLIST** *listname*. The *listname* must be a previously defined **IOLIST** list name. It specifies the variables to be written

Statements

and the sequence that those variables are written. Refer to the [IOLIST](#) statement.

**Restrictions** The *channel* must refer to a file opened with OUTPUT or UPDATE. Refer to the [OPEN](#) statement for details.

When writing to a direct access file, the *key* must be a numeric expression. Writing to a keyed or indexed access file requires that the *key* be a string expression.

Since direct, keyed, and indexed files have fixed record lengths, when this statement writes to one of those files (mode 2) and the length of the record written is longer than the allocated record length, a trappable error 47 occurs: "Truncated record."

**See also** [IOLIST](#), [MAT WRITE](#), [OPEN](#), and [PRINT](#)

### Examples

*This code writes the updated information when a payables check is printed.*

```
5070      CHK.KEY$ = FORMAT$(CHK.TRAN,"#####")
```

*Two records are written, one to the check register file and another to the transaction history file.*

```
5080      WRITE #8,CHK.KEY$: MDY$,"C",NAME$,INV.DESC$,  
              FORMAT$(CHK.NBR,"#####"),INV.KEY$,AMOUNT,BALANCE  
5090  
5100      HIST.TRAN = HIST.TRAN+1  
5110      HIST.KEY$ = CODE$&FORMAT$(DAY(POST.DATES),"#####")&  
              FORMAT$(HIST.TRAN,"#####")  
5120      WRITE #7,HIST.KEY$: STR$(CHK.NBR),INV.KEY$,"C",AMOUNT  
5130
```

```
~  
6600      PUT CHECK.NUMBER:  
6610
```

*The next check number and other control values are written to a direct access file record.*


```
6620      WRITE #5,1: A,CHK.NBR,C,D,E,F,EXP.TRAN,HIST.TRAN,I,  
              CHK.TRAN  
6630  
6640      RETURN
```

---

## YESNO\$ Function



YESNO\$ displays “(Y/N)” on the console, awaits an operator’s valid response, and returns the single character response.

YESNO\$

**Operation** At the current cursor location the prompt “(Y/N)” is displayed and processing is suspended until the operator enters a “Y” or an “N”. All other responses are ignored (except  sequences). The operator’s response is returned.

**Notes** This is one of only two functions that perform output and return a value.

Control is not returned to the program until a “Y” or “N” is entered. Pressing a key that has an [ON KEY](#) trap set will not invoke the trap routine until a “Y” or “N” is entered.

,  entry causes the YESNO\$ function to return with an “N” value.

After “Y” or “N” is entered the cursor moves to the start of the next line on the console. Scrolling will occur if the prompt “(Y/N)” was displayed on the last line of the screen or window.



When [ON TIMEOUT](#) is in effect and no response is given to this function in the time period specified for input, a “N” response is returned.

**Multinational Usage.** The actual valid responses are determined by analysis of the `SYSTEM.TEOSnnn.KEYWORD` file, keywords 1 and 2. The first character of these two keywords are the only responses accepted. For example, a system set up for the Spanish language would have these two keywords defined as “Si” and “No” thus allowing ‘S’ or ‘N’ for responses; a German system might have “Ja” and “Nein” allowing ‘J’ and ‘N’ for responses.

The possible response characters will be the characters used in the prompt text. For example, English: “(Y/N)”, Spanish: “(S/N)”, German: “(J/N)”.

The character returned as the value of this function will always be either a ‘Y’ or an ‘N,’ allowing the program to be coded independently of the language used by the operator.

**I/O Redirection** This statement actually displays the prompt text on the current standard output device (stdout) and gets the input from the current standard input device (stdin). Normally this is the console display and keyboard. However, stdout and/or stdin may have been redirected when the program was invoked:

```
>myprog > text.output < text.input
```

When stdout has been redirected the prompt text is displayed on that device or written to that file or pipe. When stdin has been redirected the character accepted by this statement comes from that file, device or pipe. When stdout is redirected, the text accepted is echoed to stdout.

A program can determine if the stdin or the stdout device has been redirected to a file or device other than the console keyboard and display by using the [SYS.ENV\\$](#) function:

```
IF SYS.ENV$(34,"STDIN") = "Y"
```

The above test is true when stdin has been redirected.

**See also** [ERRMSG\\$](#)

**Examples** The following code segment is taken from the example programs and is part of the [ON KEY](#) event trap for the quit key.

Statements

```
~
The quit question 802142 WINDOW SELECT QUIT.WIN%
window is selected 802144
and the quit ques- 802145 PRINT AT$(2,2);"Do you really wish to QUIT? ";
tion is displayed.

The YESNO$ 802146 REPLY$ = YESNO$
function accepts the 802147
operator's response 802148 WINDOW SELECT PRIOR.TO.QUIT%
which can only be a 802149 WINDOW CLOSE QUIT.WIN%
'Y' or an 'N'. 802150
802151 IF REPLY$="N" THEN RESUME ELSE RESUME END.RUN
~
```

# A: Language Summary

	Type†	Description
ABS	F	Return absolute value of number
ACOS	F	Compute inverse cosine
ACOT	F	Compute inverse cotangent
ACSC	F	Compute inverse cosecant
ACTIVATE	F	Start a subtask
ADDROF	F	Return memory location of item
ANGLE	F	Compute angle in right triangle
ASC	F	Return ASCII value of character
ASEC	F	Compute inverse secant
ASIN	F	Compute inverse sine
AT\$	F	Return cursor positioning string
ATAN	F	Compute inverse tangent
ATN	F	Compute inverse tangent
AUTO	C	Begin automatic line number entry
AVAIL.WINDOWS	F	Return number of windows currently available
BIN	F	Compute value of binary string
BINOF\$	F	Return binary string of value
BOTTOM	C	Position to bottom of source program
BREAK	C	Enable debugging break point
BREAK	S	Terminate program structure
CALL	S	Call C language routine
CALL.RETURN.VALUE	F	Value set by last called routine
CASE	S	Define test for SELECT-CEND program structure
CEIL	F	Return ceiling of value
CEND	S	End of SELECT-CEND program structure
CHAIN	S	Continue execution in another program
CHANGE	C	Modify text of source program
CHR\$	F	Return character of ASCII value
CLEAR	S	Initialize variables
CLOSE	S	Close a file or device channel
† C = Command, F = Function, S = Statement		

Type <sup>†</sup>	Description
CLSS\$	F Return screen clear or page eject string
CMDARG\$	F Return command line argument
COLOR	S Set display colors
COMMON	S Define variables common to multiple programs
COMPILE	C Compile current program
CONTINUE	C Resume execution after a STOP, break point, or <input type="button" value="Break"/> , <input type="button" value="C"/>
CONTINUE	S Branch to loop repeat statement
COPY	C Duplicate one or more lines of the program
COS	F Compute the cosine
COSH	F Compute the hyperbolic cosine
COT	F Compute the cotangent
COTH	F Compute the hyperbolic cotangent
CRT\$	F Return screen/printer attribute string
CSC	F Compute the cosecant
CSCH	F Compute the hyperbolic cosecant
CSI	S Perform a system command and return
CSI.RETURN.CODE	F Value set by last CSI or SYSTEM statement command executed.
DATA	S Define literals used by read statements
DATE\$	F Return date string
DAY	F Compute date value from date string
DECLARE CALL	S Define name and formal arguments to a C language function
DEF FN	S Begin user-defined function definition
DEG	F Compute degree equivalent of radian value
DEL\$	F Remove subfield from string
DELETE	C Delete lines of the source program
DELETE	S Delete record from file
DIM	S Define arrays
DOWN	C Position to next line of source program
DTE\$	F Validate date string
ELSE	S Begin false block of IF-IFEND statement structure
END	S End of program source
END SUB	S End of subprogram definition
† C = Command, F = Function, S = Statement	

	Type <sup>†</sup>	Description
EOF	F	Return end-of-file or record-not-found status for a file
EPS	F	Return smallest value maintained by system
ERL	F	Return error line number
ERR	F	Return error number
ERRMSG\$	F	Display message at bottom, wait for response
EVENT	F	Return status of event semaphore
EXP	F	Compute $e$ raised to a power
EXT\$	F	Return subfield of string
FILL	S	Draw and fill an irregular shape
FILL BAR	S	Draw and fill a rectangular shape
FILL CIRCLE	S	Draw and fill a circle
FILL PIE	S	Draw and fill a circular wedge
FIX	F	Return integer value of floating point number
FLOAT	F	Return floating point value of integer number
FLOOR	F	Return floor of value
FNEND	S	End multiple line function definition
FOR	S	Begin FOR-NEXT program structure
FORMAT\$	F	Convert and format value as string
FP	F	Return the fractional portion of number
GCD	F	Return greatest common denominator
GET	S	Return character from device or file
GET COMMON	S	Retrieve common data from parent task
GOSUB	S	Jump to a subroutine
GOTO	S	Jump to another location in program
HELP	C	Display help text for commands, statements, and functions
HEX	F	Return value of hexadecimal string
HEXOF\$	F	Convert value to hexadecimal string
IF	S	Single line conditional statement or start of IF-IFEND programming structure
IFEND	S	End of IF-IFEND programming structure
INCLUDE	S	Include another source program
INDENT	C	Make program source indentation uniform
INF	F	Return largest value maintained by system
INP	F	Return control character used to terminate last input
† C = Command, F = Function, S = Statement		

Type <sup>†</sup>	Description
INPUT	S Accept line or record from console or file
INS\$	F Insert a subfield into a string
INT	F Return largest whole number that is $\leq$ value
IOLIST	S Define record layout for reads and writes
IP	F Return the whole number portion of value
KILL	S Terminate a subtask
LCASE\$	F Return the lowercase equivalent of string
LEFT\$	F Return the leftmost portion of a string
LEN	F Compute length of string
LENGTH	C Display size parameters of program
LET	S Assign value to variable
LINE	F Return line length of console or printer
LINK	S Continue execution in another program
LINPUT	S Accept line or record from console or file
LINPUT USING	S Accept line or record from console
LIST	C Display source program
LOAD	C Load a program
LOADU	C Load an unnumbered program
LOCAL	S Define variables local to subprogram or user-defined function
LOCATE	C Find text in program
LOCKED.BY	F Return partition number of user locking file or record
LOG	F Compute natural logarithm
LOG2	F Compute binary logarithm
LOG10	F Compute common logarithm
LPAD\$	F Add spaces to start of string to right justify
LPLIST	C Display source program on printer
LPXREF	C Display cross-reference listing on printer
LRL	F Rotate bits of an integer left
LRR	F Rotate bits of an integer right
LSL	F Shift bits of an integer left
LSR	F Shift bits of an integer right
LTRIM\$	F Remove leading spaces from a string
MAT	S Initialize an array
MAT INPUT	S Read ASCII data into an array from console or file
<sup>†</sup> C = Command, F = Function, S = Statement	



	Type <sup>†</sup>	Description
<a href="#">MAT PRINT</a>	S	Write ASCII data from an array to console or file
<a href="#">MAT READ</a>	S	Read formatted data into an array from DATA or a file
<a href="#">MAT READNEXT</a>	S	Read formatted data into an array from a direct, keyed, or indexed record
<a href="#">MAT READPREV</a>	S	Read formatted data into an array from an indexed record
<a href="#">MAT SORT</a>	S	Sort an array's indices
<a href="#">MAT WRITE</a>	S	Write formatted data from an array to a file
<a href="#">MATCH</a>	F	Compare string to mask
<a href="#">MAX</a>	F	Return the larger of two values
<a href="#">MERGE</a>	C	Merge lines from another source program
<a href="#">MERGEU</a>	C	Merge lines from another unnumbered source program
<a href="#">MID\$</a>	F	Return a middle portion of a string
<a href="#">MIN</a>	F	Return the smaller of two values
<a href="#">MOD</a>	F	Return the remainder of one value divided by another
<a href="#">MODIFY</a>	C	Edit existing lines of the source program
<a href="#">MOUNT</a>	S	Allow change of diskette
<a href="#">MOUSE.BUTTON</a>	F	Return mouse button code
<a href="#">MOUSE.COL</a>	F	Return mouse cursor column number
<a href="#">MOUSE.FRAME</a>	F	Return mouse cursor window frame location
<a href="#">MOUSE.ROW</a>	F	Return mouse cursor row number
<a href="#">MOUSE.WINDOW</a>	F	Return mouse cursor window location
<a href="#">MOVE</a>	C	Move source lines to another location in program
<a href="#">MSEC</a>	F	Return number of "ticks" since last system boot
<a href="#">NAME</a>	C	Change the name of the source program
<a href="#">NBR</a>	F	Test if string contains a valid number
<a href="#">NEW</a>	C	Start a new source program
<a href="#">NEXT</a>	S	End of <a href="#">FOR-NEXT</a> program structure
<a href="#">NEXT.FILE</a>	F	Returns next available file channel
<a href="#">OCT</a>	F	Compute value of octal string
<a href="#">OCTOF\$</a>	F	Return octal string of value
<a href="#">OLDMOD</a>	C	Edit existing lines of source program
<a href="#">ON CENTER</a>	S	Define center mouse button event trap
† C = Command, F = Function, S = Statement		

	Type <sup>†</sup>	Description
ON ERROR	S	Enable/disable error trapping
ON EVENT	S	Enable/disable event trapping
ON GOSUB	S	Dispatch using list of subroutines
ON GOTO	S	Dispatch using list of line references
ON KEY	S	Enable/disable key trapping
ON.KEY.TOKEN	F	Return value of trapped key
ON LEFT	S	Define left mouse button event trap
ON MOUSE	S	Define mouse event trap
ON RIGHT	S	Define right mouse button event trap
ON TIMEOUT	S	Define input time-out event trap
OPEN	S	Open a file or device for access
OPTION	S	Set program compilation and execution options
ORD	F	Return ASCII value of character
OTHERWISE	S	Define default for SELECT-CEND program structure
OVR\$	F	Overlay one string onto another
PAGE	F	Return screen, window, or printer depth
PI	F	Return the value of $\pi$
PLOT	S	Draw an unfilled, irregular shape
PLOT ARC	S	Draw an arc of a circle
PLOT BAR	S	Draw a rectangle
PLOT CIRCLE	S	Draw a circle
PLOT PIE	S	Draw an unfilled wedge of a circle
POS	F	Return display column position for screen or printer
PRINT	S	Write ASCII data to screen or file
PRINT USING	S	Write ASCII formatted data to screen or file
PUT	S	Write a character to screen or file
PUT COMMON	S	Transfer common data to parent task
QUIT	C	Exit from the interpreter
QUIT	S	Exit from the program
RAD	F	Compute radian equivalent of degrees
RANDOMIZE	S	Make RND function random
READ	S	Read formatted data from DATA or a file
READNEXT	S	Read formatted data from a direct, keyed, or indexed record
READPREV	S	Read formatted data from an indexed record
† C = Command, F = Function, S = Statement		

	Type <sup>†</sup>	Description
RECALL	C	Repeat last command
REM	S	Remark or comment statement
RENUMBER	C	Renumber all or portion of source program
REP\$	F	Replace a subfield in a string
RESET	F	Turn off a semaphore
RESTORE	S	Set location of next <b>DATA</b> item used
RESUME	S	Exit from error, event, or key trapping routine
RETURN	S	Exit from a subroutine
RIGHT\$	F	Return rightmost portion of a string
RND	F	Return next pseudo-random number
ROUND	F	Round a value
RPAD\$	F	Add spaces to end of string to left justify
RPT\$	F	Repeat a character or string
RTRIM\$	F	Remove trailing spaces from string
RUN	C	Begin execution of a program
RUN	S	Start execution with another program
SAVE	C	Save source program in current format
SAVEA	C	Save source program as text file
SAVEC	C	Save source program as program file
SAVEU	C	Save source program as unnumbered text file
SCH	F	Search a string for a substring
SEC	F	Compute secant
SECH	F	Compute hyperbolic secant
SECOND	F	Compute number of seconds in time string
SELECT	S	Begin SELECT-CEND program structure
SEMAPHORE	F	Catalogue a semaphore name
SET	F	Set a semaphore and return prior status
SET FILL	S	Define graphics fill style and color
SET LINE	S	Define graphics line style and color
SET MARKER	S	Define graphics marker style and color
SET TEXT	S	Define graphics text attributes
SGN	F	Return sign of value
SHARED	S	Define variables in subprogram or user-defined function that are global to main program
SHOWSTACK	C	Display the return location of current routine
† C = Command, F = Function, S = Statement		

Type <sup>†</sup>	Description
SIN	F Compute sine
SINH	F Compute hyperbolic sine
SLEEP	S Suspend execution for a period of time
SPACE\$	F Return string of spaces
SQR	F Compute square root of value
SSTEP	C Single step execution of program, even through sub-programs and user-defined functions
STATIC	S Define variables local to subprogram or user-defined function that are static in duration
STEP	C Single step execution of program
STOP	S Interrupt execution of program
STR\$	F Convert number to a string
STRTIME\$	F Format date and time into string
SUB	S Begin subprogram definition.
SWAP	S Exchange two variable values
SYS.ENV\$	F Return system environment or user parameters
SYSTEM	S Execute system command and return
TAB	F Position output to next tab stop
TAN	F Compute tangent
TANH	F Compute hyperbolic tangent
TEXT	S Display graphics text
THEN	S True statements of IF-IFEND program structure
TIME\$	F Return time string
TIMER	S Set an event timer
TOP	C Position to top of source program
TOTAL.WINDOWS	F Returns the total number of windows supported.
TRACE	C Enable debugging trace modes
TRIM\$	F Remove leading, trailing, and multiple embedded spaces
UCASE\$	F Return uppercase equivalent of string
UNBREAK	C Disable debugging break point
UNGET	S Place character into input stream of console or file
UNLOCK	S Release record lock
UNTRACE	C Disable debugging trace mode
UP	C Position up one line in source program
† C = Command, F = Function, S = Statement	

	Type <sup>†</sup>	Description
USED.WINDOWS	F	Returns the number of currently open windows
VAL	F	Convert string to number
VARS	C	Display variable names and values
VDI	S	Generic graphics drawing
VIEW	C	Specify information displayed by BREAK, LIST, SSTEP, STEP, VARS, XREF
WAIT	S	Perform a page-wait or wait for input device ready
WAIT EVENT	S	Wait for an event to occur
WEND	S	Ending statement of WHILE-WEND structure
WHILE	S	Begin WHILE-WEND program structure
WINDOW CHOICE	S	Use a choice window to choose one item from many
WINDOW CLEAR	S	Clear a window to spaces or a specific character
WINDOW CLIP	S	Set clipping attribute for window
WINDOW CLOSE	S	Close and remove a screen window
WINDOW COPY	S	Copy a region of one window to another
WINDOW EDIT	S	Use a window to display and edit array of strings
WINDOW FRAME	S	Set frame attribute for window
WINDOW GET	S	Copy text from display window to variable
WINDOW GET TITLE	S	Copy text from window title to variable
WINDOW INVERT	S	Set invert style for window
WINDOW LOCATE	S	Return cursor location in a window
WINDOW MOVE	S	Change displayed position of a window
WINDOW OPEN	S	Open a new window
WINDOW REFRESH	S	Display a window
WINDOW REMOVE	S	Remove display of window
WINDOW RESTORE	S	Retrieve window from disk file
WINDOW SAVE	S	Save window to disk file
WINDOW SELECT	S	Select the active window
WINDOW STATUS	S	Return status of window
WINDOW TAKE	S	Move text region from one window to another
WINDOW TITLE	S	Define title of window
WRITE	S	Write formatted record to file
XREF	C	Display program cross-reference listing on screen
YESNO\$	F	Prompt for yes/no, return reply
† C = Command, F = Function, S = Statement		

# B: THEOS Character Set

The following characters, symbols, and attributes are displayed by the THEOS operating system when corresponding character code is output to the console or printer.

## Commands

	0	1	2	3	4	5	6	7	8	9
0		HOME	FON	FOFF	PON	POFF	RIGHT	BELL	BS	TAB
10	LF	ULON	CLEAR	CR	RVON	RVOFF		IL	DL	IC
20	DC	KOFF	ULOFF	EOL	EOS	KON	UP	ESC	EU	BON
30	BOFF		SP	!	"	#	\$	%	&	'
40	(	)	*	+	,	-	.	/	0	1
50	2	3	4	5	6	7	8	9	:	;
60	<	=	>	?	@	A	B	C	D	E
70	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y
90	Z	[	\	]	^	_	`	a	b	c
100	d	e	f	g	h	i	j	k	l	m
110	n	o	p	q	r	s	t	u	v	w
120	x	y	z	{		}	~	DEL	MON	MOFF
130	SON	SOFF	GON	GOFF						
140										
150										
160	┐	┌	└	┘	┙	┚	┛	├	┤	┥
170		┐	└	┘	┙	┚	┛	├	┤	┥
180	┐	┌	└	┘	┙	┚	┛	├	┤	┥
190			Ä	ä	å	à	á	È	è	ê
200	è	é	ĩ	î	ì	í	Ö	ö	ô	ò
210	ó	Ü	ü	û	ù	ú	Ç	ç	Ñ	ñ
220	Æ	æ	Å	å	ß	¿	¡	¢	£	¥
230	Pt	€	¼	½	ÿ	§	■	²		
240										
250										

# C: Multinational Characters

Multinational characters are those characters and symbols that are outside of, or in addition to, the characters and symbols of your native language, as well as your terminal keyboard and printer. These characters and symbols are used to type foreign words or special symbols that cannot be entered from your keyboard by pressing a single key.

Multinational characters can be entered from terminals whose keyboard has a functioning **[Alt]** key or some other means of transmitting characters that do not have their own key-caps.

In the following table, multinational symbols are shown along with the methods of specifying them. The characters in the column labeled Alt must be entered by pressing and holding **[Alt]** while pressing the other characters. The **[Alt]** key is released after the second character is key. For example, é is typed by pressing and holding **[Alt]**, pressing **[e]**, then releasing **[Alt]**.

The third column, labeled #, shows the numeric value of the symbol which is an alternate method of entering the multinational character. For example, é can also be entered by pressing and holding **[Alt]**, pressing the numeric keypad digits **[2]****[0]****[1]**, and then releasing **[Alt]**.

Sym	Alt	#	Sym	Alt	#	Sym	Alt	#	Sym	Alt	#	Sym	Alt	#
Ä	A"	192	É	E'	197	Ö	O"	206	ÿ	y"	234	§	so	235
À	Ao	222	ë	e"	198	ö	o"	207	¿	??	225	þ	oo	236
Æ	AE	220	ê	e^	199	ô	o^	208	¡	!!	226	2		237
ä	a"	193	è	e`	200	ò	o`	209	¢	c/	227			
â	a^	194	é	e'	201	ó	o'	210	£	l-	228			
à	a`	195	ï	i"	202	ß	ss	224	¥	Y-	229			
á	a'	196	î	i^	203	Ü	U"	211	€	C=	231			
â	ao	223	í	i'	204	ü	u"	212	₧	pt	230			
æ	ae	221	í	i'	205	û	u^	213	½	12 †	233			
Ç	C,	216	Ñ	N~	218	ù	u`	214	¼	14 †	232			
ç	c,	217	ñ	n~	219	ú	u'	215						

‡ Symbol for the Spanish peseta.

† Do not use the numeric keypad digits for these entries.

# D: Error Codes and Messages

#	Message	Comment
1	Program cancel, Break-C detected.	<p>In compiled programs, only reported when ON ERROR enabled. Otherwise, <code>Break</code>, <code>C</code> is ignored.</p> <p>In interpreted programs, <code>Break</code>, <code>C</code> displays "Break at line nnnnnn" and reverts to the interpreter's command mode.</p>
27	File is not open.	Occurs on any file access statement.
28	File is still open.	Occurs on OPEN.
30	File "AAAA" not found.	Occurs on OPEN when the specified file cannot be found with the current path and search sequence.
31	Disk is full.	May occur when opening a sequential file for output or when writing to a direct, indexed, or keyed file that needs to be extended.
32	Directory is full.	Indicates an attempt to OPEN a new OUTPUT SEQUENTIAL file to a full main directory or a full library.
33	File "AAAA" is protected.	Attempt to OPEN a file for INPUT that is read protected, or for OUTPUT or UPDATE that is write protected.
34	Invalid key.	A numeric key used on a keyed or indexed access file; a string key used on a direct access file.
35	Invalid access mode.	<p>Occurs on INPUTs, PRINTs, READs, or WRITEs. Indicates a key was used when the file is SEQUENTIAL, or that a key was not used when the file is DIRECT, KEYED, or INDEXED.</p> <p>This message is also displayed when a file is opened with a different access method than the method specified when the file was created. (For example, an attempt to open an indexed file with direct access.)</p>



#	Message	Comment
36	Out of data.	Occurs on READ statement when the pool of DATA items is exhausted. Either add more data items, reduce the number of items read, or use the RESTORE statement to reuse existing DATA items.
37	Graphics not available.	Occurs on any VDI statement when the device is not attached as a VDI graphics device.
38	Invalid file channel ... out of range.	Occurs on OPEN or CLOSE statements when the file channel requested is outside the range of 1—999.
40	Command not found.	Occurs on CHAIN, LINK, or RUN.
46	Device not attached.	Occurs on an attempt to OPEN a file on a drive or device that is not attached to your partition.
47	Truncated record.	Occurs on PRINTs or WRITES to DIRECT, KEYED, or INDEXED files, when the data to be written is longer than the allocated record length.
48	Record is locked.	Records are locked automatically when a program reads a record from a file that was OPENed for UPDATE.  This error occurs only if OPTION LOCK has been specified, and only on reads when another user or task has read, but not released, the record that this program requests.
49	File is locked.	Files are locked only when the option LOCK is specified on the OPEN.  This error occurs only if OPTION LOCK has been specified, and only on OPENS when another user or task has the same file open with option LOCK.
50	Divide by zero error at nnnnnn.	Only reported when ON ERROR enabled. Otherwise, result set to INF (floating point value) or -32768 (integer value).
51	Integer overflow error at nnnnnn.	Only reported when ON ERROR enabled. Otherwise, result set to -32768.

#	Message	Comment
52	Window is not opened.	Occurs on window statements when the indicated window number has not been opened.
53	This console does not support windowing.	Windows can only be used on the main console, THEO+GRAFX console, or on a terminal with the THEOS Window Manager text windowing product.
54	Insufficient memory for WINDOW statement.	Occurs on WINDOW OPEN or WINDOW CHOICE when there is insufficient memory to store image of new window.
55	Invalid window size specified.	The window, with its frame and shadow, must physically fit on the screen.
56	Advanced Windowing feature not supported on this console.	Windows and ON KEY trapping are not available on the current console.
57	Invalid window number.	Window numbers are in the range 0—TOTAL.WINDOWS.
58	Window statement error at line nnnnnn.	Only reported in interpreter. This message is reported in addition to specific window error.
59	Invalid ON KEY token.	Only alphabetic keys, function keys, and editing keys may be trapped with the ON KEY statement.
60	Invalid window position.	The window origin specified does not allow the window, with any frame and shadow, to fit on the screen.
61	ON KEY not supported on this console.	ON KEY trapping can only be used on the main console, THEO+GRAFX console, or on a terminal with the THEOS Window Manager text windowing product.
62	PUT COMMON failed because main task has no more memory.	Occurs only when subtask uses PUT COMMON statement and one of the common items is a string variable, and the main task's data space is full and cannot be expanded.
64	Invalid WINDOW COPY position.	The region specified with the WINDOW COPY or WINDOW TAKE statement is not wholly contained within a window.
66	Array too small for count in WINDOW CHOICE statement.	The <i>count</i> value is larger than the dimensioned size of one of the choice window arrays.

#	Message	Comment
67	Invalid WINDOW COPY parameters.	
68	COMMON variable "AAA" not found in main task.	Occurs on PUT and GET COMMON statement when a reference is made to a variable not declared as COMMON in main task.
70	Invalid operation on window number 0.	Window 0 is special and cannot be resized or moved and cannot have a frame or shadow.
71	Invalid parameters for WINDOW FRAME or TITLE statement.	
78	The specified window has no title.	Occurs only with the WINDOW GET TITLE statement.
80	The specified window has no frame defined.	The title is displayed in the frame area of a window.
86	Multitasking not supported in MultiUser BASIC Interpreter.	Occurs with ACTIVATE statement when executed in the interpreter.
120	Array too small for count in WINDOW EDIT statement.	
124	Unassigned IOLIST variable at line "nnnnnn".	Occurs on i/o statements using the IOLIST phrase when the indicated IOLIST name has never been assigned a list of variable names.
125	ON MOUSE not supported on this console.	The mouse can only be used on the main console, THEO+GRAFX console, or on a terminal with the THEOS Window Manager text windowing product.
129	String too long for old style 'C' call.	Only reported when ON ERROR enabled. Otherwise, the string is truncated to 255 characters.

Not trappable	Array dimension out of range, must be from 1 to 32767	Interpreter message only.
Not trappable	Array variable "AAAA" used as a scalar at line "nnnnnn".	The variable is currently defined as an array.
Not trappable	Attempt to change dimensions of "AAAA" at line "nnnnnn".	Arrays cannot be redimensioned except between CHAINED or LINKed programs.
Not trappable	BREAK without DEF, FOR, SELECT, or WHILE at line "nnnnnn".	The BREAK statement may only be used within the indicated programming structures.

#	Message	Comment
Not trappable	CALL argument does not match DECLARE parameter at line "nnnnnn".	New-style C function CALLs must match the DECLARE statement for the function name.
Not trappable	CALL argument does not match SUB parameter at line "nnnnnn".	CALLs to a subprogram must match the SUB statement definition for the subprogram.
Not trappable	Call function "AAAA" at line nnnnnn not found in current OPTION CALL module.	Interpreter message only.
Not trappable	Call statement at line nnnnnn with no OPTION CALL module loaded.	Interpreter message only.
Not trappable	CASE with no active SELECT at line "nnnnnn".	The CASE statement may only be used inside of a SELECT-CEND structure.
Not trappable	CEND with no active SELECT at line "nnnnnn".	The CEND statement may only be used inside of a SELECT-CEND structure. There must be only one CEND statement for each SELECT statement.
Not trappable	CONTINUE without FOR or WHILE at line "nnnnnn".	The CONTINUE statement may only be used within the indicated programming structures.
Not trappable	DEF found inside SUB program at line "nnnnnn".	Subprogram definitions may not include user-defined function definitions.
Not trappable	Double-dimensioned array "AAAA" used as a single-dimensioned array at line "nnnnnn".	
Not trappable	Duplicate argument names encountered at line "nnnnnn".	
Not trappable	Duplicate DECLARE "AAAA" at line "nnnnnn".	A C function name may be declared only once in a program. If declared multiple times, each declaration must match exactly as to number and organization of arguments.
Not trappable	Duplicate SUB program "AAAA" definition at line "nnnnnn".	A subprogram name may be defined only once.
Not trappable	Duplicate user-defined function "AAAA" definition at line "nnnnnn".	A user-defined function name may be defined only once.

#	Message	Comment
Not trappable	END SUB statement can not be conditional at line "nnnnnn".	Although the END SUB statement causes a return from the subprogram it also marks the physical end of the subprogram definition and cannot be conditionally executed.
Not trappable	END SUB without SUB at line "nnnnnn".	The END SUB statement may only be used as the end of a subprogram definition. There can only be one END SUB statement for each SUB statement.
Not trappable	Expecting xxxx item on the stack, found yyyy item.	Generally results when RESUME used instead of RETURN and vice versa.
Not trappable	FNEND statement can not be conditional at line "nnnnnn".	Although the FNEND statement causes a return from a user-defined function it also marks the physical end of the function definition and cannot be conditionally executed.
Not trappable	INCLUDE file name "AAAA" is not valid at line "nnnnnn".	The name of the include file contained invalid file name characters.
Not trappable	Incomplete "..." structure in single line IF at line "nnnnnn".	The object of a single-line IF statement must be wholly contained in the single line. Multiple-line FOR-NEXT, WHILE-WEND, SELECT-CEND, and IF-IFEND structures cannot be the object of a single line IF statement.
Not trappable	Interpreter stack overflow.	Interpreter message only.
Not trappable	Invalid argument in 'C' routine CALL at line "nnnnnn".	
Not trappable	IOLIST variable required.	IOLIST variable names do not use \$ or % characters.
Not trappable	MERGED line numbers can not exceed 999999	Interpreter message only.
Not trappable	MultiUser BASIC compiled program stack overflow.	Recompile program with larger stack size.
Not trappable	Nested SUB program found at line "nnnnnn".	Subprogram definitions may not include other subprogram definitions.
Not trappable	NEXT with no active FOR at line "nnnnnn".	The NEXT statement may only be used at the end of a FOR-NEXT structure. There must be only one NEXT statement for each FOR statement.
Not trappable	No program name to compile.	Occurs on COMPILE command only when no name has been assigned to the program.

#	Message	Comment
Not trappable	No program to compile.	Occurs on COMPILE command when no program is loaded.
Not trappable	No program to save.	Occurs on SAVE, SAVEA, or SAVEU when no program is loaded.
Not trappable	ON KEY error at line nnnnnnn.	
Not trappable	ON MOUSE error at line "nnnnnnn".	
Not trappable	OTHERWISE with no active SELECT at line "nnnnnnn".	The CASE statement may only be used inside of a SELECT-CEND structure.
Not trappable	Scalar variable "AAAA" used as an array at line "nnnnnnn".	A variable name may be a scalar or an array, but not both.
Not trappable	Shared variable "AAAA" is masked by a local variable at line "nnnnnnn".	A subprogram or user-defined function has declared a variable name as both SHARED and LOCAL or STATIC.
Not trappable	Single-dimensioned array "AAAA" used as a double-dimensioned array at line "nnnnnnn".	Only certain statements may <i>temporarily redimension</i> an array.
Not trappable	Statement cannot be executed in immediate mode.	
Not trappable	String too large for old style CALL.	A string longer than 255 characters was passed to an old-style C function.
Not trappable	SUB found inside user-defined function at line "nnnnnnn".	User-defined function definitions may not include subprogram definitions.
Not trappable	SUB without END SUB at line "nnnnnnn".	Every SUB statement needs one and only one END SUB statement.
Not trappable	Subscript range error at line nnnnnnn.	Occurs on any reference to a dimensioned variable when the subscript is less than the base or greater than the number of elements dimensioned.
Not trappable	Unable to CONTINUE execution.	
Not trappable	Variable "AAAA" has been redefined at line "nnnnnnn".	
Not trappable	Variable "AAAA" is not declared as COMMON at line "nnnnnnn".	Compiler message only. Occurs on GET COMMON or PUT COMMON statements.
Not trappable	WARNING: Shared variable "AAAA" is masked by a local variable at line "nnnnnnn".	A subprogram or user-defined function has declared a variable name as both SHARED and LOCAL or STATIC.

#	Message	Comment
Not trappable	WARNING: The above line has a line number, this number was ignored.	An unnumbered source program contained a line with a line number. The line is kept but the line number is ignored.
Not trappable	WARNING: The above line was not numbered and was ignored.	A numbered source program contained an unnumber line. The entire program line is ignored.
Not trappable	WARNING: Variable "AAAA" declared as SHARED instead of LOCAL/STATIC in MAIN at line "nnnnnn".	The main section of a program cannot have LOCAL or STATIC variables. The declaration for the variable is changed to SHARED.
Not trappable	WARNING: Variable "AAAA" declared at line "nnnnnn" after first reference.	LOCAL, SHARED and STATIC variables must be declaration prior to their references. The indicated variable keeps its default declaration.

# E: MultiUser BASIC Files

---

The following files are copied to the system as part of the THEOS MultiUser BASIC installation.

File	Description
SYSTEM.B3220LIB.*	The library of functions and routines used during compilation phase.
SYSTEM.CMD $nnn$ .B3220ASM	Compiler, assembler phase
SYSTEM.CMD $nnn$ .B3220COM	Compiler, compilation phase
SYSTEM.CMD $nnn$ .B3220INT	Interpreter
SYSTEM.CMD $nnn$ .B3220LNK	Compiler, linker phase
SYSTEM.CMD $nnn$ .B3220OPT	Compiler code optimizer
SYSTEM.CMD $nnn$ .B3220	MultiUser BASIC command
SYSTEM.HELP $nnn$ .B3220	The B3220 command CSI level help text
SYSTEM.MENU $nnn$ .B3220	Execution-time error message text.
SYSTEM.TEOS $nnn$ .B3220MTH	Compiled program, math procedures
SYSTEM.TEOS $nnn$ .B3220PLT	Compiled program, VDI graphics
SYSTEM.TEOS $nnn$ .B3220STD	Compiled program, string manipulation procedures
SYSTEM.TEOS $nnn$ .B3220WM	Compiled program, windowing procedures



# F: Tables

## ■ Program Control Statement Interactions

Statements that cause another program to execute frequently have additional affects on the current program. The following table lists the major side affects for each of these statements.

Actions	Statements						
	CHAIN	LINK	RUN	CSI	SYSTEM	QUIT	END
Close all files	✓		✓	✓		✓	✓
Clear all non-COMMON data	✓	✓	✓			✓	✓
Clear all COMMON data			✓			✓	✓
Terminate open program structures	✓	✓	✓			✓	✓
Clear the current ON ERROR trap	✓	✓	✓			✓	✓
Clear ON KEY, ON EVENT and ON MOUSE	✓	✓	✓			✓	✓
Begin execution of another program	✓	✓	✓	✓	✓	✓	
Return to calling program				✓	✓		
Reset all OPTIONS						✓	
Close windows							✓
Deactivate subordinate subtasks				✓	✓	✓	✓

## ■ Color Codes

The color of text and text background can be changed with the COLOR statement and the various WINDOW statements. The normal colors available on a display are specified with the following codes.

Color	Code	Color	Code
Black	0	Red	4
Blue	1	Magenta	5
Green	2	Yellow	6
Cyan	3	White	7

## ■ VDI Color Codes

The color of lines, text, and objects drawn with the VDI statements can be specified with the following standard color codes.

**Monochrome Displays**

Color	Code
Black	0
White	1

**Color Displays**

Color	Code	Color	Code
Black	0	Red	4
Blue	1	Magenta	5
Green	2	Yellow	6
Cyan	3	White	7

**VGA Color Displays**

Color	Code	Color	Code	Color	Code	Color	Code
Black	0	Red	4	Int Grey	8	Int Red	12
Blue	1	Magenta	5	Int Blue	9	Int Magenta	13
Green	2	Yellow	6	Int Green	10	Int Yellow	14
Cyan	3	White	7	Int Cyan	11	Int White	15

## ■ VDI Fill Patterns

The following are the minimum fill patterns available on THEOS supported VDI graphics devices.

Style	Pattern	Sample
0	Hollow	
1	Solid	
2	Vertical lines	
3	Horizontal lines	
4	45° diagonals	
5	135° diagonals	
6	Cross hatch	
7	Diagonal cross hatch	

## ■ VDI Line Styles

The following are the minimum line styles available on THEOS supported VDI graphics devices.

Code	Line style	Sample
1	Solid	—————
2	Short dashes	- - - - -
3	Dotted	. . . . .
4	Dash, dot	- . - . - .
5	Long dashes	- - - - -
6	Dash, dot, dot	- . . - . .
7	Small dots	. . . . .

## ■ VDI Marker Styles

The following are the minimum marker styles available on THEOS supported VDI graphics devices.

Code	Marker	Sample
1	Dot	•
2	Plus sign	+
3	Asterisk	*
4	Circle	o
5	X	×





















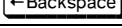








## ■ MATCH Function Pattern Specifications

The MATCH function uses the following characters as wild card specifications in the pattern or mask field.




Character	Meaning
?	Any single character matches
@	Any single letter or space matches
#	Any single digit matches (not decimal points or commas)
*?	Any number (0—255) of any characters match
*@	Any number (0—255) of letters or space characters match
*#	Any number (0—255) of digits match
%	A special <i>escape</i> character that allows specification of ? @ # and % characters as literals. This character is followed by the special character that you want treated normally.

## ■ ON KEY Terminal Key Codes

The ON KEY statement and the ON.KEY.TOKEN function use the following codes for terminal key specifications.

Key	Code	Key	Code	Key	Code	Key	Code
	256		261		269		277
	257		262		270		278
	258		263		271		279
	259		264		272		280
	260		265		273		281
			266		274		282
			267		275		283
			268		276		284

The above key codes can be modified by adding one or more of the following values to specify a key combination.

Key Modifier	Code	Hex Value
	4096	1000H
	8192	2000H
	16384	4000H

## ■ **ANSI Forms Control Characters**

A sequential file OPEN for OUTPUT and with option FORMAT, uses the following characters as the first character of each line PRINTed.

Character	Meaning
1	Start a new page.
+	Do not advance—overprint the previous line. This can only be done if the printer does not perform an automatic line advance with each carriage return. (Option ALF not specified on printer attachment.)
0	Advance two lines (skip one blank line)
-	Advance three lines (skip two blank lines)
	All other characters are not printed and cause one line to advance. By convention, the space character is used for this purpose.

## ■ **Formatted Record Field Codes**

Variables written to a data file with the WRITE or MAT WRITE statements are written with a code at the beginning of each field. The following codes are used to identify the type of field in the record.

Code	Meaning	Length
1	Integer value	3
2	Numeric value	9
4	String value, len<256 chars	2+contents
5	String value, len>255 chars	5+contents
0	End of record	1



# Index

---

-  
See  
prompt, command 24

---

## Symbols

(Alt) 639

---

## A

ALT 639

arguments

- command line 103
- to functions 38, 49
- to subprograms 38

array variable 34

arrays

- COMMON statement 136
- creating 35
- cross-reference 135
- DIM statement 136
- single dimension 34
- two dimension 34
- viewing 125, 126

AUTO command 59, 61

automatic variables 38

---

## B

binary expressions 45, 47

Boolean values 44

BOTTOM command 61

BREAK command 62, 63, 64, 65

---

## C

C language functions

- cross-reference 134

CHANGE command 66, 67

character

- multinational 639

command

- AUTO 59, 61

- BOTTOM 61

- BREAK 62, 63, 64, 65

CHANGE 66, 67

COMPILE 68

CONTINUE 69, 70

COPY 70, 71, 72

DELETE 72, 73

DOWN 74

HELP 75

INDENT 76, 77, 78

LENGTH 79

LIST 80, 81

LOAD 82

LOCATE 83

LPLIST 84, 85

LPXREF 86

MERGE 87, 88

MERGEU 88

MODIFY 89

MOVE 91

NAME 92

NEW 93

OLDMOD 94, 95

QUIT 97, 98

RECALL 99, 100

RENUMBER 101, 102

RUN 103, 104

SAVE 105, 107

SAVEU 110

STEP 111, 112

TOP 83, 113

TRACE 114, 115, 117

UNBREAK 118, 119

UNTRACE 120

UP 121

VARS 122, 123

VIEW 124, 125, 126, 130

XREF 131, 133, 134, 135, 137

command line

- arguments 103, 104

common variables 38

COMPILE command 68

compiling

- multiple programs 23

- programs 23, 68

constant names 31

constants 25, 28  
     cross-reference 137  
     numeric 28  
     string 29, 30  
     viewing 125, 126  
 CONTINUE command 69, 70  
 COPY command 70, 71, 72  
 cross-reference listing  
     See  
         XREF and LPXREF commands 131

---

## D

debugging aids  
     BREAK command 62  
     LPXREF command 86  
     STEP command 111  
     TRACE command 114  
     VARS command 122  
     VIEW command 124  
     XREF command 131  
 DELETE command 72, 73  
 DOWN command 74  
 duration 36, 37, 38  
     static 37, 38  
     temporary 37, 38

---

## E

environment variable  
     VIEW\_CONSTANTS 126  
     VIEW\_ELEMENTS 125  
     VIEW\_INCLUDE 125  
     VIEW\_LINEREF 126  
     VIEW\_LOCAL 126  
     VIEW\_SAVE 106  
 error message  
     Array variable used as a scalar 34  
     Inconsistent usage 34  
     No program name to compile 68  
     Scalar variable used as an array 34  
     See also  
         Appendix D 640  
         Subscript range error 35  
 expressions 26, 39  
     arithmetic 41, 42  
     binary 45, 47

---

logical 44  
 relational 48  
 string 42

---

## F

false 44  
 floating point values  
     See  
         numeric 33  
 formulas 39  
 function 26  
 functions 39, 49  
     intrinsic 49  
     user-defined 49

---

## G

global line labels 27  
 global variables 36, 37

---

## H

HELP command 75  
 hexadecimal digits 29

---

## I

include files 52, 115  
     cross-reference 137  
     size 79  
     viewing 125, 126  
 INDENT command 76, 77, 78  
 integer  
     hexadecimal 29  
     range 33  
     variable 33  
 intrinsic functions 49

---

## L

LENGTH command 79  
 line label 27  
     cross reference 133  
     global 27  
     local 28, 50, 133  
 line number 25, 26  
     local 50, 52



- line reference 26, 115
  - viewing 125
- XREF 132, 133
- LIST command 80, 81
- LOAD command 82
- local line labels 28, 50, 133
- local variables 36, 37, 38, 49
  - viewing 126
- LOCATE command 83
- logical expressions 44
- LPLIST command 84, 85
- LPXREF command 86

---

## M

- MERGE command 87, 88
- MERGEU command 88
- MODIFY command 89
- MOVE command 91

---

## N

- NAME command 92
- NEW command 93
- null string 33
- numeric
  - constant 28
  - hexadecimal 29
  - range 33
  - variables 33

---

## O

- OLDMOD command 94, 95
- operator precedence 40
- operators 25
  - binary 40
  - unary 40
- OPTION
  - LOGICAL, used 44

---

## P

- precedence 39
- pre-scan 104
- program
  - compiling 23
  - data size 79

- line syntax 25
- size 79
- program name
  - BASIC32 command 23
  - COMPILE command 68
  - LOAD command 82
  - MERGE command 87
  - MERGEU command 87
  - NAME command 92
  - NEW command 93
- prompt
  - command 24

---

## Q

- QUIT command 97, 98

---

## R

- RECALL command 99, 100
- relational expressions 48
- RENUMBER command 101, 102
- RUN command 103, 104

---

## S

- SAVE command 105, 107
- SAVEU command 110
- scalar variable 34
- scientific notation 29
- scope 36, 37, 38
  - global 36, 37
  - illustrated 36
  - local 36, 37, 38, 49
- shared variables 37
  - viewing 126
- size
  - See
    - LENGTH command 79
- special symbols 639
- static variables
  - viewing 126
- STEP command 111, 112
- string
  - constants 29, 30
  - expressions 42
  - literals 29, 30

- maximum length 33
- string variable 33
- subprograms 26, 51
  - cross-reference 134
- subroutines 26, 50
- subscripts 34
  - See also
    - arrays 34
- substring
  - operator 43

---

## T

- TOP command 113
- TRACE command 114, 115, 117
  - display 127
  - output 127
- true 44

---

## U

- UNBREAK command 118, 119
- unnumbered
  - source programs 82, 110
- UNTRACE command 120
- UP command 121
- user-defined functions 49
  - cross-reference 134

---

## V

- variable names 32
- variables 25, 32
  - array 34, 135
  - assignment 136
  - common 38
  - COMMON statement 136
  - cross-reference 135
  - duration 36, 37, 38
  - local 135
  - LOCAL statement 136
  - numeric 33
  - scope 36, 37, 38, 49
  - shared 37, 135
  - SHARED statement 136
  - static 135
  - STATIC statement 136

---

- subscripted 34
- viewing 126
- VARS
  - display 126
- VARS command 122, 123, 125
  - output 127
- VIEW
  - FILEREF, used 132
  - LINEREF, used 132
  - SAVE used 106
  - TRACEFILE, used 114
- VIEW command 124, 125, 126, 130
- VIEW\_CONSTANTS
  - environment variable 126
- VIEW\_ELEMENTS
  - environment variable 125
- VIEW\_INCLUDE
  - environment variable 125
- VIEW\_LINEREF
  - environment variable 126
- VIEW\_LOCAL
  - environment variable 126
- VIEW\_SAVE
  - environment variable 106

---

## X

- XREF command 131, 133, 134, 135, 137