

IBM BASIC Compiler/2™ Computer Language Series

Language Reference

Programming Family

The IBM logo, consisting of the letters 'IBM' in a stylized, horizontally-lined font.

00F8662

IBM BASIC Compiler/2™ Computer Language Series

Language Reference

Programming Family



Third Edition (September 1987)

The following paragraph does not apply to the United Kingdom or any country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This publication could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time.

It is possible that this publication may contain reference to, or information about, IBM products (machines and programs), programming, or services that are not announced in your country. Such references or information must not be construed to mean that IBM intends to announce such IBM products, programming, or services in your country.

Requests for copies of this publication and for technical information about IBM products should be made to your IBM Authorized Dealer or your IBM Marketing Representative.

CodeView® and Microsoft® are registered trademarks of the Microsoft Corporation.

Operating System/2 and OS/2 are trademarks of the International Business Machines Corporation.

© Copyright International Business Machines Corporation 1984, 1987
All rights reserved. No part of this publication may be reproduced or distributed in any form or by any means without prior permission in writing from the International Business Machines Corporation.

Preface

This book is volume 3 of the three – volume set explaining the IBM BASIC Compiler/2. It contains descriptions of each of the elements of the BASIC language. The commands, functions, and statements are listed in alphabetical order.

This book is intended for people who have experience writing BASIC programs, though no experience with IBM's version of BASIC or the IBM BASIC Compiler/2 is required. Users should also be familiar with their computer and operating system.

Related Publications

If You Want To...	Refer to...
Install the product	Compile, Link, and Run
Learn basic facts about the language	Fundamentals
Know the syntax of an instruction	Language Reference
Understand error messages	Language Reference
Debug a program	Compile, Link, and Run
Compile a program	Compile, Link, and Run
Link a program	Compile, Link, and Run
Write a program	Fundamentals, Language Reference, and Compile, Link, and Run

The following books contain topics related to information in the IBM Basic Compiler Library :

- *IBM BASIC Compiler/2 Fundamentals*
- *IBM BASIC Compiler/2 Compile, Link, and Run*
- *IBM Disk Operating System Version 3.30 User's Guide*
- *IBM Disk Operating System Version 3.30 Technical Reference*

- *IBM Operating System/2 User's Guide*
- *IBM Operating System/2 User's Reference*
- *IBM Operating System/2 Programmer's Guide*
- *IBM Operating System/2 Technical Reference*

- *IBM Guide to Operations*
- *IBM Technical Reference.*

Contents

BASIC Compiler Language Reference	1
How to Use This Book	1
How This Book Is Organized	1
Book Organization	3
Terms and Conventions	5
Hexadecimal Representation	5
Compiler Metacommands	6
\$DYNAMIC Metacommand	7
\$INCLUDE Metacommand	9
\$LINESIZE Metacommand	12
\$LIST Metacommand	13
\$MODULE Metacommand	14
\$OCODE Metacommand	15
\$PAGE Metacommand	16
\$PAGEIF Metacommand	17
\$PAGESIZE Metacommand	18
\$SKIP Metacommand	19
\$STATIC Metacommand	20
\$SUBTITLE Metacommand	22
\$TITLE Metacommand	23
Compiler Commands, Functions, and Statements	24
ABS Function	29
ASC Function	30
ATN Function	31
BEEP Statement	32
BLOAD Command	33
BSAVE Command	35
CALL Statement	37
Calling BASIC Subprograms	38
Calling IBM Operating System/2 Functions	40
Calling IBM Macro Assembler/2 Subprograms	43
Calling Pascal Subprograms	46
Calling C Subprograms	48
CALLS Statement	50
CALL ABSOLUTE Statement	52
CALL INT86 Statement	54
CALL INT86X Statement	57
CASE Statement	59
CDBL Function	61

CHAIN Statement	62
CHDIR Command	64
CHR\$ Function	66
CINT Function	68
CIRCLE Statement	69
CLEAR Command	73
CLNG Function	75
CLOSE Statement	76
CLS Statement	78
COLOR Statement	80
The COLOR Statement in Text Mode	80
The COLOR Statement in Graphics Mode	84
COM(n) Statement	86
COMMAND\$ Function	88
COMMON Statement	90
COS Function	95
CSNG Function	96
CSRLIN Variable	97
CVI, CVL, CVS, CVD Functions	98
CVSMBF, CVDMBF Functions	100
DATA Statement	102
DATE\$ Variable and Statement	104
DECLARE Statement	106
DEF FN and END DEF and EXIT DEF Statements	110
DEF SEG Statement	116
DEFtype Statement	118
DIM Statement	120
DO Statement	126
DRAW Statement	130
END Statement	135
ENVIRON Statement	136
ENVIRON\$ Function	138
EOF Function	141
ERASE Statement	142
ERDEV and ERDEV\$ Variables	144
ERR and ERL Variables	146
ERROR Statement	148
EXP Function	150
FIELD Statement	151
FILEATTR Function	154
FILES Command	156
FIX Function	159
FOR and NEXT Statements	160

FRE Function	164
FREEFILE Function	166
FUNCTION Statement	168
GET Statement (Files)	173
GET Statement (Graphics)	175
GOSUB Statement	178
GOTO Statement	180
HEX\$ Function	182
IF Statement	183
INKEY\$ Variable	190
INP Function	192
INPUT Statement	193
INPUT # Statement	196
INPUT\$ Function	198
INSTR Function	200
INT Function	201
IOCTL Statement	203
IOCTL\$ Function	205
KEY Statement	206
KEY(n) Statement	212
KILL Command	215
LBOUND Function	216
LCASE\$ Function	219
LEFT\$ Function	220
LEN Function	221
LET Statement	223
LINE Statement	224
LINE INPUT Statement	228
LINE INPUT # Statement	230
LOC Function	232
LOCATE Statement	234
LOCK Statement	237
LOF Function	239
LOG Function	241
LPOS Function	243
LPRINT and LPRINT USING Statements	244
LSET and RSET Statements	246
LTRIM\$ Function	248
MID\$ Function and Statement	250
MKDIR Command	253
MKI\$, MKL\$, MKS\$, MKD\$ Functions	255
MKSMBF\$, MKDMBF\$ Functions	257
NAME Command	259

OCT\$ Function	261
ON COM(n) Statement	262
ON ERROR Statement	265
ON...GOSUB and ON...GOTO Statements	267
ON KEY(n) Statement	269
ON PEN Statement	273
ON PLAY(n) Statement	275
ON SIGNAL Statement	278
ON STRIG(n) Statement	281
ON TIMER Statement	284
OPEN Statement	286
OPEN "COM. . . Statement	292
OPEN "PIPE... Statement	298
OPTION BASE Statement	300
OUT Statement	302
PAINT Statement	303
PEEK Function	311
PEN Statement and Function	313
PLAY Statement	316
PLAY(n) Function	320
PMAP Function	321
POINT Function	324
POKE Statement	327
POS Function	329
PRINT Statement	330
PRINT USING Statement	333
String Fields	333
Numeric Fields	334
PRINT # and PRINT # USING Statements	340
PSET and PRESET Statements	343
PUT Statement (Files)	345
PUT Statement (Graphics)	347
RANDOMIZE Statement	353
READ Statement	356
REDIM Statement	358
REM Statement	362
RESET Command	364
RESTORE Statement	365
RESUME Statement	366
RETURN Statement	368
RIGHT\$ Function	369
RMDIR Command	370
RND Function	372

RTRIM\$ Function	375
RUN Command	377
SADD Function	380
SCREEN Function	382
SCREEN Statement	384
SETMEM Function	387
SGN Function	389
SHARED Statement	390
SHELL Function	393
SHELL Statement	396
SIGNAL Function	400
SIN Function	402
SOUND Statement	403
SPACE\$ Function	407
SPC Function	408
SQR Function	409
STATIC Statement	410
STICK Function	412
STOP Statement	414
STR\$ Function	416
STRIG Statement and Function	417
STRIG(n) Statement	419
STRING\$ Function	420
SUB and END SUB and EXIT SUB Statement	421
SWAP Statement	424
SYSTEM Command	425
TAB Function	426
TAN Function	427
TIME\$ Variable and Statement	428
TIMER Function and Statement	430
TRON and TROFF Commands	433
TYPE, ENDTYPE Statements	435
UBOUND Statement	438
UCASE\$ Function	441
UNLOCK Statement	442
VAL Function	444
VARPTR Function	446
VARPTR\$ Function	448
VARSEG Function	450
VIEW Statement	452
VIEW PRINT Statement	456
WAIT Statement	457
WHILE and WEND Statements	459

WIDTH Statement	461
WINDOW Statement	464
WRITE Statement	470
WRITE # Statement	472
Appendix A. BASIC Compiler Error Messages	A-1
Errors While Compiling a Program	A-2
Prompt Errors	A-2
Listing Errors	A-6
Errors while Running a Program	A-23
Errors that Cannot be Trapped	A-32
Communication Errors	A-34
Appendix B. ASCII Character Codes	B-1
Appendix C. Scan Codes	C-1
Extended Codes	C-6
Appendix D. CodeView Error Messages	D-1
Appendix E. Linker Error Messages and Limits	E-1
Appendix F. Library Manager Error Messages	F-1
Index	1

BASIC Compiler Language Reference

How to Use This Book

This section tells you how to find information in this book and describes the notational conventions that this book uses.

How This Book Is Organized

This book gives specific information about BASIC statements, functions, and commands. The information is organized in the following manner:

- The compiler metacommands are listed first in alphabetical order. The metacommands are commands that control the operation of the compiler.
- Next, each command, function, and statement of the BASIC language is listed in alphabetical order.
- **Appendix A**, “Error Messages,” lists all the error messages you can receive while using the compiler.
- **Appendix B**, “ASCII Character Codes,” is a table of all the ASCII characters and their decimal codes.
- **Appendix C**, “Scan Codes,” lists the scan codes, in decimal and in hexadecimal, for all the keys on the PC keyboard and the IBM Enhanced Keyboard.

- **Appendix D**, “CodeView Error Messages” lists all the CodeView error messages.
- **Appendix E**, “Linker Error Messages and Limits,” lists error messages produced by the Linker.
- **Appendix F**, “Library Manager Error Messages,” lists error messages produced by the Library Manager.

For information on the differences between IBM BASIC Compiler 2.00 and IBM BASIC Compiler/2, see chapter 1 of *IBM BASIC Compiler/2 Fundamentals*.

The following sample pages describe the organization of the entries in *IBM BASIC Compiler/2 Language Reference*.

Purpose

Gives a brief functional description of the statement.

Format

Shows the correct format for the statement. The format of the statements follow these rules:

- Items in square brackets [] are optional.
- Items separated by a vertical bar (|) mean that you can enter one of the separated items. For example:

ON|OFF

means you can enter ON or OFF, but not both.

- An ellipsis (...) shows you can repeat an item as many times as you want.
- You must include all punctuation, where it is shown (commas, parentheses, angle brackets, slashes, or semicolons), except square brackets.
- Ctrl+Break and Ctrl+C perform the same function. Any time Ctrl+Break is documented, you may also use Ctrl+C.
- Italics are used for:
 - New terms when they are first defined in a book.
Example: An *object module* is produced...
 - Variables in command formats and within text. You supply these items.
Example: TIME [hh.mm.ss.xx]
 - Book titles.
Example: The *IBM BASIC Compiler/2 Language Reference*.

Book Organization

- Boldface is used for:
 - Anything you must type exactly as it appears in the book.
Example: Now, type **dir** and press...
 - Anything that appears on a screen that is referred to in text.
Example: The **Stack Overflow** message tells you...
 - Single alphabetic keys on the keyboard.
Example: Type **S** and...
- Small capital letters are used for:
 - Sample file names in text.
Example: Use the AUTOEXEC file...
 - Operating System and BASIC programming commands.
Example: The COPY command...
 - Suffixes (file or language extensions) used alone.
Example: A .BAT file is required...
 - All acronyms and other fully capitalized words.
Examples: IBM, DOS
 - Library names.
Example: Place it in the LIB1.LIB...

Comments

Gives detailed information about the statement.

Book Organization

Examples

Examples demonstrate how this statement can be used in a program. An explanation is included where the purpose of the example is not obvious.

Terms and Conventions

Throughout this book, the following terms and conventions apply:

BASIC is the BASIC language that has been developed specifically for IBM.

BASIC Interpreter

is the IBM BASIC Interpreter. This is the interactive version of BASIC that is included with DOS and OS/2.

compiler is the IBM BASIC Compiler/2.

disk refers to either a diskette or a fixed disk.

diskette refers only to a diskette.

DOS IBM Disk Operating System Version 3.30 (DOS).

DOS mode

both DOS 3.30 and the DOS mode of IBM Operating System/2^{TM1}

OS/2^{TM1} IBM Operating System/2.

OS/2 mode

the OS/2 mode of Operating System/2

Hexadecimal Representation

This book represents hexadecimal numbers with the letter **H**, such as 59H.

¹ OS/2 and Operating System/2 are trademarks of IBM Corporation.

Compiler Metacommands

The metacommands are commands that supply information to the compiler or tell it how to operate, but do not produce any executable code. The metacommands are as follows:

<code>\$DYNAMIC</code>	<code>\$PAGE</code>
<code>\$INCLUDE</code>	<code>\$PAGEIF</code>
<code>\$LINESIZE</code>	<code>\$PAGESIZE</code>
<code>\$LIST</code>	<code>\$SKIP</code>
<code>\$MODULE</code>	<code>\$STATIC</code>
<code>\$OCODE</code>	<code>\$SUBTITLE</code>
	<code>\$TITLE</code>

You include metacommands in your source file as part of a remark statement; that is, you must insert metacommands after the `REM` keyword or the single quote. You can have more than one metacommand in a remark statement. For example:

```
REM $LINESIZE:120 $PAGESIZE:55
```

If you use `$INCLUDE`, it must be the *last* metacommand on the line. You can separate metacommands on one line by spaces, tabs, or line feed characters. If the compiler sees any other character that is not part of a metacommand, it ignores the rest of the remark.

The compiler does not print the header for the source listing until the compiler scans the first line of the program for metacommands. In this way, metacommands such as `$TITLE` can affect the first page of the source listing.

\$DYNAMIC Metacommand

Purpose

The \$DYNAMIC metacommand causes the compiler to allocate dynamically all subsequently dimensioned arrays.

Format

\$DYNAMIC

Comments

You can only dimension a statically allocated array once, but you can redimension dynamic arrays using ERASE, DIM, and REDIM at any point in the program.

Static allocation of array space is the default. If you issue the \$DYNAMIC metacommand, the compiler treats all following array declarations in a DIM statement as dynamic array declarations.

\$DYNAMIC

Metacommand

Examples

The following example demonstrates the use of static and dynamic arrays within the same program:

```
120 ' $STATIC
130 DIM C(5,5)
140 C(5,5)=1
145 C=5
150 ERASE C
160 PRINT C,C(5,5)
180 ' $DYNAMIC
190 DIM A(20,20,20)
200 I = 40
210 DIM B(I,I)
220 'ASSIGN VALUES INTO A AND B
223 A(1,1,1)=3
225 B(1,1) = 17
227 'ERASE AND REDIMENSION A
230 ERASE A
240 REDIM A(5,5,5)
260 PRINT B(1,1),A(1,1,1)
270 END
```

Results

```
5      0
17     0
```

\$INCLUDE Metacommand

Purpose

The \$INCLUDE metacommand tells the compiler to include source code from another BASIC file.

Format

\$INCLUDE: '*filespec*'

Comments

filespec is a string expression for the file specification. It can contain a path and must conform to the rules outlined under "File Names" and "File Specification" in *IBM BASIC Compiler/2 Fundamentals*; otherwise, an error occurs.

The default extension for the included file is .BAS. The included file must be in ASCII format. The file specification must be enclosed in single quotation marks.

The compiler imbeds the specified file into the source file at the point where it encounters this metacommand. The included file can be a subroutine, a single line, or any type of partial program, but it must be written in IBM BASIC.

When using the \$INCLUDE metacommand, the following restrictions apply:

- The \$INCLUDE must be the *last* metacommand on the line.
- The \$INCLUDE cannot immediately follow a line continuation.
- An included file cannot end with a line continuation.
- CodeView cannot debug a program with included files.

\$INCLUDE

Metacommand

You should take care that any variables in the included files match their counterparts in the main program, and that included lines do not contain GOTOS to nonexistent lines or similarly erroneous code.

Included files can be very useful for COMMON declarations existing in more than one program or for subroutines that you might have in an external library of subroutines.

If you create the included file using the BASIC program editor from within the BASIC Interpreter, you must remember to save it using SAVE with the **A** option.

Also, because the interpreter does not support the \$INCLUDE metacommand, a program that contains a \$INCLUDE metacommand might not run correctly if you try to run it under the interpreter.

If you use an editor other than the BASIC program editor, be sure that editor saves your file in ASCII format.

If you use a text editor other than the BASIC program editor, you can create a file of lines **without line numbers**. This feature can make it very easy to include the same file in many different programs.

You can nest \$INCLUDE metacommands up to five levels deep. If you are nesting them deeper than three levels under DOS mode, you must create a CONFIG.SYS file (or modify an existing one) to contain the statement:

FILES = xx

Where xx is the number of nesting levels plus 3 which BASIC uses.

Under DOS mode the maximum number of file handles per process is 20.

Under OS/2 mode, additional file handles can be obtained by using the OS/2 mode DOSSETMAXFH function.

\$INCLUDE Metacommand

Examples

The first example includes the file named SUBR.BAS:

```
90 REM $INCLUDE: 'SUBR'
```

The second example uses the \$INCLUDE metacommand in a remark beginning with a single quote. The included file is named PROC.ASC:

```
9999 ' $INCLUDE:'PROC.ASC'
```

The following example uses an included file that contains a routine for calculating the average of two numbers:

```
10 PRINT "ENTER A NUMBER FROM 1 TO 10"  
20 INPUT A  
30 PRINT "ENTER ANOTHER NUMBER 1-10"  
40 INPUT B  
50 ' $INCLUDE:'AVERAGE.BAS'  
60 PRINT "THE AVERAGE OF THE TWO IS ",AVG  
70 END
```

Also see the comments under “COMMON Statement”, in this book.

\$LINESIZE

Metacommand

Purpose

The \$LINESIZE metacommand tells the compiler to change the maximum line width in the listing file.

Format

\$LINESIZE: *number*

Comments

number is a constant in the range 40 through 255.

The default line size is 80 characters.

The \$LINESIZE metacommand must appear in the first line of your program if you want the entire source listing to be the same width. If \$LINESIZE appears anywhere else in the program, it changes only the width of the following lines.

Examples

```
1000 REM $LINESIZE: 120
```

\$LIST Metacommand

Purpose

The \$LIST metacommand turns the listing of source code on and off.

Format

\$LIST + | -

Comments

The default setting is \$LIST + .

\$LIST+ turns the listing of source code on. \$LIST- turns the source code listing off.

The compiler always lists errors.

\$LIST is useful, for instance, if you make a change to a large program and you want a listing of only the change. You can cause a partial listing by using \$LIST- in the first line of the program to turn the source code listing off, then place \$LIST+ at the start of the new code and \$LIST- at the end of the new code.

Examples

```
100 REM $LIST-
```

\$MODULE

Metacommand

Purpose

The \$MODULE metacommand allows you to change the internal module name that is passed to the linker.

Format

\$MODULE : '*string*'

Comments

string is a string expression one through eight characters long.

The \$MODULE metacommand is useful when you want the module name to be different from that of the source file. If you use this metacommand, it must appear before the first executable statement.

Examples

```
REM $MODULE: 'TEMP'
```

\$OCODE Metaccommand

Purpose

The \$OCODE metaccommand turns the listing of object code on and off.

Format

\$OCODE + | -

Comments

The default setting is \$OCODE-. The \$OCODE metaccommand controls listing the generated code in the same way \$LIST controls the source listing: \$OCODE+ turns the listing of object code on, \$OCODE- turns the object code listing off.

\$OCODE works independently of the setting of the /A parameter when you compile your program. /A includes all the object code unless you turn it off with \$OCODE-. You can use \$OCODE to list just parts of the object code.

The format of the object code listing is basically like an assembler listing, with code addresses and operation mnemonics.

Examples

```
REM $OCODE-
```

\$PAGE

Metacommand

Purpose

The \$PAGE metacommand tells the compiler to force a new page in the compiler listing file.

Format

\$PAGE

Comments

The compiler forces a page by putting the form feed character (OCH) into the listing file and writing a heading for the new page.

Examples

REM \$PAGE

\$PAGEIF **Metaccommand**

Purpose

The \$PAGEIF metaccommand skips to the next page if there are less than n printable lines left on the current page.

Format

\$PAGEIF: n

Comments

n is a numeric constant in the range of 1 through 255.

The last six lines of each page are always blank. The compiler does not consider these lines to be printable lines.

Examples

```
100 REM $PAGEIF: 10
```

\$PAGESIZE

Metaccommand

Purpose

The \$PAGESIZE metaccommand sets the number of lines per page in the compiler listing file.

Format

\$PAGESIZE: *n*

Comments

n is a numeric constant in the range of 15 through 255. The default page size is 66.

The *n* specifies the number of lines that fit on one piece of paper. The compiler separates pages in the listing file by form feed characters (OCH), and each page starts with a heading.

If *n* is 255, the compiler does not add form feed characters to the listing file, and the listing file prints without page breaks.

The \$PAGESIZE metaccommand must appear in the first line of your program if you want all the pages in your source listing to be the same length. If \$PAGESIZE appears anywhere else in the program, it changes only the length of the following pages.

Examples

```
100 REM $PAGESIZE: 60
```

\$\$SKIP Metacommand

Purpose

The \$\$SKIP metacommand skips n printable lines or to the end of the page, whichever occurs first.

Format

\$\$SKIP: n

Comments

n is a numeric constant in the range of 1 through 255.

The last six lines of each page are always blank. The compiler does not consider these lines to be printable lines.

Examples

```
100 REM $$SKIP: 10
```

\$STATIC

Metacommand

Purpose

The \$STATIC metacommand causes the compiler to allocate statically all subsequently dimensioned arrays.

Format

\$STATIC

Comments

Static allocation of array space is the default. You can dimension statically allocated arrays only once, while you can redimension dynamic arrays using ERASE, DIM, and REDIM at any point in the program.

If \$STATIC is in effect, the compiler statically allocates space for any array whose bounds you declare using integer constants. If an upper or lower bound is not an integer constant, the compiler dynamically allocates the array at runtime.

\$STATIC Metacommand

Examples

The following example demonstrates the use of static and dynamic arrays within the same program:

```
120 ' $STATIC
130 DIM C(5,5)
140 C(5,5)=1
145 C=5
150 ERASE C
160 PRINT C,C(5,5)
180 ' $DYNAMIC
190 DIM A(20,20,20)
200 I = 40
210 DIM B(I,I)
220 'ASSIGN VALUES INTO A AND B
223 A(1,1,1)=3
225 B(1,1) = 17
227 'ERASE AND REDIMENSION A
230 ERASE A
240 REDIM A(5,5,5)
260 PRINT B(1,1),A(1,1,1)
270 END
```

Results:

```
5      0
17     0
```

\$SUBTITLE

Metacommand

Purpose

The \$SUBTITLE metacommand sets the subtitle on the listing page.

Format

\$SUBTITLE: *'string'*

Comments

string is a character string constant that you enclose in single quotation marks. The maximum length of the string is 60 characters. If a program does not contain a \$SUBTITLE command, the compiler uses the null string as a subtitle.

The compiler prints the specified string on each page of the listing. The compiler might truncate a long title string if \$LINESIZE is set to a value less than 80.

The \$SUBTITLE metacommand must appear in the first line of your program if you want the subtitle to appear on the first page of the source listing. If \$SUBTITLE appears anywhere else in the program, it affects only the following pages.

Examples

```
100 ' $SUBTITLE: 'Entry Routine'
```

\$TITLE Metacommand

Purpose

The \$TITLE metacommand provides a title for the compiler listing.

Format

\$TITLE: *'string'*

Comments

string is a character string constant that you enclose in single quotation marks. The maximum length of the string is 60 characters.

The compiler prints the specified string on each page of the listing. The compiler might truncate a long title string if \$LINESIZE is set to a value less than 80.

The \$TITLE metacommand must appear in the first line of your program if you want the title to appear on the first page of the source listing. If \$TITLE appears anywhere else in the program, it affects only the title on the following pages.

Examples

```
100 ' $TITLE: 'Update Program'
```

Compiler Commands, Functions, and Statements

The statements, functions, and commands that the BASIC Compiler/2 supports are listed below. They are described in detail in the sections that follow.

Arithmetic

ABS	SGN
ATN	SIN
COS	SQR
EXP	TAN
LOG	

Communications

COM(N)	OPEN "COM...
ON COM(N)	

Data Conversion

ASC	FIX
CDBL	HEX\$
CHR\$	INT
CINT	MKI\$, MKL\$, MKS\$, MKD\$
CLNG	MKSMBF\$, MKDMBF\$
CSNG	OCT\$
CVI, CVL, CVS, CVD	STR\$
CVSMBF, CVDMBF	VAL

Compiler Commands, Functions, and Statements

Defining Variables

CLEAR	OPTION BASE
COMMON	REDIM
DEFTYPE	SHARED
DIM	STATIC
ERASE	TYPE/ENDTYPE
FRE	

Error Handling

ERDEV	ERL
ERDEV\$	ERROR
ERR	ON ERROR

File and Subdirectory Management

CHDIR	MKDIR
FILES	NAME
KILL	RMDIR

Graphics

CIRCLE	POINT
COLOR (GRAPHICS MODE)	PSET
DRAW	PRESET
GET (GRAPHICS)	PUT (GRAPHICS)
LINE	SCREEN
PAIN	VIEW
PMAP	WINDOW

Hardware Interface

CALL INT86	OUT
CALL INT86X	TIMER FUNCTION
INP	TIMER STATEMENT
ON TIMER	WAIT

Compiler Commands, Functions, and Statements

Input/Output

BLOAD	LOF
BSAVE	LPOS
CLOSE	LPRINT
DATA	LPRINT USING
EOF	LSET
FIELD	OPEN
FILEATTR	PRINT
FREEFILE	PRINT USING
GET (FILES)	PRINT#
INKEY\$	PRINT# USING
INPUT	PUT (FILES)
INPUT#	READ
INPUT\$	RESET
IOCTL	RSET
IOCTL\$	SPC
LINE INPUT	TAB
LINE INPUT#	UNLOCK
LOC	WRITE
LOCK	WRITE#

Joystick and Light Pen Interface

ON PEN	STICK
ON STRIG(N)	STRIG
PEN	STRIG(N)

Key Trapping

KEY	ON KEY(N)
KEY(N)	

Compiler Commands, Functions, and Statements

Memory References

DEF SEG
PEEK
POKE
SADD

VARPTR
VARPTR\$
VARSEG

Operating System Interface

CALL INT86
CALL INT86X
COMMAND\$
DATE\$
ENVIRON
ENVIRON\$
ON SIGNAL

OPEN "PIPE...
SETMEM
SHELL FUNCTION
SHELL STATEMENT
SIGNAL
TIME\$

Program Flow Control

CALL
CALLS
CALL ABSOLUTE
CASE
CHAIN
DO
END
FOR/NEXT
GOSUB/RETURN
GOTO

IF
ON...GOSUB
ON...GOTO
RESUME
RETURN
RUN
STOP
SYSTEM
WHILE/WEND

Compiler Commands, Functions, and Statements

Sound

BEEP
ON PLAY(N)
PLAY

PLAY(N)
SOUND

Strings

FRE
INSTR
LCASE\$
LEFT\$
LEN
LTRIM\$

MID\$
RIGHT\$
RTRIM\$
SPACES\$
STRING\$
UCASE\$

Subprogram Definition

DECLARE
DEF FN

FUNCTION
SUB

Text Screen

CLS
COLOR (TEXT)
CSRLIN
LOCATE

POS
SCREEN
VIEW PRINT
WIDTH

Miscellaneous

LBOUND
LET
RANDOMIZE
REM
RESTORE

RND
SWAP
TRON, TROFF
UBOUND

Purpose

The ABS function returns the absolute value of the expression x .

Format

$v = \text{ABS}(x)$

Comments

x can be any numeric expression.

The absolute value of a number is always positive or 0.

Examples

This example shows that the absolute value of -35 is positive 35:

```
PRINT ABS(7*(-5))
```

Results:

35

ASC Function

Purpose

The ASC function returns the ASCII code for the first character of a string (x\$).

Format

$v = \text{ASC}(x\$)$

Comments

x\$ can be any string expression.

The result of the ASC function is a numeric value that is the ASCII code of the first character of the string x\$. See Appendix B, "ASCII Character Codes," for a list of ASCII codes. If x\$ is null, BASIC returns an **illegal function call** error.

The ASC function is the opposite of the CHR\$ function, which converts an ASCII code to a character.

Examples

This example shows that the ASCII code for a capital T is 84. The statement PRINT ASC("TEST") gives you the same result.

```
100 X$ = "TEST"  
200 PRINT ASC(X$)
```

Results:

84

Purpose

The ATN function returns the arctangent of x .

Format

$v = \text{ATN}(x)$

Comments

x can be a numeric expression of any type.

The ATN function returns the angle whose tangent is x . The result is a value in radians in the range $-\text{PI}/2$ through $\text{PI}/2$, where $\text{PI} = 3.141593$.

If you want to convert radians to degrees, multiply by $180/\text{PI}$.

Examples

The first example shows the use of the ATN function to calculate the arctangent of 3:

```
100 PRINT ATN(3)
```

Results:

```
1.249046
```

The second example finds the angle whose tangent is 1.

```
100 PI=3.141593
200 RADIANS=ATN(1)
300 DEGREES=RADIANS*180/PI
400 PRINT RADIANS,DEGREES
```

Results:

```
.7853982      45
```

BEEP Statement

Purpose

The BEEP statement causes the speaker to sound, beep.

Format

BEEP

Comments

The BEEP statement causes the speaker to sound at 800 Hz for 1/4 second. BEEP has the same effect as:

```
PRINT CHR$(7);
```

Examples

In this example, the program checks to see if X is out of range. If it is, the computer warns you by beeping.

```
100 IF X < 20 THEN BEEP
```

Purpose

The BLOAD command loads a memory image file, created by BSAVE, into memory.

Format

BLOAD *filespec* [,*offset*]

Comments

filespec is a string expression for the file specification. It can contain a path and must conform to the rules outlined under “File Names” and “File Specification” in *IBM BASIC Compiler/2 Fundamentals*; otherwise, an error occurs.

offset is an integer value in the range of 0 through 65535. This is an offset at which BASIC loads the file into the current segment specified by the latest DEF SEG statement.

If you do not specify *offset*, BASIC uses the offset you specified at BSAVE. That is, BASIC loads the file into the same location from which you saved the file using BSAVE.

When you run a BLOAD command, BASIC loads the named file into memory, starting at the location that *offset* specifies.

If you do not specify the device name in *filespec*, BASIC uses the default drive.

You should use BLOAD with a file that you have previously saved with BSAVE. BLOAD and BSAVE are useful for loading and saving machine language programs, but you do not have to use them strictly for machine language programs. For example, you can specify any segment as the target or source for these statements, through the DEF

BLOAD

Command

SEG statement. You can save and display screen images from or to the screen buffer.

See “Other Interface Methods” in *IBM BASIC Compiler/2 Fundamentals* for more information on using BLOAD with machine language programs.

Warning: BASIC does not check the offset of the current segment where you are loading with BLOAD. You can use BLOAD anywhere in memory, but it is your responsibility to be sure that a file loaded with BLOAD does not conflict with the current contents of memory.

For OS/2 users:

In OS/2 mode, BLOAD treats the segment as a selector. Illegal memory references may cause exceptions or return a **Permission denied** error.

Examples

This example works only in DOS mode.

This example loads the screen buffer, which is at segment address HB8000, for the IBM Color/Graphics Monitor Adapter. To load the screen buffer for the IBM Monochrome Display and Printer Adapter, you would have to change line 300 to read &HB000. Line 500 loads PICTURE at offset 0, segment &HB800.

```
100 'load the screen buffer
200 'point SEG at screen buffer
300 DEF SEG= &HB800
400 'load PICTURE into screen buffer
500 BLOAD "PICTURE",0
```

The example for the BSAVE command (see the next entry) illustrates how PICTURE was saved.

BSAVE Command

Purpose

The BSAVE command saves portions of the computer's memory on the specified device.

Format

BSAVE *filespec,offset,length*

Comments

- filespec* is a string expression for the file specification. It can contain a path and must conform to the rules outlined under "File Names" and "File Specification" in *IBM BASIC Compiler/2 Fundamentals*; otherwise, an error occurs.
- offset* is a 2-byte integer value in the range of 0 through 65535. This is the offset into the segment that you declared in the last DEF SEG. Saving starts from this location. See the DEF SEG statement for more information.
- length* is an integer expression in the range of 1 through 65535. This is the length of the memory image that you want to save.

If you do not specify *offset* or *length*, BASIC returns **Illegal syntax** and does not save the portion of memory.

If you do not specify the device name in *filespec*, BASIC uses the default disk drive.

When you use the DEF SEG statement, you can specify any segment as the source segment for the BSAVE data. For example, you can save an image of the screen by doing a BSAVE of the screen buffer.

BSAVE Command

For OS/2 users:

In OS/2 mode BSAVE treats the segment as a selector. Illegal memory references may cause exceptions or return a **Permission denied** error.

Examples

This example works only in DOS mode.

As explained under the BLOAD Command, the segment address of the 16K-byte screen buffer for the IBM Color/Graphic Monitor Adapter is HB8000. The segment address of the 4K-byte screen buffer for the IBM Monochrome Display and Printer Adapter is HB0000.

Use the DEF SEG statement to set up the segment address to the start of the screen buffer. The offset of 0 and length &H4000 tell BASIC to save the entire 16K-byte screen buffer.

```
100 'Save the color screen buffer
200 'point segment at screen buffer
300 DEF SEG= &HB800
400 'save buffer in file PICTURE
500 BSAVE "PICTURE",0,&H4000
```

CALL Statement

Purpose

The CALL statement transfers control to a subprogram.

Format

[CALL] *subname* [([BYVAL|SEG] *parameter* [, [BYVAL|SEG] *parameter*] ...)]

Comments

- CALL** is an optional keyword. If you use the CALL keyword, enclose the parameters in parentheses. If you omit the CALL keyword, do not enclose the parameters in parentheses.
- subname*** is the name of the subprogram that you want to call.
- BYVAL** is a keyword that can precede a parameter to indicate that you want to pass the actual value of the parameter to the subprogram rather than the address of the parameter. You can only use the BYVAL keyword to pass a parameter to a subprogram that is not written in BASIC. Also, do not use BYVAL on array parameters.
- SEG** is a keyword that can precede a parameter to indicate that you want to pass the segmented address of the parameter to the subprogram. BASIC passes the address as a 4-byte integer representing a segment address and an offset. You can only use the SEG keyword to pass a parameter to a subprogram that is not written in BASIC.
- parameter*** is the name of a simple variable or an array that you want to pass to the subprogram. If the parameter is an array, its name must be followed by a pair of parentheses (for example, "ARRAY%()"). You can pass a maximum of 60 parameters to a subprogram.

CALL

Statement

You can use the CALL statement to call BASIC compiled subprograms, IBM Macro Assembler/2 subprograms, IBM Pascal Compiler/2 subprograms, and IBM C/2 compiled subprograms.

Note: You can specify the BYVAL and SEG keywords in a CALL statement if either you did not list the procedure's parameters in the DECLARE statement or you specified BYVAL and SEG in the DECLARE statement. Otherwise, you get a **Parameter type mismatch** error at compile time.

For more information, see "SUB and END SUB and EXIT SUB Statement" and the "CALLS Statement" in this book and the "Modular Programming" section in *IBM BASIC Compiler/2 Fundamentals*.

Calling BASIC Subprograms

When BASIC passes a simple variable or array element to a subprogram, it passes by reference. This means that the subprogram knows the address of the variable and can change the value of the variable in the calling routine. The subprogram can change the value of the variable by assigning a new value to its corresponding formal parameter in the formal parameter list through an assignment statement or any other statement that assigns values to a memory location.

You can also pass expressions as arguments to subprograms. When BASIC encounters an expression in the formal parameter list, it assigns the result of the expression to a temporary variable. BASIC then passes this variable by reference to the subprogram. This is functionally equivalent to call by value, where BASIC passes the value itself rather than the address of a variable.

You can prevent a subprogram from changing a simple variable or array element's value by enclosing the argument within an extra set of parentheses. This forces BASIC to treat the argument as an expression. For example, in the following call:

```
CALL SUBPROG1 ((A),B)
```

CALL Statement

the value of A cannot be changed by SUBPROG1. However, the value of B can be changed.

You can use the CALL statement to pass array arguments to BASIC subprograms. You specify an array argument by following the array name with parentheses. For example:

```
CALL MATADD2(5,10,ARRAY1(),ARRAY2(),TOTAL())
```

Two types of programming errors are commonly made in calling BASIC subprograms:

- Argument lists in which the order, type, or number of arguments passed to the subprogram do not exactly match the corresponding formal parameters in the subprogram.

If you declare the parameters of a function or a subprogram in a DECLARE statement, the IBM BASIC Compiler/2 checks that the number of parameters and the types of the parameters that you are going to pass to the procedure are the same as those you declared in the DECLARE statement.

If you do not declare the parameters in a DECLARE statement, the compiler does not check for this discrepancy. An error message is not generated, but subtle errors may occur. For example, type mismatching can occur when an integer argument is mistakenly matched with a parameter that expects to receive a single-precision argument.

- Aliasing of variables.

Aliasing occurs whenever an argument passed to a subprogram can be referred to in the subprogram more than one way. This commonly occurs when the same nonexpression argument is passed more than once to a subprogram. Passing variables both in COMMON blocks and as an argument, or passing both an array element and the array itself, can cause aliasing. The IBM BASIC Compiler/2 does not support or check for aliasing. If aliasing of variables occurs, the results are unpredictable.

CALL

Statement

Examples

The following program calls a BASIC subprogram that converts decimal notation into binary notation:

```
100 REM BINARY CONVERSION
110 DEFINT A, B, C, I
120 PRINT "DECIMAL TO BINARY CONVERSION"
130 PRINT
140 INPUT "ENTER NUMBER FROM 0 TO 32767: "; B
150 CALL BINARY(B,BINSTR$)
160 PRINT "THE BINARY VALUE IS " + BINSTR$
170 END
180 REM CONVERT TO BINARY REPRESENTATION
190 SUB BINARY(C,D$) STATIC
200 DIM A(15)
210 FOR I = 15 TO 0 STEP -1
220   IF C < 2 ^ I THEN A(I) = 0: GOTO 250
230   A(I) = 1
240   C = C - 2 ^ I
250 NEXT I
260 REM CONVERT TO STRING FOR VIEWING PURPOSES
270 D$ = ""
280 FOR I = 15 TO 0 STEP -1
290   D$ = D$ + RIGHT$(STR$(A(I)),1)
300 NEXT I
310 END SUB
```

Calling IBM Operating System/2 Functions

Note: Even though they are called "functions," you must declare the IBM Operating System/2 functions as SUBS in your program. Their values are returned in the AX register and cannot be accessed through BASIC.

You can pass parameters to OS/2 functions either by reference or by value.

Normally, parameters are passed to OS/2 functions by reference. The 2-byte offset of the argument is passed, which allows the function to change the contents of the argument. Some OS/2 functions may require the full 4-byte address (segment and offset) to work with the parameter. You can pass the full address to the function with the SEG keyword.

CALL Statement

You can also pass parameters to OS/2 functions by value. If you specify the BYVAL keyword before an argument, its value is passed to the function instead of its address.

For details on the OS/2 functions and their parameters, refer to *IBM Operating System/2 Technical Reference*.

Note: BASIC is not reentrant. BASIC does not support OS/2 functions that require an address within your program that is to be executed.

CALL

Statement

Examples

The following example uses the OS/2 DosGetDateTime function to print the current date and time.

```
'Define a data type to hold the results returned by the
'function
TYPE DateTime
    HoursMinutes      AS INTEGER
    SecondsHundredths AS INTEGER
    DayMonth          AS INTEGER
    Year              AS INTEGER
    TimeZone          AS INTEGER
    DayOfWeek         AS INTEGER
END TYPE
DIM DateTime AS DateTime

'Declare the OS/2 function
DECLARE SUB DosGetDateTime(SEG DateTime AS DateTime)

'Define our own function to convert pieces of the date and
'time into strings for printing
FUNCTION Format$(N AS INTEGER, Length AS INTEGER) STATIC
    Format$ = RIGHTS$(STRING$(Length,"0")+MID$(STR$(N),2),Length)
END FUNCTION

'Call the OS/2 function
CALL DosGetDateTime(DateTime)

'Convert the returned values and print them
CurrentTime$ = Format$(DateTime.HoursMinutes MOD 256,2) _
    + ":" + Format$(DateTime.HoursMinutes\256,2) _
    + ":" + Format$(DateTime.SecondsHundredths MOD 256,2)
CurrentDate$ = Format$(DateTime.DayMonth\256,2) _
    + "-" + Format$(DateTime.DayMonth MOD 256,2) _
    + "-" + Format$(DateTime.Year,4)
PRINT CurrentDate$, CurrentTime$

END
```

Results:

```
09-15-1987  11:37:50
```

CALL Statement

Calling IBM Macro Assembler/2 Subprograms

You can pass parameters to IBM Macro Assembler/2 subprograms either by reference or by value.

Normally, parameters are passed to IBM Macro Assembler/2 subprograms by reference. The 2-byte offset of the argument is passed, which allows the subprogram to change the contents of the argument. Some IBM Macro Assembler/2 subprograms may require the full 4-byte address (segment and offset) to work with the parameter. You can pass the full address to the subprogram with the `SEG` keyword.

You can also pass parameters to IBM Macro Assembler/2 subprograms by value. If you specify the `BYVAL` keyword before an argument, its value is passed to the subprogram instead of its address and the subprogram cannot change the value of the original argument.

You should not pass arrays as formal parameters to assembler language subprograms. Instead, pass the base element of the array by reference if the assembler language subprogram needs to access the entire array.

When transporting assembler language subprograms from the BASIC Interpreter to the compiler, the string descriptor requires four bytes rather than three (low byte, high byte of the length, followed by low byte, high byte of the address) because the IBM BASIC Compiler/2 allows strings up to 32767 bytes in length. If your assembler language subprogram uses string arguments, you should recode the subprogram to take this difference into account.

The linker determines the starting address of the subprogram; `DEF SEG` is unimportant when calling a Macro Assembler subprogram from a compiled program.

CALL Statement

Examples

Note: Because of the structure for the assembler language subprogram, the following example is for DOS mode only. To work in OS/2 mode, the subprogram would need to be restructured. However, the BASIC program would not have to be changed.

BASIC Program:

```
'ASMTEST.BAS
' This example calls an assembly language subprogram.
' -- LINK ASMTEST+ADD;
' Declare the assembly routine:
'   The first two arguments will be passed by value.
'   The result variable's segmented address will be passed.
'   The routine is called "Sum" locally but "Add" externally.
DECLARE SUB Sum ALIAS "Add" (BYVAL X%, BYVAL Y%, SEG Result%)

OldResult% = 0
FOR I%=1 TO 10

    Sum I%, OldResult%, NewResult%
    PRINT "The sum of numbers from 1 to "I%" is "NewResult%
    OldResult% = NewResult%

NEXT I%
END
```

CALL Statement

Assembler Language Subprogram:

```
; ADD.ASM
; Routine that adds two integers.
;
; The BASIC call is:
;
;   Add X%, Y%, Sum%
;
;   WHERE:
;
;       X%, Y% = the numbers to add.
;       Sum%   = the result.

X      EQU    12          ; BP offsets for parameters
Y      EQU    10
SUMSeg EQU    8
SUMOfs EQU    6

TestSub SEGMENT
        ASSUME cs:TestSub
        PUBLIC  Add

Addrtn  PROC    FAR
        PUSH   BP          ; Save base pointer
        MOV    BP,SP      ; Get our own
        PUSH   ES

        MOV    AX,[BP+SUMSeg] ; Get segmented address to
        MOV    ES,AX      ; return summed value in
        MOV    SI,[BP+SUMOfs]

        MOV    AX,[BP+X]   ; Sum X and Y
        ADD   AX,[BP+Y]

        MOV    [ES:SI],AX  ; Save in output variable

        POP    ES          ; Restore segment register
        POP    BP          ; Restore base pointer
        RET    8           ; Return

Addrtn  ENDP
TestSub ENDS
        END
```


CALL

Statement

Calling Pascal Subprograms

The Pascal calling convention is the convention that the IBM Pascal Compiler/2 and the BASIC Compiler/2 use. You can also use the Pascal calling convention to call OS/2 environment functions directly. In this convention, the first parameter in the parameter list is the first parameter that the program puts on the stack. The Pascal convention is the default.

You can pass parameters to IBM Pascal Compiler/2 subprograms either by reference or by value. Normally, parameters are passed to IBM Pascal Compiler/2 subprograms by reference. The 2-byte offset of the argument is passed, which allows the subprogram to change the contents of the argument. Some IBM Pascal Compiler/2 subprograms may require the full 4-byte address (segment and offset) to work with the parameter. You can pass the full address to the subprogram with the `SEG` keyword.

You can also pass parameters to IBM Pascal Compiler/2 subprograms by value. If you specify the `BYVAL` keyword before an argument, its value is passed to the subprogram instead of its address and the subprogram cannot change the value of the original argument.

CALL Statement

Examples

BASIC Program:

```
' PATEST.BAS
' This example calls a Pascal subprogram.

' -- LINK PATEST+ADDSUB

' Declare the Pascal routine:
'   The first two arguments will be passed by value.
'   The result variable's segmented address will be passed.
DECLARE SUB AddInts (BYVAL X%, BYVAL Y%, SEG Result%)

OldResult% = 0
FOR I%=1 TO 10

    CALL AddInts(I%, OldResult%, NewResult%)
    PRINT "The sum of numbers from 1 to "I%" is "NewResult%
    OldResult% = NewResult%

NEXT I%
END
```

Pascal Subprogram:

```
(* ADDSUB.PAS *)
MODULE AddSub;

(* AddInts -- A subroutine that adds two integers.          *)
(* The procedure must be declared with the PUBLIC attribute. *)
PROCEDURE AddInts (X, Y: INTEGER; VARS Result: INTEGER) [PUBLIC]:

BEGIN

    Result := X + Y;

END;

END. (* End of MODULE *)
```

CALL

Statement

Calling C Subprograms

The C calling convention puts the parameters in reverse order on the stack. When the call to the subprogram or function is complete, the subprogram or function is responsible for cleaning the stack. (BASIC will handle this for you.) To pass parameters with the C convention, specify CDECL in the DECLARE statement for the subprogram.

You can pass parameters to IBM C/2 Compiler subprograms either by reference or by value.

Normally, parameters are passed to IBM C/2 Compiler subprograms by reference. The 2-byte offset of the argument is passed, which allows the subprogram to change the contents of the argument. Some IBM C/2 Compiler subprograms may require the full 4-byte address (segment and offset) to work with the parameter. You can pass the full address to the subprogram with the SEG keyword.

You can also pass parameters to IBM C/2 Compiler subprograms by value. If you specify the BYVAL keyword before an argument, its value is passed to the subprogram instead of its address and the subprogram cannot change the value of the original argument.

CALL Statement

Examples

BASIC Program:

```
'CTEST.BAS
' This example demonstrates linking BASIC with a C program.

' -- LINK CTEST+CSUM; (with appropriate C and BASIC libs)

' Declare the C routine:
'   The first two arguments will be passed by value.
'   The result variable's segmented address will be passed.
'   BASIC will use the C calling convention (because of CDECL)
DECLARE SUB Sum CDECL (BYVAL A%, BYVAL B%, SEG Result%)

OldResult% = 0
FOR I%=1 TO 10

    Sum I%, OldResult%, NewResult%
    PRINT "The sum of numbers from 1 to "I%" is "NewResult%
    OldResult% = NewResult%

NEXT I%
END
```

C Subprogram

```
/* CSUM.C */

/* Simple routine to add two numbers */

/* Routine must be declared a "far" routine */
int far sum(a, b, result)

int a, b;          /* input numbers are values */
int far *result;  /* output is a "far" pointer */

{ /* sum */

    *result = a + b;

} /* sum */
```

CALLS

Statement

Purpose

The `CALLS` statement calls and transfers program control to subprograms compiled with the IBM C/2 Compiler, the IBM Pascal Compiler/2, or assembled with the IBM Macro Assembler/2.

Format

`CALLS subname [(parameter [, parameter] ...)]`

Comments

subname is the name of the subprogram you are calling. The *subname* is limited to 40 characters. For external subprograms, LINK must recognize *subname* as a PUBLIC symbol. The procedure for declaring *subname* depends on the language in which the subprogram is written.

For Macro Assembler subprograms, *subname* must be a name declared in a PUBLIC statement.

parameter is the name of a variable that you want to pass to the subprogram.

CALLS Statement

The CALLS statement works like the CALL statement, with one difference.

The CALLS statement handles communication of parameters to the subprogram differently than the CALL statement. For each parameter in the formal parameter list, a segment and an offset for that argument are pushed onto the stack; the segment is the location of the DATA segment of the compiler and the offset is the offset into the DATA segment.

You should use CALLS for existing subprograms that expect both the segment and offset of each argument to be on the stack upon routine entry.

For related information, see the CALL Statement. See also “Modular Programming” in *IBM BASIC Compiler/2 Fundamentals*.

CALL ABSOLUTE

Statement

Purpose

The CALL ABSOLUTE statement transfers control to a machine language subroutine.

Format

CALL ABSOLUTE ([*parameter*[,*parameter*]...,] *intvar*)

Comments

parameter is the name of any argument that you want to pass to the machine language subroutine.

intvar is the name of an integer variable.

You must use the word ABSOLUTE. It is a global symbol that is the name of a library routine that transfers control to your subroutine.

The *parameters* are optional. They are the arguments that are passed to the machine language subroutine.

You must include *intvar* in the parameter list. The value of *intvar* is the starting memory location of the subroutine as an offset into the current segment of memory as defined by the last DEF SEG statement. The *intvar* is an argument to the ABSOLUTE routine, but is not passed as an argument to your machine language subroutine. If other parameters are included in the list, *intvar* must appear last. The value of *intvar* must be set to the offset value before the CALL ABSOLUTE statement is run. A DEF SEG statement *must* be run before the CALL ABSOLUTE statement is performed to ensure that the code segment is correct.

Because the subroutine is not linked to the BASIC Compiler calling module, the machine language subroutine does not have segment naming requirements as defined by CALL and CALLS.

CALL ABSOLUTE Statement

In a compiled program, you can put the routine into an integer variable array by following these steps:

1. Dimension an integer array so the number of elements in the array is the number of words, not bytes, in the subroutine.
2. Use a FOR...NEXT loop to read the hex values for the machine language code from DATA statements into the array.
3. Call VARPTR(*arrayname*) to define the offset before you perform the CALL ABSOLUTE.

See "Calling Assembler Language Subprograms" in *IBM BASIC Compiler/2 Fundamentals* for more information.

The last CALL ABSOLUTE argument (the address) is treated in the same manner as the address arguments in PEEK. That is, real numbers are changed to two-byte integers. The two-byte integers are treated as offsets from the current DEF SEG. Long (four-byte) integers are treated as segment (or selector) and offset.

For OS/2 users:

In OS/2 mode, ABSOLUTE attempts to obtain a code segment alias for the specified region. If this fails, a **Permission denied** error occurs.

Examples

This example calls a subroutine located at OFFSET% = 0 of the segment returned by VARSEG. Parameters A, B, and C are passed to the subroutine.

```
1000 DEF SEG = VARSEG(ARRAY(1)) 'set segment
1010 OFFSET% = 0 'set offset
1020 CALL ABSOLUTE (A,B,C,OFFSET%)
```

CALL INT86

Statement

Purpose

The CALL INT86 statement allows a BASIC Compiler program to perform any DOS mode software interrupt.

This statement is not available in OS/2 mode.

Format

[CALL] INT86 (*intnum%*,*inarray%*(), *outarray%*())

Comments

The interrupt is performed with the registers set to values specified in the integer array *inarray%*. The values of the registers after the interrupt are stored in the integer array *outarray%*.

CALL INT86 sets only the nonsegment registers. Use CALL INT86X to set all registers.

intnum% is the integer interrupt number. It can range from 0 through 255. See *IBM Disk Operating System Version 3.30 Reference* or the technical reference for your computer for information about interrupt numbers and the functions they perform.

inarray%() is an integer array of register values that the interrupt uses. CALL INT86 uses an 8-element array. The register values are:

<i>inarray%</i> (<i>x</i>)	AX
<i>inarray%</i> (<i>x</i> +1)	BX
<i>inarray%</i> (<i>x</i> +2)	CX
<i>inarray%</i> (<i>x</i> +3)	DX
<i>inarray%</i> (<i>x</i> +4)	BP
<i>inarray%</i> (<i>x</i> +5)	SI
<i>inarray%</i> (<i>x</i> +6)	DI
<i>inarray%</i> (<i>x</i> +7)	flags

CALL INT86 Statement

outarray%() is an integer array of register values after the interrupt. The *outarray%* has the same structure as *inarray%*, that is, *outarray%(y)* to *outarray%(y+7)*.

If the CALL INT86 proceeds without an error, *intnum%* remains unchanged and *outarray%* contains the register values after the interrupt.

If an error occurs during the CALL INT86, *intnum%* changes to -1, *outarray%* remains unchanged, and the program does not complete the CALL INT86. An error occurs if the first argument, *intnum%*, is not in the range 0 through 255.

The CALLINT86 routine alters all registers except BP and DS.

CALL INT86

Statement

Examples

The following example uses INT86 to open a file and place some text in it:

```
DIM INARY%(7),OUTARY%(7)      'define input and output arrays for INT86

INT21%=&h21                   'interrupt number is 21H

AXREG% = 0                    'define register array indices
BXREG% = 1                    'to make program easier to understand
CXREG% = 2
DXREG% = 3
BPREG% = 4
SIREG% = 5
DIREG% = 6
FLREG% = 7

INARY%(AXREG%) = &H3C00      'DOS function to create a file
INARY%(CXREG%) = 0          'DOS attribute for created file
INARY%(DXREG%) = SADD("FOO.TXT"+CHR$(0)) 'Pointer to filename string
                                'with zero byte termination
CALL INT86(INT21%,INARY%(),OUTARY%()) 'Perform the operation

INARY%(BXREG%) = OUTARY%(AXREG%) 'Move created file handle for write
INARY%(AXREG%) = &H4000      'DOS function to write to file
TEXT$ = "hello, world"+CHR$(13)+CHR$(10) 'Define text to write to file

INARY%(CXREG%) = LEN(TEXT$)  'Get length of text string
INARY%(DXREG%) = SADD(TEXT$) 'Get address of text string
CALL INT86(INT21%,INARY%(),OUTARY%()) 'Perform write operation
INARY%(AXREG%) = &H3E00      'DOS function to close a file
CALL INT86(INT21%,INARY%(),OUTARY%()) 'Perform the close
```

CALL INT86X Statement

Purpose

The CALL INT86X statement allows a BASIC Compiler program to perform any software interrupt.

This statement is not available under the OS/2 mode.

Format

[CALL] INT86X (*intnum%*,*inarray%*(), *outarray%*())

Comments

The interrupt is performed with the registers set to values specified in the integer array *inarray%*(). The values of the registers after the interrupt are stored in the integer array *outarray%*().

Use CALL INT86X to set all registers, including the **DS** and **ES** registers.

Use CALL INT86 when it is not necessary to change **DS** and **ES**.

Warning: Use CALL INT86X only when the system call requires segment register values. Altering segment registers incorrectly can cause serious problems.

intnum% is the integer interrupt number. It can range from 0 through 255. See *IBM Disk Operating System Version 3.30 Reference* or the technical reference for your computer for information about interrupt numbers and the functions they perform.

inarray%(*x*) is an integer array of register values that the interrupt uses.

CALL INT86X uses a 10-element array. The last two array elements in INT86X represent the segment register values. The register values are as follows:

inarray%(*x*) AX

CALL INT86X

Statement

<i>inarray%</i> (x+1)	BX
<i>inarray%</i> (x+2)	CX
<i>inarray%</i> (x+3)	DX
<i>inarray%</i> (x+4)	BP
<i>inarray%</i> (x+5)	SI
<i>inarray%</i> (x+6)	DI
<i>inarray%</i> (x+7)	flags
<i>inarray%</i> (x+8)	DS (If -1, then use the current runtime value of DS.)
<i>inarray%</i> (x+9)	ES (If -1, then use the current runtime value of ES.)

outarray%(*i*) is an integer array of register values after the interrupt. *outarray%* has the same structure as *inarray%*, that is, *outarray%*(*y*) to *outarray%*(*y* + 9).

If the CALL INT86X proceeds without an error, *intnum%* remains unchanged and *outarray%* contains the register values after the interrupt.

If an error occurs during the CALL INT86X, *intnum%* changes to -1, *outarray%* remains unchanged, and the program does not complete the CALL INT86X. An error occurs if the first argument, *intnum%*, is not in the range 0 through 255.

The CALL INT86X routine alters all registers except BP and DS.

Examples

See the example for the CALL INT86 statement.

CASE Statement

Purpose

The CASE statement selects statements to be executed based on the value of the expression.

Format

```
SELECT CASE expression
CASE caseitem [,caseitem]...
    statements
[CASE caseitem [,caseitem]...
    statements]
:
[CASE ELSE
    statements]
END SELECT
```

Comments

expression can be any numeric expression

caseitem can be any of the following forms:

- *is relational operator constant*

The relational operators are: "<", "<=", "=", ">=", ">", and "<>".

- *constant*

The syntax of CASE *constant* is the same as CASE IS = *constant*.

- *constant TO constant*

statements are any statements you want to execute in that case.

CASE Statement

When more than one *caseitem* exists for a single CASE statement, the *caseitems* are Ored.

The statement or statements executed have a *caseitem* condition that is satisfied by the current value of the expression.

The CASE ELSE clause is optional, but if used, it must be the last CASE within the SELECT/END SELECT block. An error occurs if no CASE block qualifies and there is no CASE ELSE clause.

The expression and all constants must be of the same type.

Examples

```
SELECT CASE index
CASE IS > 10
    PRINT "Index too high."
CASE 1 TO 10
    PRINT "Index OK."
CASE ELSE
    PRINT "Index too low."
END SELECT
```

Purpose

The CDBL function converts x to a double-precision number.

Format

$v = \text{CDBL}(x)$

Comments

x can be any numeric expression.

BASIC follows the rules for converting from one numeric precision to another, as explained in “How BASIC Converts Numbers from One Precision to Another” under “Numeric Variables” in *IBM BASIC Compiler/2 Fundamentals*. Also see the CINT function for converting numbers to integers, the CLNG function for converting numbers to long integers, and the CSNG function for converting numbers to single-precision.

Examples

The value of CDBL(A) is accurate only to the second decimal place after rounding. This is so because only two decimal places of accuracy are supplied with A.

```
110 A = 454.67
120 PRINT A;CDBL(A)
```

Results:

```
454.67 454.6699829101563
```

CHAIN Statement

Purpose

The CHAIN statement transfers control to another program.

Format

CHAIN *filespec*

Comments

filespec is a string expression for the file specification. It can contain a path and must conform to the rules outlined under “File Names” and “File Specification” in *IBM BASIC Compiler/2 Fundamentals*; otherwise, an error occurs.

The CHAIN statement performs two different ways, depending on whether you are using the runtime module or not (to keep from using the runtime module, you can compile with the /O parameter).

When you use the runtime module, chaining works as in the interpreter. Files are left open, device status is preserved, and you can use COMMON to pass parameters to the chained-to program. Both the chaining program and the chained-to program must have been compiled without the /O parameter.

In a program that does not use the runtime module, CHAIN works just like RUN *filespec*, with the exception that both the chaining program and the chained-to program must have been compiled with /O.

Note: To share variables between programs, use the COMMON statement. See the “COMMON Statement” for more information.

CHAIN Statement

Examples

This example shows how to run a second program from within your application:

```
CHAIN "A:OVERLAY2.EXE"
```

CHDIR

Command

Purpose

The CHDIR command changes the current directory.

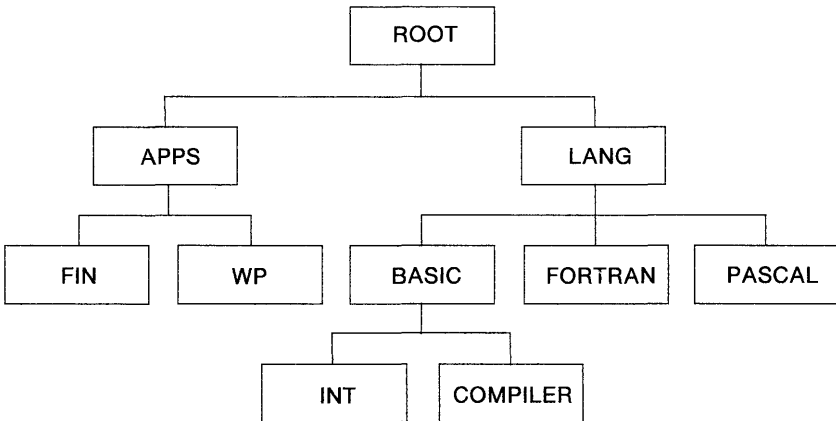
Format

CHDIR *path*

Comments

path is a string expression, not exceeding 63 characters, identifying the new directory that becomes the current directory. For more information on paths, refer to "File Specification" and "Tree-Structured Directories" in *IBM BASIC Compiler/2 Fundamentals*.

Examples



CHDIR Command

(The examples that follow refer to the tree structure shown on the previous page.)

To change to the root directory from any subdirectory, use:

```
CHDIR "\"
```

To change to the directory WP from the root directory, use:

```
CHDIR "APPS\WP"
```

To change to the directory FORTRAN from the directory LANG, use:

```
CHDIR "FORTRAN"
```

To change from the directory FIN to the directory APPS, use:

```
CHDIR ".."
```

CHR\$ Function

Purpose

The CHR\$ function converts an ASCII code to its character equivalent.

Format

$v\$ = \text{CHR}\(n)

Comments

n must be in the range 0 through 255.

The CHR\$ function returns the one-character string with ASCII code n . CHR\$ is commonly used to send a special character to the screen or printer. For instance, you might include the BEL character, CHR\$(7), which makes the speaker beep, as a preface to an error message (instead of using BEEP).

See Appendix B, "ASCII Character Codes," for more information. Also see the "ASC Function" for information on how to convert a character back to its ASCII code.

Examples

This example prints the character equivalent of ASCII code 66:

```
PRINT CHR$(66)
```

Results:

```
B
```

The next example sets function key F1 to the string "FILES," plus Enter. This is a good way to set the function keys so Enter is automatic when you press the function key.

```
KEY 1,"FILES"+CHR$(13)
```

CHR\$ Function

The following example is a program that shows all the displayable characters, along with their ASCII codes, on the screen in 80-column width:

```
110 CLS
120 FOR I=1 TO 255
130 ' ignore nondisplayable characters
140 IF (I>6 AND I<14) OR (I>27 AND I<32) THEN 1000
150 COLOR 0,7 ' black on white
160 PRINT USING "###"; I ; ' 3-digit ASCII code
170 COLOR 7,0 ' white on black
180 PRINT " "; CHR$(I); " ";
190 IF POS(0)>75 THEN PRINT ' go to next line
1000 NEXT I
```

CINT

Function

Purpose

The CINT function converts x to an integer.

Format

$v = \text{CINT}(x)$

Comments

x can be any numeric expression. If x is not in the range of -32768 through 32767, an **Overflow** error occurs.

BASIC converts x to an integer by rounding the fractional portion, following the rules for converting from one numeric precision to another, as explained in "How BASIC Converts Numbers from One Precision to Another" under "Numeric Variables" in *IBM BASIC Compiler/2 Fundamentals*.

See the FIX and INT functions, both of which also return integers. See the CLNG function for converting numbers to long integers, the CSNG function for converting numbers to single-precision, and the CDBL function for converting numbers to double-precision.

Examples

Observe, in both examples, how rounding occurs:

```
PRINT CINT(45.499)
```

Results:

45

```
PRINT CINT(-2.89)
```

Results:

-3

CIRCLE Statement

Purpose

The CIRCLE statement draws an ellipse on the screen with the center (x,y) and radius (r) .

The CIRCLE statement is valid only in graphics mode.

Format

CIRCLE [STEP](x,y), r [, [*attribute*][, [*start*][, [*end*][, [*aspect*]]]]

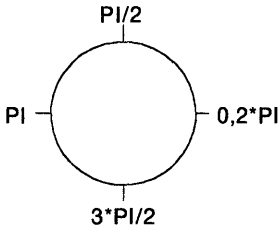
Comments

- STEP** shows that you are using relative coordinates. See “Specifying Coordinates” under “Graphics Modes” in *IBM BASIC Compiler/2 Fundamentals* for more information about STEP.
- (x,y)** are the coordinates of the center of the ellipse. You can give the coordinates in either absolute or relative form. See “Specifying Coordinates” under “Graphics Modes” in *IBM BASIC Compiler/2 Fundamentals*.
- r** is the radius (major axis) of the ellipse in points.
- attribute*** is an integer expression that specifies a color attribute. In SCREEN 1, (medium resolution), *attribute* can range from 0 through 3. In SCREEN 2, (high resolution), *attribute* can be 0 or 1.
- The default color attribute for the foreground is the maximum color attribute for that screen mode.
- The default color attribute for the background is always 0.
- start, end*** are angles, in radians, and can range from $-2*PI$ through $2*PI$, where $PI = 3.141593$.

CIRCLE

Statement

The *start* and *end* variables specify where the drawing of the ellipse begins and ends. The angles are positioned in the standard mathematical way, with 0 to the right and going counterclockwise:

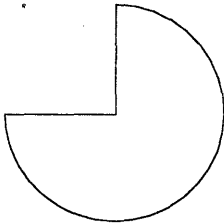


If the start or end angle is negative (-0 is not allowed), the ellipse is connected to the center point with a line, and the angles are treated as if they were positive (note that this is not the same as adding $2 \cdot \text{PI}$).

The start angle can be greater or less than the end angle. For example:

```
110 PI=3.141593
120 SCREEN 1
130 CIRCLE (160,100),60,,-PI,-PI/2
```

draws a part of a circle similar to the following:



CIRCLE Statement

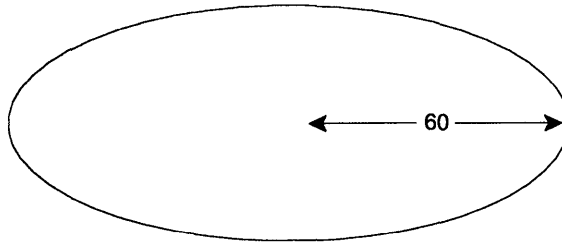
aspect is a numeric expression.

The *aspect* affects the ratio of the x-radius to the y-radius. The default for *aspect* is 5/6 in medium resolution and 5/12 in high resolution. These values give a visual circle, assuming, that the aspect ratio of the screen is the standard of 4/3.

If *aspect* is less than 1, *r* is the x-radius. That is, the radius is measured in points in the horizontal direction. If *aspect* is greater than 1, *r* is the y-radius. For example:

```
110 SCREEN 1
120 CIRCLE (160,100),60,,,5/18
```

draws an ellipse like this:



In many cases, an *aspect* of 1 results in nicer-looking circles in medium resolution. It also causes BASIC to draw the circle somewhat faster.

The last point referred to after a circle is drawn is the center of the circle.

Points that are off the screen are clipped, and no error occurs.

CIRCLE

Statement

Examples

The following example draws a face:

```
110 PI=3.141593
120 SCREEN 1 ' medium res. graphics
130 COLOR 0,1 ' black background, palette 1
140 'two circles in color 1 (cyan)
150 CIRCLE (120,50),10,1
160 CIRCLE (200,50),10,1
170 'two horizontal ellipses
180 CIRCLE (120,50),30,,,5/18
190 CIRCLE (200,50),30,,,5/18
200 'arc in color 2 (magenta)
210 CIRCLE (160,0),150,2, 1.3*PI, 1.7*PI
220 'arc, one side connected to center
230 CIRCLE (160,52),50,, 1.4*PI, -1.6*PI
```

Purpose

The CLEAR statement sets all numeric variables to 0's and all string variables to null. The options set the amount of stack space.

CLEAR is supported for the run-time module only.

Format

```
CLEAR [,n] [,s]
```

Comments

- n* is an integer expression. This parameter is ignored by the compiler. If you are compiling a program that was originally written for the interpreter, the value for *n* can be included without causing an error.
- s* is an integer expression that sets aside stack space for the BASIC compiler. The default is 768 bytes. Include *s* if you use many nested GOSUB statements or FOR...NEXT loops in your program, or if you use PAINT to do complex scenes.

CLEAR performs the following actions:

- Closes all files.
- Clears all COMMON variables.
- Resets the stack and string space.
- Releases all buffers (that is, space allocated for file buffers is returned to string space).
- Resets all variables and arrays to 0 or null.
- Resets DEF SEG to the default.
- Turns off any sound that is running and resets to Music Fore-ground.

CLEAR Command

- Resets PEN and STRIG to OFF.
- Resets the DRAW graphics parameters.
- Resets all events.

Because type definitions are determined when the program is compiled, any definitions set by DEFTYPE are not cleared.

If you use CLEAR, it should be the first statement in your program because it erases all variables.

The ERASE statement is useful to free some memory without erasing all the data in the program. It erases only specified arrays from the work area. See the "ERASE Statement."

Examples

This example clears all data from memory (without erasing the program):

```
CLEAR
```

The next example clears the data and sets the size of the stack to 2000 bytes:

```
CLEAR ,,2000
```

Purpose

The CLNG statement converts x to a long integer.

Format

$V = \text{CLNG}(X)$

Comments

x can be any numeric expression. If x is not in the range of -2147483648 through 2147483647 , an error occurs.

BASIC follows the rules for converting from one numeric precision to another, as explained in "How BASIC Converts Numbers from One Precision to Another" under "Numeric Variables" in *IBM BASIC Compiler/2 Fundamentals*. Also see the CINT function for converting numbers to integers, the CSNG function for converting numbers to single-precision, and the CDBL function for converting numbers to double-precision.

Examples

```
A = 33456.789  
PRINT A#, CLNG(A#)
```

Results:

```
33456.789      33457
```

CLOSE Statement

Purpose

The CLOSE statement terminates I/O to a device or file.

Format

CLOSE [[#] *filenum* [, [#] *filenum*],...]

Comments

filenum is the number used on the OPEN statement.

The association between a particular file or device and its file number stops when CLOSE is run. Subsequent I/O operations specifying that file number are invalid. The file or device can be opened again using the same or a different file number, or the file number can be reused to open any device or file.

A CLOSE to a file or device opened for output causes the final buffer to be written to the file or device.

A CLOSE with no file numbers specified causes all open devices and files to be closed.

A RESET, RUN, STOP, SYSTEM, CLEAR or Ctrl + Break causes all open files and devices to be automatically closed.

See also "OPEN Statement" for information about opening files.

CLOSE Statement

Examples

This example causes the files and devices associated with file numbers 1, 2, and 3 to be closed:

```
100 CLOSE #1,#2,#3
```

This example causes all open devices and files to be closed:

```
100 CLOSE
```

CLS Statement

Purpose

The CLS statement clears the screen.

Format

CLS

Comments

When the screen is in text mode, the active page is cleared to the background color. See also "COLOR Statement" and "SCREEN Statements."

When the screen is in graphics mode, the entire screen buffer is cleared to the background color.

The CLS statement also returns the cursor to the home position. In text mode, this means that the cursor is located in the upper left corner of the screen. In graphics mode, the home position is the center of the screen. This becomes the "last point referred to" for future graphics statements. In medium resolution, the center of the screen is (160,100). In high resolution, the center of the screen is (320,100).

Changing the screen mode or width by using the SCREEN or WIDTH statements also clears the screen.

When you are using the VIEW statement, CLS clears only the current viewport. To clear the entire screen, you must use VIEW to disable the active viewport and then use CLS to clear the screen.

CLS Statement

Examples

In color graphics mode, this example clears the screen to blue:

```
110 SCREEN 0,0,0
120 COLOR 10,1
130 CLS
```

COLOR

Statement

The COLOR Statement in Text Mode

Purpose

In Text mode the COLOR statement sets the colors for the foreground, background, and screen border. See “Screen Displays” in *IBM BASIC Compiler/2 Fundamentals* for an explanation of these items.

Format

COLOR [*foreground*] [, [*background*] [, *border*]]

Comments

foreground is a numeric expression in the range 0 through 31, representing the character color.

background is a numeric expression in the range 0 through 7 for the background color.

border is a numeric expression in the range 0 through 15. It is the color for the border screen.

If you have the Color/Graphics Monitor Adapter, the following colors are allowed as values for *foreground*:

0 Black	8 Gray
1 Blue	9 Light Blue
2 Green	10 Light Green
3 Cyan	11 Light Cyan
4 Red	12 Light Red
5 Magenta	13 Light Magenta
6 Brown	14 Yellow
7 White	15 High-intensity White

Colors and intensity can vary depending on your display device.

COLOR Statement

You might like to think of colors 8 to 15 as “light” or “high-intensity” values of colors 0 to 7. To obtain the “light” shade for any of the colors 0 to 7, add 8 to that color value.

You can make the characters blink by setting *foreground* equal to 16 plus the number of the desired color. That is, values from 16 to 31 cause blinking characters.

You can select only color attributes 0 through 7 for *background* in text mode. The foreground color attribute can equal the background color attribute. This makes any character displayed invisible. Changing the foreground or background color attribute makes subsequent characters visible again.

If you have the IBM Monochrome Display and Printer Adapter, you can use the following values for *foreground*:

0	Black when <i>background</i> is 0 or 7, green otherwise.
1	Underlined green.
2-7	Green.
8	Black when <i>background</i> is 0 or 7, intense green otherwise.
9	Intense underlined green.
10-15	Intense green.
16	Blinking black when <i>background</i> is 0 or 7, blinking green otherwise.
17	Underlined blinking green.
18-23	Blinking green.
24	Blinking black when <i>background</i> is 0 or 7, blinking green otherwise.
25	Intense blinking underlined green
26-31	Intense blinking green.

COLOR Statement

For *background*, you can select the following values:

0-6 Black

7 Green, if *foreground* is 0, 8, 16, or 24. Black otherwise.

Note: The color attributes:

0,7

8,7

16,7

24,7

all produce reverse video (black on green).

All other color combinations produce some form of standard video (green on black) on the IBM Monochrome Display.

Any parameter can be omitted. Omitted parameters assume the old value. If the `COLOR` statement ends in a comma (,), you get **Syntax Error**. For example:

```
COLOR 1,7,
```

is incorrect.

Any values entered outside the range 0 through 255 result in an **Illegal function call** error.

COLOR Statement

Examples

This statement sets a yellow foreground, a blue background, and a black border screen when using the Color/Graphics Monitor Adapter or equivalent:

```
COLOR 14,1,0
```

The following example can be used with either the Color/Graphics Monitor Adapter or the IBM Monochrome Display and Printer Adapter:

```
100 PRINT "Enter your ";
120 COLOR 15,0 'highlight next word
130 PRINT "password";
140 COLOR 7 'return to default (white on black)
150 PRINT " here: ";
160 COLOR 0 'invisible (black on black)
170 INPUT PASSWORD$
180 IF PASSWORD$="secret" THEN 220
190 ' blink and highlight error message
200 COLOR 31: PRINT "Wrong Password": COLOR 7
210 GOTO 110
220 COLOR 0,7 'reverse image (black on white)
230 PRINT "Program continues...";
240 COLOR 7,0 'return to default (white on black)
```

COLOR

Statement

The COLOR Statement in Graphics Mode

Purpose

In graphics mode the COLOR statement sets the colors for the background and palette in Screen mode 1.

This statement is available in Screen mode 1 only.

Format

COLOR [*background*][,*palette*]

Comments

background is an integer expression in the range 0 through 15. It specifies the background color attribute.

palette is an integer expression. It selects one of two palettes of color. The maximum color attribute in each palette is 3.

Palette is always 0 in OS/2 mode.

In screen 1, the COLOR statement sets a background color and chooses one of two palettes with four color attributes each (0-3). Color attribute 0 is always the current background. You can select one of three color attributes for the foreground color to be used with PSET, PRESET, LINE, CIRCLE, PAINT, VIEW, and DRAW.

The colors selected when you choose each palette are as follows:

Attribute	Palette 0	Palette 1
1	Green	Cyan
2	Red	Magenta
3	Brown	White

COLOR Statement

If *palette* is an even number, palette 0 is selected. This associates green, red, and brown to the color attributes 1, 2, and 3.

If *palette* is an odd number, palette 1 is selected. This associates cyan, magenta, and white to the color attributes 1, 2, and 3.

The default for palette is 0. The color selected for background can be the same as any of the palette colors. Any parameter can be omitted from the COLOR statement. Omitting parameters does not cause the current background or palette to change. Any values entered outside the range 0 through 255 cause an **Illegal function call** error. Previous values are retained.

Graphics mode can display text in any of the three colors available in the current palette. However, if you are not using a U.S. keyboard, refer to the GRAFTABL command in the *IBM Disk Operating System Version 3.30 Reference* for information regarding additional character support for the Color/Graphics Monitor Adapter and other keyboards.

Examples

This statement sets the background to light blue and selects palette 0:

```
110 SCREEN 1  
120 COLOR 9,0
```

The following example is for DOS mode only:

In the next example, the background stays light blue, and palette 1 is selected:

```
COLOR ,1
```

COM(*n*) Statement

Purpose

The COM(*N*) statement enables or disables trapping of communications activity to the specified communications adapter.

This statement is only supported under DOS 3.30 and OS/2 mode.

Format

COM(*n*) ON

COM(*n*) OFF

COM(*n*) STOP

Comments

n is the number of the communications adapter (1 or 2).

A COM(*n*) ON statement must be run to allow trapping by the ON COM(*n*) statement. If a non-0 line number is specified in the ON COM(*n*) statement, BASIC checks every time a new statement or line (depending on whether you compiled using **/V** or **/W**) is run to see if any characters have come in to the communications adapter.

If COM(*n*) is off, no event trapping takes place, and any communication activity is not remembered even if it does take place. When COM(*n*) is off, data is not lost unless the amount of data exceeds the size of the communications buffer.

Note: If your program contains any event-trapping statements, such as ON COM, you need to compile your program using the **/V** or **/W** switch.

If a COM(*n*) STOP statement has been run, no trapping can take place. However, any communications activity that does take place is remembered so that an immediate trap occurs when COM(*n*) ON is run.

COM(n) Statement

Examples

This example enables trapping of communications activity on COM1:

```
10 COM(1) ON
```

COMMAND\$

Function

Purpose

The `COMMAND$` statement returns the parameters from the command line used to call the program.

Format

`v$ = COMMAND$`

Comments

You can use this function to determine which command parameters are currently in effect. With this information you can verify, for example, the number of players selected for a game, and then branch to the appropriate section of code.

All leading blanks are removed from the command line. The following characters, used to control redirection of I/O are also removed:

```
<  
>  
>>  
|
```

Anything following these characters is also removed. All letters are converted to uppercase (capital letters).

Examples

Suppose you have written a program named `SORT` that sorts an input file alphabetically. If you call your program with the following:

```
SORT TENNIS.NAM
```

`COMMAND$` will return `TENNIS.NAM` to your program. You can then include a statement similar to the following in your `SORT` program:

```
X$ = COMMAND$  
OPEN X$ AS #1
```

COMMANDS

Function

This procedure allows you to pass the name of the file you want to sort as a command parameter, rather than as input from the user during a program run.

COMMON Statement

Purpose

The COMMON statement passes variables to a chained or called program.

Format

```
COMMON [SHARED][/blockname/]variable [(integer)] [AS type]  
[,variable[(integer)] [AS type]]...
```

Comments

SHARED allows you to share variables among all subprograms in a module.

blockname is an identifier up to 40 characters long.

variable is the name of a variable to be passed to the chained-to program. Arrays are specified by appending “()” to the array name.

integer is the number of dimensions the array has if it is a dynamic array.

type is one of the following:

- INTEGER
- LONG
- SINGLE
- DOUBLE
- STRING [* *bytecount*]
- *typename*

typename must have been defined in a previous TYPE statement.

The IBM BASIC Compiler/2 supports named common blocks, making it easier to pass information between subprograms and modules.

COMMON Statement

Note: A standard COMMON (blank COMMON) statement, becomes a named COMMON statement with the addition of the optional *blockname* parameter. The standard COMMON statement can be used to pass information between subprograms and modules, but does not allow structured development of subprogram modules and libraries.

There are two forms of array declaration in COMMON statements. If the array declaration uses the form:

```
COMMON variable()
```

the array is treated as a static array and must have been declared with integer constant subscripts in a previous DIM statement.

If the array is declared using the following form:

```
COMMON variable(integer)
```

the array is treated as a dynamic array, where *integer* is an integer constant indicating the number of dimensions. The array elements are not allocated at this time; only space for the dynamic array descriptor is reserved in the COMMON area. The array must be allocated at some point by a DIM or REDIM statement referring to the array. If the array is allocated in the chaining program, its element values are passed to the chained—to program. The presence of a subsequent DIM statement in the chained—to program produces an error. A subsequent REDIM erases the passed values.

The SHARED attribute allows the variables to be shared globally by the main program and all subprograms within a module. To declare variables as global variables, place the word “SHARED” directly after the word “COMMON.” This differs from the SHARED statement. The SHARED statement only affects variables within a *single* subprogram. For more information, see the “SHARED Statement.”

Unlike a blank COMMON statement, items in a named COMMON statement are not preserved across chaining to new programs.

COMMON Statement

If the same block name is used in more than one module, the variables in the variable list must be the same type and size and must be listed in the same order. However, the names may be different. For example:

```
A,B,C(2)
  and
E,F,G(2)
```

are correct, but

```
A,B,C(2)
  and
E,F(2),G
```

are not.

The COMMON statement must appear in a program before any executable statements. The nonexecutable statements for the IBM BASIC Compiler/2 are as follows:

```
COMMON
DATA
DECLARE
DEFtype
DIM (static arrays only)
OPTION BASE
REM
SHARED
STATIC
TYPE
all compiler metacommands.
```

All other statements are executable.

If there are any variables in the COMMON statement whose types are defined by a DEFtype statement, the DEFtype statement must precede the COMMON statement.

For static arrays, the COMMON statement must appear after the DIM statement. For dynamic arrays, the COMMON statement must appear before the DIM statement.

COMMON Statement

When you use `COMMON` to communicate with a chained-to program, you must use the run-time module (compile without the `/O` parameter). Also, both the chaining program and the chained-to program require a `COMMON` statement. The *order* of variables in the two statements must be the same. The common variables must be common to *all* programs you are chaining to. If the size of the `COMMON` in the chained-to program is smaller, the extra common variables are ignored. If the common size is larger, the additional common variables are initialized to 0 or null.

When you compile with the `/O` parameter, `COMMON` may only be used to pass variables to assembly language subprograms. To refer to variables in `COMMON`, your assembly language subroutine needs this segment definition:

```
COMMON SEGMENT COMMON 'BLANK'
```

and this group definition record:

```
DGROUP GROUP COMMON
```

A convenient way to share `COMMON` areas between programs is to place `COMMON` declarations and necessary preceding `DIM` statements in a single included file and use the `$INCLUDE` metacommand in each program. For example:

MENU.BAS

```
100 REM $INCLUDE: 'COMDEF'  
.  
.  
.  
1000 CHAIN "PROG1"
```

PROG1.BAS

```
1100 REM $INCLUDE: 'COMDEF'  
.  
.  
.  
2000 CHAIN "MENU"
```

COMDEF.BAS

COMMON Statement

```
DIM A(100),B$(200)
COMMON I,J K A()
COMMON A$,B$( ),X,Y,ZIP
```

COMDEF.BAS shows how static arrays are passed by adding () to the array name.

Examples

This example chains to program PROG3 on the disk in drive A:, and passes the static array D along with the variables A, BEE1, C, and G\$:

```
100 COMMON A,BEE1,C,D(),G$
110 CHAIN "A:PROG3"
```

This is an example that uses dynamic arrays. Dynamic arrays must be dimensioned after the COMMON statement. The chaining program in the following example contains the first allocation of array X:

```
COMMON X(2)           '2 indicates 2 dimensions
DIM X(N,M)           'first allocation
```

The chained-to program reallocates array X, erasing its original contents:

```
COMMON X(2)
REDIM X(0,P)         'reallocation
```

Purpose

The cos function returns the trigonometric cosine function.

Format

$v = \text{COS}(x)$

Comments

x is the angle whose cosine is to be calculated. The value of x must be in radians. To convert from degrees to radians, multiply the degrees by $\text{PI}/180$, where $\text{PI}=3.141593$.

Examples

This example shows that the cosine of PI radians is equal to -1 . Then it calculates the cosine of 180 degrees by first converting the degrees to radians (180 degrees is the same as PI radians).

```
10 PI=3.141593
20 PRINT COS(PI)
30 DEGREES=180
40 RADIANS=DEGREES*PI/180
50 PRINT COS(RADIANS)
```

Results:

```
-1
-1
```

CSNG Function

Purpose

The CSNG statement converts x to a single-precision number.

Format

$v = \text{CSNG}(x)$

Comments

x can be any numeric expression.

BASIC follows the rules for converting from one numeric precision to another, as explained in “How BASIC Converts Numbers from One Precision to Another” under “Numeric Variables” in *IBM BASIC Compiler/2 Fundamentals*. Also see the CINT function for converting numbers to integers, the CLNG function for converting numbers to long integers, and the CDBL function for converting numbers to double-precision.

Examples

In this example, the value of the double-precision number $A\#$ is rounded at the seventh digit and returned as CSNG($A\#$):

```
10 A# = 975.3421222#  
20 PRINT A#; CSNG(A#)
```

Results:

```
975.3421221999999 975.3421
```

Purpose

The CSRLIN statement returns the vertical coordinate of the cursor.

Format

V = CSRLIN

Comments

The CSRLIN variable returns the current line (row) position of the cursor on the active page. The active page is explained under the SCREEN statement. The value returned is in the range 1 through 25.

The POS function returns the column location of the cursor. See "POS Function."

See also LOCATE statement to see how to set the cursor line.

Examples

This example saves the cursor coordinates in the variables X and Y, then moves the cursor to line 24 to put the words "HI MOM" on that line. The cursor is then moved back to its former position.

```
10 Y = CSRLIN 'record current line
20 X = POS(0) 'record current column
30 LOCATE 24,1: PRINT "HI MOM"
40 LOCATE Y,X 'restore position
```

CVI, CVL, CVS, CVD Functions

Purpose

The CVI, CVL, CVS, and CVD functions convert string variable types from random files to numeric variable types.

Format

v = CVI(*two-byte string*)

v = CVL(*four-byte string*)

v = CVS(*four-byte string*)

v = CVD(*eight-byte string*)

Comments

string is a numeric value stored in a random file.

Numeric values read from a random file must be converted from strings into numbers. CVI converts a two-byte string to an integer. CVL converts a four-byte string to a long integer. CVS converts a four-byte string to a single-precision number. CVD converts an eight-byte string to a double-precision number.

The CVI, CVL, CVS, and CVD functions do *not* change the bytes of the actual data. They change only the way BASIC interprets those bytes.

If floating-point numbers were written to the random file by the BASIC Interpreter or by a previous version of the BASIC Compiler, you must use the CVSMBF and CVDMBF functions instead of the CVS and CVD functions. See the next section for details on these functions.

See also the MKI\$, MKL\$, MKS\$, MKD\$ functions in this book and "BASIC Disk Input and Output" in *IBM BASIC Compiler/2 Fundamentals*.

CVI, CVL, CVS, CVD Functions

Examples

This example uses a random file (#1) that has previously been opened, with fields defined as in line 100. Line 110 reads a record from the file. Line 120 uses the cvs function to interpret the first four bytes (N\$) of the record as a single-precision number. N\$ was originally a number that was written to the file using the MKS\$ function.

```
100 FIELD #1,4 AS N$, 12 AS B$  
110 GET #1  
120 Y=CVS(N$)
```

CVSMBF, CVDMBF Functions

Purpose

The CVSMBF and CVDMBF functions interpret the contents of a string argument as a number in Microsoft Binary Format and convert it to a number in IEEE format.

Format

$v = \text{CVSMBF}(\text{four-byte string})$

$v = \text{CVDMBF}(\text{eight-byte string})$

Comments

string is a numeric value stored in Microsoft Binary Format.

Numeric values read from a random file that was created with an earlier version of the BASIC Compiler or with the BASIC Interpreter must be converted to IEEE format. CVSMBF converts a four-byte string to a single-precision number. CVDMBF converts an eight-byte string to a double-precision number.

You can also use the **ICV** parameter on the IBM BASIC Compiler/2 command line to make the conversion automatic.

See also the MKSMBF\$ MKDMBF\$ functions and the CVI, CVL, CVS, and CVD functions in this book.

CVSMBF, CVDMBF Functions

Examples

This example reads a record from an old format random access file and allows new values to be entered in its fields. It uses CVSMBF, CVDMBF, MKSMBF\$ and MKDMBF\$ to convert from the old Microsoft Binary format to the current IEEE number format.

```
' Define the record structure for file record:
TYPE OldRecord
  ID   AS STRING * 10
  Cost AS STRING * 4   ' Single precision number
  Amt  AS STRING * 8   ' Double precision number
END TYPE

' Define a variable of the above structure:
DIM Buff AS OldRecord

' Open file:
OPEN "OLD.DAT" FOR RANDOM AS #1

' Get the first record:
GET #1, 1, Buff

' Decode values:
CostVal = CVSMBF(Buff.Cost) ' Single precision value
AmtVal# = CVDMBF(Buff.Amt) ' Double precision value

' Get updated values:
PRINT ""Current: "Buff.ID", "CostVal", "AmtVal#"
INPUT "New?   : ", NewID$, CostVal, AmtVal#

' Encode the new values for writing to the file:
Buff.Cost = MKSMBF$(CostVal)
Buff.Amt  = MKDMBF$(AmtVal#)
Buff.ID   = NewID$

' Write the updated record to the file:
PUT #1, 1, Buff

END
```

DATA Statement

Purpose

The DATA statement stores the numeric and string constants that are accessed by the READ statements of a program.

Format

DATA *constant*[,*constant*]...

Comments

constant can be a numeric or string constant. No expressions are allowed in the list. The numeric constants can be in any format — integer, long integer, fixed point, floating point, hex, or octal. String constants in DATA statements do not have to be enclosed by quotation marks unless the string contains commas, colons, or significant leading or trailing blanks.

DATA statements are not executable and can be placed anywhere in the program. A DATA statement can contain as many constants as will fit on a line, and any number of DATA statements can be used in a program. The information contained in the DATA statements can be thought of as one continuous list of items, regardless of how many items are on a line or where the lines are placed in the program. The READ statements get access to the DATA statements in line-number order.

The variable type (numeric or string) in the READ statement must agree with the corresponding constant in the DATA statement or an error occurs.

You can use :REM to add a remark to a DATA statement. You cannot, however, use the single quote (') to add remarks to the end of a DATA statement. If you do, the compiler treats it as part of a string.

DATA Statement

Use the RESTORE statement to reread information from any line in the list of DATA statements. See the “RESTORE Statement” in this book.

Examples

See the examples, under the “READ Statement” in this book.

DATE\$

Variable and Statement

Purpose

The DATE\$ statement sets or retrieves the date.

Format

As a variable:

$v\$ = \text{DATE\$}$

As a statement:

$\text{DATE\$} = x\$$

Comments

For the variable ($v\$ = \text{DATE\$}$):

A 10-character string in the form *mm-dd-yyyy* is returned. Here, *mm* represents two digits for the month, *dd* is the day of the month (also two digits), and *yyyy* is the year. The date can be set using the operating system DATE command before running your application.

For the statement ($\text{DATE\$} = x\$$):

The $x\$$ is a string expression used to set the current date. You can enter $x\$$ in any one of the following forms:

mm-dd-yy
mm/dd/yy
mm-dd-yyyy
mm/dd/yyyy

The year must be in the range 1980 through 2099 in DOS, 1980 through 2079 in OS/2. If you use only one digit for the month or day, a 0 (zero) is assumed in front of it. If you enter only one digit for the year, a 0 is

DATE\$

Variable and Statement

appended to make it two digits. If you enter only two digits for the year, the year is assumed to be 19yy.

Examples

In this example, we set the date to August 29, 1984. Notice how, when we read the date back using the DATE\$ function, a 0 is included in front of the month to make it two digits, and the year becomes 1984. Also, the month, day, and year are separated by hyphens even though we enter them as slashes.

```
10 DATE$="8/29/84"  
20 PRINT DATE$
```

Results:

```
08-29-1984
```

DECLARE Statement

Purpose

The `DECLARE` statement identifies external procedures and procedures that a program calls before it defines.

Format

```
DECLARE SUB | FUNCTION name [CDECL] [ALIAS "aliasname"] [parameters]
```

Comments

name is the name of the procedure. If the procedure is external and its name is not a proper BASIC identifier, you can give it a nickname here to use in your program and specify its real name for *aliasname*.

CDECL declares that the procedure expects parameters to be passed in the same order as the C language uses; that is, the first parameter is pushed on the stack last. It also means that BASIC must clean the parameters from the stack for you after the procedure returns.

If CDECL is used without ALIAS, BASIC adds an underscore to the beginning of *name*. If CDECL, is used with ALIAS, BASIC does not add an underscore to the beginning of *aliasname* unless it is specified explicitly as part of *aliasname*.

aliasname is the real name of the procedure. You don't have to specify an *aliasname* unless the procedure is external and its name is different from *name*.

parameters is a list of the parameters that the calling program will pass to the subprogram or function.

When you are declaring a procedure that was compiled with the IBM C/2 Compiler, the IBM Pascal

DECLARE Statement

Compiler/2, or assembled with the IBM Macro Assembler/2, *parameters* can be of the form:

(([[BYVAL | SEG]*parameter* [(*integer*) [AS *type*],]...])

BYVAL indicates that the procedure expects the actual value of the parameter (rather than the address of the value). When the procedure is called, BASIC attempts to convert BYVAL parameters to the type you specify in the DECLARE statement. If BASIC cannot convert the parameter, an error occurs.

SEG indicates that the procedure expects the far address of the value of the parameter. Do not use SEG or BYVAL to declare BASIC procedures. Do not use SEG to pass array descriptors. They are private and known only to BASIC.

The *type* is one of the following:

- INTEGER
- LONG
- SINGLE
- DOUBLE
- STRING
- *typename*

typename must have been defined in a previous TYPE statement.

- ANY

ANY can be used only with external procedures.

Note: The ANY keyword allows you to specify a parameter without having the compiler check its type. It is primarily intended for allowing variables of different user-defined types to be passed to the same external procedure.

DECLARE Statement

When you specify parameters in the `DECLARE` statement, BASIC checks the type and count of the parameters against the actual parameters in the `SUB` or `FUNCTION` statement. If parentheses are not present in the `DECLARE` statement, BASIC does not check the type and count of the parameters.

For all `BYVAL` parameters, BASIC changes the actual parameters to the `DECLARED` type before the procedure is invoked, if possible. If BASIC cannot change the type (for example, changing a string to single-precision), an error occurs. If BASIC finds a type mismatch for a parameter in a `SUB` or `FUNCTION` statement that is not defined as `BYVAL`, BASIC returns an error.

DECLARE Statement

Examples

This example uses DECLARE to allow a forward reference to a function:

```
' DECLARE function defined below so it can be used before it is defined:
DECLARE FUNCTION NumVowels (Txt$)
```

```
CLS
PRINT "Type some text then press <ENTER>:"
LINE INPUT T$

PRINT
PRINT "The text contains "; NumVowels(T$); " Vowels."
END
```

```
' Now, define the function
FUNCTION NumVowels (T$) STATIC

    Count = 0
    FOR I=1 TO LEN(T$)

        ' Extract a character:
        Char$ = MID$(T$, I, 1)

        ' If char is in the set of vowels, count it:
        IF INSTR("aeiou", LCASE$(Char$)) <> 0 THEN
            Count = Count + 1
        END IF

    NEXT I

    NumVowels = Count

END FUNCTION
```

See also "CALL Statement" for examples of using DECLARE to identify external procedures.

DEF FN and END DEF and EXIT DEF Statements

Purpose

The DEF FN and END DEF and EXIT DEF statements define and name a function that you write.

Format

Single-Line Format:

```
DEF FNname[(parameter [AS type] [,parameter[AS type]]...)] =  
expression
```

Multi-Line Format:

```
DEF FNname[(parameter [AS type] [,parameter[AS type]]...)]  
  statements  
  [FNname = expression]  
  [EXIT DEF]  
  statements  
  FNname = expression  
END DEF
```

DEF FN and END DEF and EXIT DEF Statements

Comments

- name* is a correct variable name. This name, preceded by FN, becomes the name of the function. Use a type-declaration character on the end of *name* to specify the type of value the function will return.
- parameter* is the name of a simple variable.
- type* is one of the following:
- INTEGER
 - LONG
 - SINGLE
 - DOUBLE
 - STRING
 - *typename*
- typename* must have been defined in a previous TYPE statement.
- expression* defines the returned value of the function. The type of the *expression* (numeric or string) must match the type declared by *name*.
- statements* are the BASIC statements to be performed when the function is called.

DEF FN and END DEF and EXIT DEF Statements

The function definition can occupy one line, or as many program lines as required. If your function definition fits on a single program line, only the DEF FN statement is needed.

Parameters that appear in the function definition serve only to define the function; they do not affect program variables that have the same name. A variable name used in the *expression* does not have to appear in the list of parameters. If it does, the value of the corresponding argument supplied when the function is called, is used. Otherwise, the current value of the variable is used. If you want to define a function that uses local variables, see the information on the FUNCTION statement.

When a DEF FN function is called, BASIC converts the values of the arguments to those specified by the parameters in the DEF FN statement. If BASIC cannot convert the type (for example, changing a string to single-precision), an error is generated by the compiler.

DEF FN and END DEF and EXIT DEF Statements

The function type determines whether the function returns a numeric or string value. The type of the function is declared by *name* in the same way as variables are declared. See *IBM BASIC Compiler/2 Fundamentals* for more information. If the type of *expression* (string or numeric) does not match the function type, an error occurs; otherwise, the value of the expression is converted to the precision specified by *name* before it is returned to the calling statement.

You must define a DEF FN function before you refer to it. If you do not, BASIC returns an error. You can define a DEF FN function only once. An attempt to redefine a DEF FN function produces an error.

DEF FN functions cannot appear inside DEF/END DEF, IF/THEN/ELSE, FOR/NEXT, SUB/END SUB, or WHILE/WEND blocks.

DEF FN and END DEF and EXIT DEF Statements

Multi-Line Functions

If your function definition occupies more than one program line, you must begin the definition with a DEF FN statement and end the definition with an END DEF statement. The EXIT DEF statement is available for branching out of the defined function.

Results must be assigned to the multi-line function name prior to leaving or ending the function; otherwise, the results are lost.

Because multi-line functions are very powerful, they must be used with care. Some things to be aware of are:

- The RETURN statement is *not* equivalent to the END DEF statement, or the EXIT DEF statement. Using a RETURN statement in a multi-line function can cause unpredictable results.
- If you are not careful when constructing multi-line functions, you may get unexpected side effects. Most of these occur because, as an optimizing compiler, the IBM BASIC Compiler/2 may rearrange arithmetic expressions for greater efficiency. Make use of temporary variables and parentheses to ensure that expressions are evaluated in the intended order.

DEF FN and END DEF and EXIT DEF Statements

Examples

In this example, line 20 defines the function FNAREA, which calculates the area of a circle with radius R. The function is called in line 40.

```
10 PI=3.141593
20 DEF FNAREA(R)=PI*R^2
30 INPUT "Radius? ",RADIUS
40 PRINT "Area is " FNAREA(RADIUS)
```

Results:

```
Radius? 2
Area is 12.56637
```

The following example contains a function definition that converts an angle measure in degrees, minutes, and seconds to an angle measure in radians. (An angle must be given in radians for the trigonometric functions of BASIC to return a meaningful answer.)

```
DEF FNDEGRAD(D,M,S)
  STATIC PI
  PI = 3.14159263
  D = D + M/60 + S/3600
  FNDEGRAD = D * (PI/180)
END DEF

PRINT "Enter the angle (degrees, minutes, seconds). "
PRINT "Enter 0 for degrees to end." : PRINT
NEWVAL:
  INPUT ;"SIN(",DEG
  IF DEG = 0 THEN PRINT ")" : END
  PRINT CHR$(248) " ";
  INPUT ;"',MIN : PRINT "' ";
  INPUT ;"',SEC : PRINT CHR$(34);
  RAD = FNDEGRAD(DEG,MIN,SEC)
  PRINT ") =" SIN(RAD)
  GOTO NEWVAL
```

Results:

```
Enter the angle (degrees, minutes, seconds).
Enter 0 for degrees to end.
```

```
SIN(45° 10' 10") = .7091949
SIN(45° 10' 20") = .7092291
SIN(0)
```

DEF SEG Statement

Purpose

The DEF SEG statement defines the current segment of memory. A subsequent BLOAD, BSAVE, CALL ABSOLUTE, PEEK, or POKE definition specifies the offset into this segment.

Format

DEF SEG [= *segment*]

Comments

The segment is a numeric expression in the range 0 through 65535.

The initial setting for the segment when your application begins is the data segment (DS) of the compiler. It is the beginning of your workspace in memory.

Note: DEF and SEG must be separated by a space; otherwise, BASIC interprets the statement **DEFSEG = 100** to mean “Assign the value 100 to the variable DEFSEG.”

Any value entered outside the range indicated results in an **Illegal function call** error. The previous value is retained.

See also “Calling Assembly Language Subprograms” in *IBM BASIC Compiler/2 Fundamentals* for more information on using DEF SEG.

Note to OS/2 users:

In the OS/2 mode, DEF SEG treats the segment as a selector. Illegal memory references may cause exceptions or return a **Permission denied** error.

DEF SEG Statement

Examples

The first example restores a segment to BASIC data segment:

```
DEF SEG ' restore segment to the BASIC data segment
```

In the second example, the screen buffer for the Color/Graphics Monitor adapter is at segment B800H, offset 0. Because segments are specified on 16-byte boundaries, the last hex digit can be dropped on the DEF SEG specification. This example is for DOS mode only.

```
DEF SEG=&HB800
```

DEFTYPE Statement

Purpose

The DEFTYPE statement declares variable types as integer, long integer, single-precision, double-precision, or string.

Format

DEFTYPE *letter*[-*letter*] [,*letter* [-*letter*]]...

Comments

type is one of the following:

- INT
- LNG
- SNG
- DBL
- STR

letter is a letter of the alphabet (A – Z).

A DEFTYPE statement declares that the variable names beginning with the letter or letters specified will be that type of variable. However, a type-declaration character (% , & , ! , # , or \$) takes precedence over a DEFTYPE statement in the typing of a variable. See “How to Declare Variable Types” in *IBM BASIC Compiler/2 Fundamentals*.

If no type-declaration statements are encountered, the compiler assumes that all variables without declaration characters are single-precision variables.

A DEFTYPE statement takes effect as soon as it is encountered in your program *during compilation*. Once the type has been defined for the listed variables, that type remains in effect either until the end of the program or until another DEFTYPE statement changes the type of the variable. Unlike the interpreter, the compiler cannot branch around the DEFTYPE statement (or any statement) by using a GOTO.

DEFtype Statement

Examples

In this example, line 10 declares that all variables beginning with the letter L, M, N, O, or P are double-precision variables.

Line 20 causes all variables beginning with the letter A to be string variables.

Line 30 declares that all variables beginning with the letter X, D, E, F, G, or H are integer variables.

```
10 DEFDBL L-P
20 DEFSTR A
30 DEFINT X,D-H
40 ORDER = 1#/3: PRINT ORDER
50 ANIMAL = "CAT": PRINT ANIMAL
60 X=10/3: PRINT X
```

Results:

```
.33333333333333333333333333333333
CAT
3
```

DIM

Statement

Purpose

The DIM statement specifies the maximum values for array variable subscripts and allocates memory accordingly. It also assigns the scope of a simple variable or array variable within a module.

Format

```
DIM [SHARED]variable[(subscripts)] [AS type] [,variable  
[(subscripts)] [AS type] ]...
```

Comments

- SHARED* allows you to share simple variables and arrays among all subprograms in a module.
- variable* is the name of a simple variable or array and can be up to 40 characters in length.
- subscripts* define the dimensions of the array. The *subscripts* can be in two forms:
- (*exp*[,*exp*]...), where *exp* is a numeric expression that defines the upper bound of the dimension. The lower bound is implicitly defined by the OPTION BASE statement.
 - (*min* TO *max*[,*min* TO *max*]...), where *min* and *max* are numeric expressions that you use to explicitly define the upper and lower bounds of each dimension in the array.
- type* is one of the following:
- INTEGER
 - LONG
 - SINGLE
 - DOUBLE
 - STRING [** bytcount*]

DIM Statement

- *typename*
typename must have been defined in a previous TYPE statement.

The SHARED attribute allows variables to be shared globally by the main program and all subprograms within a module. To declare all variables as global variables, place the word “SHARED” directly after the word “DIM”. This differs from the SHARED statement. The SHARED statement only affects variables within a **single** subprogram. For more information, see “SHARED Statement.” See also, “STATIC Statement” for an example.

Variables that you define with the AS clause are not affected by the DEF type statement. The type of the variable is the type defined in the DIM statement. Do not use the type-declaration characters (% , & , ! , # , or \$) with variables that have an AS clause.

You can use the AS clause in the following statements only:

```
COMMON  
DECLARE  
DEF FN  
DIM  
FUNCTION  
REDIM  
SHARED  
STATIC  
SUB  
TYPE
```

The first reference to a variable must have an AS clause if any reference has an AS clause. After the first reference, any reference to the variable in any of the statements listed above may have an AS clause as long as the type is the same as in the first AS clause. Types are the same only when the *typename* is the same. Fixed-length strings must also be the same length.

The DIM statement sets all the elements of the specified numeric arrays to an initial value of 0. String array elements, except fixed-

DIM

Statement

length strings defined with the AS clause, are all variable-length and assigned an initial null value (0-length).

If an array variable name is used without a DIM statement, the maximum value of its subscript is assumed to be 10. If a subscript is greater than the maximum specified, an error occurs.

You can specify the range of subscripts in two ways. The first way is to specify the range of each subscript with a single integer, variable, or expression. In this case, the maximum value of the subscript is the number you specify. BASIC assumes the minimum value of the subscript is 0, unless you use the OPTION BASE statement. See "OPTION BASE Statement."

For example, you could dimension an array with the following statement:

```
DIM A(4,I+2)
```

This statement defines a two-dimensional array. The first dimension is five elements long (0 through 4); the second dimension depends on the value of I. If you had previously assigned I the value of 1, the preceding example would be equivalent to:

```
DIM A(4,3)
```

and the valid elements in array A would be:

```
A(0,0) A(0,1)  A(0,2)  A(0,3)
A(1,0) A(1,1)  A(1,2)  A(1,3)
A(2,0) A(2,1)  A(2,2)  A(2,3)
A(3,0) A(3,1)  A(3,2)  A(3,3)
A(4,0) A(4,1)  A(4,2)  A(4,3)
```

The second way to specify the range of subscripts is to specify explicitly both the minimum and maximum values for the subscripts. For example, you could dimension an array with the following statement:

```
DIM B(-2 TO 2,10 TO 13)
```

DIM Statement

Like the previous example, this statement also defines a two-dimensional array. However, the valid elements in array B are:

```
B(-2,10)  B(-2,11)  B(-2,12)  B(-2,13)
B(-1,10)  B(-1,11)  B(-1,12)  B(-1,13)
B(0,10)   B(0,11)   B(0,12)   B(0,13)
B(1,10)   B(1,11)   B(1,12)   B(1,13)
B(2,10)   B(2,11)   B(2,12)   B(2,13)
```

The maximum number of dimensions for an array is 60. The valid range for subscripts is -32768 through 32767. The maximum subscript for an array varies with the type of data stored in the array.

If you dimension an array with an integer constant and `$DYNAMIC` is not in effect, BASIC allocates space for the array statically. You cannot change the dimensions of a static array. For example, the statement

```
DIM S(5,30,9)
```

defines a three-dimensional static array. If you try to dimension a static array more than once, an error occurs.

If you dimension an array with a variable or an expression, BASIC allocates space for the array dynamically. You can change the dimensions of the array at any time in the program with the `REDIM` statement. For example, the statement

```
DIM D(J,J+4,7)
```

defines a three-dimensional dynamic array.

For more information on redimensioning arrays, see the “ERASE Statement,” the “REDIM Statement,” the “\$DYNAMIC Metacommand,” and the “\$STATIC Metacommand.” See also “Data Types” in *IBM BASIC Compiler/2 Fundamentals*.

Note: You must compile with the `/D` parameter to enable the compiler to check array bounds.

DIM

Statement

Examples

The following example performs the multiplication of two arithmetic arrays and assigns the product to a third array. The accompanying figure shows this process.

```

130 DIM A(4,2), B(2,4), C(4,4)
140 FOR I = 1 TO 4
150   FOR K = 1 TO 2
160     READ A(I,K)
170   NEXT K
180 NEXT I
190 FOR K = 1 TO 2
200   FOR J = 1 TO 4
210     READ B(K,J)
220   NEXT J
230 NEXT K
240 FOR I = 1 TO 4
250   FOR J = 1 TO 4
260     FOR K = 1 TO 2
270       C(I,J) = A(I,K) * B(K,J) + C(I,J)
280     NEXT K
290   NEXT J
300 NEXT I
310 CLS
320 FOR I = 1 TO 4
330   LOCATE I,1
340   FOR J = 1 TO 4
350     PRINT C(I,J);
360   NEXT J
370 NEXT I
380 END
390 REM MATRIX A
400 DATA 8,3,4,2,5,6,10,12
410 REM MATRIX B
420 DATA 9,11,13,15,5,6,4,8

```

A			B					C			
8	3	x	9	11	13	15	=	87	106	116	144
4	2		5	6	4	8		46	56	60	76
5	6							75	91	89	123
10	12							150	182	178	246

DIM Statement

The following example shows how to use the DIM statement to declare variables:

```
TYPE MYTYPE
A AS STRING * 10 '10-character fixed-length string
B AS INTEGER
END TYPE
DIM X AS INTEGER, Y(100) AS MYTYPE
:
Y(10).A="This is a test" 'assign a 14-character string
PRINT "!"+Y(10).A+"!" 'print the string
```

Results:

```
!This is a !
```

Note that the string was cut off to 10 characters when it was assigned to the 10-character fixed-length string field of the variable of type MYTYPE.

DO Statement

Purpose

The DO statement repeats a series of statements as long as or until a given condition is true.

Format

First form:

```
DO [WHILE|UNTIL expression]  
  statements  
  [EXIT DO]  
  statements  
LOOP
```

Second form:

```
DO  
  statements  
  [EXIT DO]  
  statements  
LOOP WHILE|UNTIL expression
```

Comments

statements is any statement or statements that you want BASIC to repeat.

expression is a numeric expression that tells BASIC how long to repeat the loop.

DO Statement

BASIC runs the program lines following the DO statement until it encounters the LOOP statement. Then BASIC evaluates the expression in the WHILE or UNTIL clause.

If the expression is part of a WHILE clause and the expression is true, BASIC branches back to the statement following the DO statement and continues the process. Otherwise, BASIC exits the loop and continues processing with the statement after the LOOP statement.

For example, in the following loop:

```
DO WHILE X<5
PRINT X
X=X+1
LOOP
END
```

BASIC repeats the loop until X is equal to 5 (when the expression $X < 5$ becomes false). Then BASIC skips to the END statement.

DO Statement

For example, in the following loop:

```
DO WHILE X<5
PRINT X
X=X+1
LOOP
END
```

BASIC repeats the loop until X is equal to 5 (when the expression $X < 5$ becomes false). Then BASIC skips to the END statement.

If the expression is part of UNTIL clause and the expression is false, BASIC branches back to the statement following the DO statement and continues the process. Otherwise, BASIC exits the loop and continues processing with the statement after the LOOP statement.

For example, in the following loop:

```
DO UNTIL X>5
PRINT X
X=X+1
LOOP
END
```

BASIC repeats the loop until X is equal to 6 (when the expression $X > 5$ becomes true). Then BASIC skips to the END statement.

If you do not use a WHILE clause or an UNTIL clause with a DO or LOOP statement, you must provide the program with some means of escaping the loop, or the loop will run indefinitely. You can escape a loop by using the EXIT DO statement, as the following example shows:

```
DO
INPUT X
IF X=99 THEN EXIT DO
PRINT X "SQUARED IS " X^2
LOOP
END
```

DO Statement

You can also escape a loop by explicitly branching outside the loop by using the GOTO statement, as the following example shows:

```
10 DO
20 INPUT X
30 IF X>99 GOTO 70
40 IF X=99 GOTO 80
50 PRINT X "SQUARED IS " X^2
60 LOOP
70 PRINT "Your input is too high."
80 END
```

You can nest DO loops; that is, you can place one DO loop inside another DO loop. You must nest DO loops physically as well as logically. The LOOP statement for the inside DO loop must appear before the LOOP statement for the outside DO loop.

DRAW

Statement

Purpose

The DRAW statement draws an object as specified by *string*.

Graphics mode only.

Format

DRAW *string*

Comments

The DRAW statement draws objects using a *graphics definition language*. The language commands are contained in the string expression *string*. The string defines an object, which is drawn at run time. When a movement command is given, a line is drawn from the last point referred to.

In the following movement commands, *n* indicates the distance to move. The *n* can be any integer. The number of points moved is *n* times the scaling factor (set by the **S** command). The movement commands are detailed here.

- U** *n* Move up.
- D** *n* Move down.
- L** *n* Move left.
- R** *n* Move right.
- E** *n* Move diagonally up and right.
- F** *n* Move diagonally down and right.
- G** *n* Move diagonally down and left.
- H** *n* Move diagonally up and left.

DRAW Statement

M *x,y* Move absolute or relative. If *x* has a plus sign (+) or a minus sign (−) in front of it, it is relative; Otherwise, it is absolute.

The following two prefix commands can precede any of these movement commands:

B Move, but do not plot any points.

N Move, but return to the original position when finished.

The following commands are also available:

A *n* Set angle *n*. The value of *n* can range from 0 through 3, where 0 is 0 degrees, 1 is 90, 2 is 180, and 3 is 270. Figures rotated 90 or 270 degrees are scaled so they appear the same size with 0 or 180 degrees on a display screen with standard aspect ratio 4/3.

C *n* Set color *n*. *n* is an integer that specifies a color attribute. In SCREEN 1 (medium resolution), *n* can range from 0 through 3. In SCREEN 2 (high resolution), *n* can be 0 or 1.

The default color attribute for the foreground is the maximum color attribute for that screen mode.

The default color attribute for the background is always 0.

P *paint,boundary*

Set figure color to *paint* and border color to *boundary*. The *paint* parameter is an integer expression. You select this attribute from the attribute range for the current screen mode.

The *boundary* parameter is the border color attribute of the figure to be filled in. You select this attribute from the attribute range for the current screen mode. In SCREEN 1, (medium resolution), *attribute* can range from 0 through 3. In SCREEN 2, (high resolution), *attribute* can be 0 or 1.

The default color attribute for the foreground is the maximum color attribute for that screen mode. For example, in SCREEN 5 the default color attribute is 15.

DRAW

Statement

The default color attribute for the background is always 0.

S *n* Set scale factor. The value of *n* can range from 1 through 255. The scale factor is *n* divided by 4. For example, if *n* = 1, then the scale factor is 1/4. The scale factor multiplied by the distances given with the U, D, L, R, E, F, G, H, and relative M commands gives the actual distance moved. The default value is 4, so the scale factor is 1.

TA *n* Turn angle *n*. The value of *n* can range from -360 through +360. If *n* is positive (+), the angle turns counterclockwise. If *n* is negative (-), the angle turns clockwise. Values entered that are outside of the range -360 through +360 cause an **illegal function call** error.

X *variable*;

Run substring. This allows you to run a second string from within a string.

Use the VARPTR\$ format for this command, as follows:

```
"X" + VARPTR$(variable)
```

Unlike the interpreter, the compiler does not allow trailing semicolons in the command string.

There are two ways you can specify variables in a DRAW string for the compiler:

- Use the "X" variable form, concatenated (" + ") with the VARPTR\$ of the variable itself.
- Use the DRAW macro, followed by an equal sign (=) and concatenated (" + ") with the VARPTR\$ of the variable itself.

The following examples demonstrate both methods:

```
DRAW "X"+VARPTR$(A$)  
DRAW "S="+VARPTR$(SC)
```

The **X** command can be a very useful part of DRAW. It allows you to define segments of a picture in different **X** variables and to combine these **X** variables into a single DRAW statement. For example, if you are creating a scene of a house with a chimney and a tree, each of

DRAW Statement

these objects can be defined in an **X** variable so your DRAW statement can look like this:

```
DRAW "X"+VARPTR$(HOUSE$)+"X"+VARPTR$(CHIM$)+"X"+VARPTR$(TREES$)
```

The aspect ratio of your screen determines the spacing of the horizontal, vertical, and diagonal points. The DRAW statement does not take into account the aspect ratio of the current screen mode; that is, DRAW "R50 U50" plots exactly 50 points to the right and then 50 up, but the two lines will not appear to be equal in length.

The aspect ratio is used to correct the shape of objects drawn on a nonlinear surface. The idea is to be able to draw a square, for example, that indeed looks square.

If there are 640 by 640 dots on a screen evenly spaced along the x- and y- axes, the aspect ratio is "1 to 1" or 1/1; this is an ideal surface. If you run the statement:

```
DRAW "R100 D100 L100 U100"
```

the box appears square.

The compiler, however, only supports two screen resolutions, each with its own aspect ratio:

Resolution	Aspect ratio
Medium resolution	320 by 200 dots—5/6
High resolution	640 by 200 dots—5/12

To draw a box that appears square in any resolution, scale the y-axis by the corresponding aspect ratio; or scale the x-axis by 1/aspect ratio.

For example, to draw a square box 100 dots high, scale the x-axis as follows:

```
10 '100*6/5 is 120
20 DRAW "U100 R120 D100 L120"
```


DRAW

Statement

Examples

To draw a box using variables:

```
10 SCREEN 1
20 A=20
30 DRAW "U="+VARPTR$(A)+"R="+VARPTR$(A)+_
  "D="+VARPTR$(A)+"L="+VARPTR$(A)
40 ' DUMMY TIMING LOOP
50 FOR I=1 TO 10000: NEXT I
```

To draw a box and paint the interior:

```
5 SCREEN 1
10 DRAW "U50R50D50L50" 'Draw a box
20 DRAW "BE10" 'Move up and right into box
30 DRAW "P1,3" 'Paint interior
40 ' DUMMY TIMING LOOP
50 FOR I=1 TO 10000: NEXT I
```

To draw a triangle:

```
10 SCREEN 1
20 DRAW "E15 F15 L30"
30 ' DUMMY TIMING LOOP
40 FOR I=1 TO 10000: NEXT I
```

To create a "shooting star":

```
10 SCREEN 1,0: COLOR 0,0: CLS
20 DRAW "BM300,25" ' initial point
30 STAR$= "M+7,17 M-17,-12 M+20,0 M-17,12 M+7,-17"
40 FOR SCALE=1 TO 40 STEP 2
50 DRAW "C1;S="+VARPTR$(SCALE)+"BM-2,0;X"+VARPTR$(STAR$)
60 NEXT
```

To draw some spokes:

```
10 SCREEN 1,0:CLS
20 FOR D=0 TO 360 STEP 10
30 DRAW "TA="+VARPTR$(D)+"NU50"
40 NEXT D
```

Purpose

The END statement ends the program, closes all files, and returns to the operating system.

Format

END

Comments

END statements can be placed anywhere in the program to end the program.

An END statement at the end of a program is optional. The program returns to the operating system after an END statement is run and resets the screen to the initial screen mode.

Examples

This example ends the program if K is greater than 1000; otherwise, the program branches to the label "START."

```
100 IF K>1000 THEN END ELSE GOTO START
```

ENVIRON

Statement

Purpose

The ENVIRON statement modifies parameters in the operating system environment table when running BASIC programs.

Format

ENVIRON "*parm*[=] [*text*] [;]"

Comments

parm is the name of the parameter, such as "PATH."

text is a string expression that defines the new parameter.

The *parm* must be separated from *text* by an equal sign or a blank. ENVIRON takes everything left of the first blank or equal sign as *parm*. The first "nonblank, nonequal" character after *parm* is taken as the beginning of *text*.

If *text* is a null string or consists only of ";" (a single semicolon), such as:

```
"PATH=;"
```

the parameter is removed from the environment table and the table is compressed.

If *parm* does not exist, the new parameter is added at the end of the environment table.

If *parm* exists, it is deleted, the environment table is compressed, and *parm* is added at the end.

Note: When your compiled program is called, the size of its environment table is the current size of the operating system environment table (rounded up to the next 16-byte paragraph boundary). Your program cannot expand its environment table. If you wish to add ele-

ENVIRON Statement

ments to the environment table, you must expand the table from the operating system to the size your application needs before calling your BASIC program.

Use ENVIRON to pass configuration parameters, such as a path specification, to a child process called using SHELL or to pass configuration parameters to your application from the operating system environment.

Note: For related information, see also "ENVIRON\$ Function" and "SHELL Statement" in this book. Also the SET command in *IBM Disk Operating System Version 3.30 Reference* and the EXEC function call in *IBM Disk Operating System Version 3.30 Technical Reference*.

If you are using OS/2, refer to *IBM Operating System/2 Programmer's Guide* and *IBM Operating System/2 Technical Reference*.

Examples

You can create a default PATH to the root directory on drive A: with the following statement:

```
ENVIRON "PATH=A:\"
```

You can call the operating system from your BASIC program using the SHELL statement and issue any valid operating system command. If a disk file (.CMD, .COM, .EXE, or .BAT) is needed to run the command, the operating system now automatically searches for it in the root directory on drive A: if it is not on the current drive or directory.

You can add a new parameter to the environment table:

```
ENVIRON "HELP=C:\HELP" 'defines file parameter called "HELP"  
CHDIR ENVIRON$ ("HELP") 'changes dir to "C:\HELP"
```

You can delete this parameter in the table by:

```
ENVIRON "HELP="; 'deletes parameter "HELP" from table
```

ENVIRON\$

Function

Purpose

The ENVIRON\$ statement retrieves the specified string from the operating system environment table for a BASIC program.

Format

$v\$ = \text{ENVIRON\$} (\text{parm\$})$

or

$v\$ = \text{ENVIRON\$} (n)$

Comments

parm\$ is a string containing the parameter to be retrieved.

n is an integer expression returning a value in the range 1 through 255.

If a string argument is used, ENVIRON\$ returns from the environment table a string containing the text that follows *parm\$*. If *parm\$* is not found or no text follows the equal sign, a null string is returned.

If a numeric argument is used, ENVIRON\$ returns a string containing the *nth parm\$* from the environment table, along with the *parm\$=* text. If there is no *nth parm*, a null string is returned.

ENVIRON\$ distinguishes between uppercase letters and lowercase letters. If you add to the table in this format:

```
ENVIRON "LOAD=high"
```

and want to check to see if the operation was successful, you can use the ENVIRON\$ function like this:

```
PRINT ENVIRON$ ("LOAD")
```

ENVIRON\$ Function

But if you run:

```
PRINT ENVIRON$ ("load")
```

ENVIRON\$ returns a null string because "load" is not in the table; however, "LOAD" *is* in the table.

Note: All parameters entered into the environment are converted to uppercase by the operating system.

See also "ENVIRON Statement" and "SHELL Command."

Examples

Note: This example is for DOS mode.

When the operating system loads initially, it sets a parameter called "COMSPEC" that tells DOS where to locate the COMMAND.COM file, and it sets up a null path. To observe the contents of the environment table at startup time, run the following from your program:

```
PRINT ENVIRON$ (1)  
PRINT ENVIRON$ (2)
```

Results:

```
PATH=  
COMSPEC=A:\COMMAND.COM
```

If you run:

```
PRINT ENVIRON$ ("COMSPEC")
```

the response from the computer is:

```
A:\COMMAND.COM
```

Note: If you booted from a fixed disk, the previous example would display **C:** instead of **A:** for the drive specification.

ENVIRON\$

Function

The following program saves the environment table of the compiler in an array so that it can be modified for a child process. After the child process is completed, the environment is restored.

```
10 DIM TABLE$(10) 'assume no more than 10 parms
20 PARMS = 1 'initial number of parameters
30 WHILE LEN(ENVIRON$(PARMS)) > 0
40 TABLE$(PARMS) = ENVIRON$(PARMS)
50 PARMS = PARMS+1
60 WEND
70 PARMS = PARMS - 1 'adjust to correct number
80 'now store new environment
90 ENVIRON "DATAIN=C:\DATAIN\INP.FIL"
100 ENVIRON "SORT.DAT=SORT<" + _
    ENVIRON$ ("DATAIN") + ">LPT1:"
    .
    .
    .
1000 SHELL ENVIRON$("SORT.DAT") 'data is sorted
1010 FOR I = 1 TO PARMS
1020 ENVIRON TABLE$(I) 'restore parameters
1030 NEXT I
    .
    .
    .
```

Purpose

The EOF statement indicates an end-of-file condition.

For communication files, EOF checks whether or not there are characters in the input buffer.

Format

$v = \text{EOF}(\text{filenum})$

Comments

filenum is the number specified on the OPEN statement.

EOF returns -1 (true) if end of file has been reached on the specified file. A 0 is returned if end of file has not been reached.

EOF is significant only for a file opened for sequential input from disk or for a communications file. A -1 for a communications file means that the buffer is empty. If you attempt to GET a record from beyond the end of a random access file, EOF is still false and no error is detected, but a null record is returned.

EOF(0) returns the end-of-file condition on standard input devices used with redirection of I/O.

Examples

This example reads information from the sequential file named DATA. Values are read into the array M until end of file is reached.

```
10 OPEN "DATA" FOR INPUT AS #1
20 C=0
30 IF EOF(1) THEN END
40 INPUT #1,M(C)
50 C=C+1: GOTO 30
```

ERASE Statement

Purpose

The ERASE statement removes or resets the elements of an array.

Format

ERASE *arrayname*[,*arrayname*]...

Comments

The arrayname is the name of the array you want to erase or reset.

The ERASE statement performs differently for static and dynamic arrays. When an ERASE statement is run on a static array, the array elements are set to either 0's or null strings. Running an ERASE on a dynamic array frees the array elements. Before making another reference to the dynamic array, you must first redimension it using either a DIM or REDIM statement.

You may want to use the ERASE statement if you are running short of memory space while running a program. After dynamic arrays are erased, the space in memory allocated for the arrays can be used for other purposes.

ERASE must be used when you want to redimension arrays in your program. If you try to redimension an array without first erasing it, a **Duplicate definition** error occurs.

ERASE Statement

If an array is declared dynamic inside a subprogram, that array still exists after you exit the subprogram. This causes an error if you call the subprogram a second time and again declare it. To avoid this, use ERASE to free the array before you EXIT the subprogram.

The CLEAR command erases *all* variables from the work area.

Examples

This example uses the FRE function to show how ERASE can be used to free memory. The array BIG used up about 40K bytes of memory when it was dimensioned as BIG(100,100). After it is erased, it can be redimensioned to BIG(10,10).

```
100 '$DYNAMIC
110 START=FRE(-1)
120 DIM BIG(100,100)
130 MIDDLE=FRE(-1)
140 ERASE BIG
150 REDIM BIG(10,10)
160 FINAL=FRE(-1)
170 PRINT START, MIDDLE, FINAL
```

Results:

```
415168  374336  414656
```

ERDEV and ERDEV\$ Variables

Purpose

The ERDEV AND ERDEV\$ variables hold the interrupt 24 error code of a device error and the name of the device generating the error. They are read-only variables.

These variables are not available in OS/2 mode.

Format

V = ERDEV

V\$ = ERDEV\$

Comments

ERDEV is a read-only variable. When a critical error is detected, ERDEV holds the DOS interrupt 24 error code in the lower eight bits, and the upper eight bits contain bits 13, 14, and 15 of the attribute word of the device header block.

ERDEV\$ is a read-only variable. If the error was on a character device, ERDEV\$ contains the eight-byte character device name. If the error was not on a character device, ERDEV\$ contains the two-character block device name (A:, B:, C:, and so on).

See also the "IOCTL Statement" and the "IOCTL\$ Function".

ERDEV and ERDEV\$ Variables

Examples

This example simulates a printer error:

```
10 CLS
20 ON ERROR GOTO 60
30 LPRINT"The printer is ready"
40 PRINT"The printer is ready"
50 END
60 V$=HEX$(ERDEV)
70 PRINT "ERDEV = ";V$
80 D$=ERDEV$
90 PRINT "ERDEV$ = ";D$
100 RESUME NEXT
```

If you run this example with the printer turned off, the computer displays:

```
ERDEV = 8009
ERDEV$ = LPT1
```

Note: If you are using a printer other than the IBM Graphics Printer, you may receive a different error code.

The lower eight bits (bits 0–7) of the binary equivalent equal 9, which is the interrupt 24 error code for **Printer out of paper**. The meaning of bits 13, 14, and 15 of the value returned by ERDEV is explained in “Attribute Field” in the *IBM Disk Operating System Technical Version 3.30 Reference* under “Installable Device Drivers.”

ERR and ERL Variables

Purpose

The ERR AND ERL variables return the error code and line number associated with an error.

Format

$V = \text{ERR}$

$V = \text{ERL}$

Comments

The variable ERR contains the error code for the last error.

If line numbers are used, ERL returns the number of the last line run before the error was detected. If line numbers are not used, ERL returns 0. The ERR and ERL variables are usually used in IF...THEN statements to direct program flow in the error-handling routine. See "ON ERROR Statement."

If you test ERL in an IF...THEN statement and you are using the interpreter as an editor, be sure to put the line number on the right side of the relational operator, like this:

```
IF ERL = line number THEN ...
```

ERR and ERL can be set using the ERROR statement.

Compiler error codes are listed in Appendix A, "BASIC Compiler Error Messages."

ERR and ERL Variables

Examples

This example tests to see if the drive door is open when the program needs to open a file:

```
10 ON ERROR GOTO 100
20 OPEN "DATA" FOR INPUT AS #1
30 END
.
.
.
100 IF ERR=71 THEN LOCATE 23,1:
    PRINT "DISK IS NOT READY":RESUME
```

ERROR

Statement

Purpose

The ERROR statement simulates the occurrence of a BASIC error or allows you to define your own error codes.

Format

ERROR *n*

Comments

n must be an integer expression from 1 through 255.

If the value of *n* is the same as an error code used by BASIC (see Appendix A, "BASIC Compiler Error Messages"), the ERROR statement simulates the occurrence of that error. If an error-handling routine has been defined by the ON ERROR statement, the error routine is entered. Otherwise, the error message corresponding to the code is displayed, and running halts. See the first example.

Note: If your program contains any ON ERROR or RESUME statements, you need to compile using the /X or /E parameter. See "Compiler Parameters" in *IBM BASIC Compiler/2 Compile, Link, and Run* for more information.

To define your own error code, use a value that is different from any used by BASIC. (We suggest you use the highest available values; for example, values greater than 200.) This new error code can then be tested in an error handling routine, just like any other error. See the second example.

If you define your own code in this way and you do not handle it in an error handling routine, the message **Unprintable error** appears, and running halts.

ERROR Statement

Examples

The first example simulates a **String formula too complex** error:

```
10 T = 16  
20 ERROR T
```

Results:

String formula too complex in line 20

The next example is a part of a game program that allows you to make bets. An error code of 210 is chosen because it is normally unused. The program traps the error if you exceed the house limit.

```
100 ON ERROR GOTO 1000  
110 INPUT "WHAT IS YOUR BET";B  
120 IF B > 5000 THEN ERROR 210  
.  
.  
.  
1000 IF ERR = 210 THEN PRINT  
    "HOUSE LIMIT IS $5000"  
1010 IF ERL = 120 THEN RESUME 110
```

EXP Function

Purpose

The EXP function calculates the exponential function.

Format

$v = \text{EXP}(x)$

Comments

The x can be any numeric expression.

This function returns the mathematical number e raised to the x power. The e is the base for natural logarithms. An overflow occurs if x is greater than 88.02969.

Examples

This example calculates e raised to the $(2-1)$ power, which is e :

```
10 X = 2
20 PRINT EXP(X-1)
```

Results:

2.718282

Purpose

The FIELD statement allocates space for variables in a random file buffer.

Format

FIELD [#]*filenum*, *width* AS *stringvar* [,*width* AS *stringvar*]...

Comments

filenum is the number under which the file was opened.

width is a numeric expression specifying the number of character positions to be allocated to *stringvar*.

stringvar is a string variable that is used for random file access.

A FIELD statement defines variables used to get data out of a random buffer after a GET or to enter data into the buffer for a PUT.

The statement:

```
FIELD 1, 20 AS N$, 10 AS ID$, 40 AS ADD$
```

allocates the first 20 positions (bytes) in the random file buffer to the string variable N\$, the next 10 positions to ID\$, and the next 40 positions to ADD\$. FIELD does *not* actually place any data into the random file buffer. This is done by the LSET and RSET statements. See “LSET and RSET Statements.”

FIELD does not remove data from the file either. Information is read from the file into the random file buffer with the GET (file) statement. Information is read from the buffer by simply referring to the variables defined in the FIELD statement.

FIELD Statement

The total number of bytes allocated in a FIELD statement must not exceed the record length specified when the file was opened. Otherwise, a **Field overflow** error occurs.

Any number of FIELD statements can be run for the same file number, and all FIELD statements that have been run are in effect at the same time. Each new FIELD statement redefines the buffer from the first character position, so this has the effect of having multiple field definitions for the same data.

Note: Be careful about using a variable name defined in a FIELD statement in an input or assignment statement. Once a variable name is defined in a FIELD statement, it points to the correct place in the random file buffer. If a subsequent input statement or LET statement with that variable name on the left side of the equal sign is run, the variable is moved to string space and is no longer in the file buffer. This can be avoided by assigning the input to a temporary variable, then using LSET or RSET to move the input into the variable declared in the FIELD statement.

See “BASIC Disk Input and Output” in *IBM BASIC Compiler/2 Fundamentals* for a complete explanation of how to use random files.

FIELD Statement

Examples

This example opens a file named CUST as a random file. The variable CUSTNO\$ is assigned to the first two positions in each record, CUSTNAME\$ is assigned to the next 30 positions, and ADDR\$ is assigned to the next 35 positions.

Lines 30 through 50 put information into the buffer, and the PUT statement in line 60 writes the buffer to the file. Line 70 reads back that same record, and line 90 displays the three fields. Note in line 80 that it is permissible to use a variable name that was defined in a FIELD statement on the *right* side of an assignment statement.

```
10 OPEN "A:CUST" AS #1
20 FIELD 1, 2 AS CUSTNO$, 30 AS CUSTNAME$,
   35 AS ADDR$
30 LSET CUSTNAME$= "O'NEIL INC"
40 LSET ADDR$= "50 SE 12TH ST, NY, NY"
50 LSET CUSTNO$=MKI$(7850)
60 PUT 1,1
70 GET 1,1
80 CNUM%= CVI(CUSTNO$): N$ = CUSTNAME$
90 PRINT CNUM%, N$, ADDR$
```

The following program shows two different ways to retrieve information from the same random file. The contents of the file do not change.

```
10 OPEN "PROG" AS #1
20 FIELD 1, 20 AS LASTNAME$,15 AS FIRSTTWO$
30 FIELD 1, 34 AS WHOLENAME$, 1 AS INITIALS
```

FILEATTR

Function

Purpose

The FILEATTR function returns information about an open file.

Format

$v = \text{FILEATTR}(\text{filenum}, \text{fieldnum})$

Comments

filenum is the file number.

fieldnum is the field number to return. It has two possible values:

1. File open mode. Use this when you want to know how the file was opened. This returns one of the following values as *v*:
 - 1 Sequential INPUT
 - 2 Sequential OUTPUT
 - 4 RANDOM access
 - 8 APPEND
 - 10 RANDOM character device (such as PIPE:)
2. DOS file handle. This returns the file handle as *v*.

FILEATTR Function

Examples

' This example uses FILEATTR to retrieve the file handle and
' status from a file.

```
OPEN "TEST.DAT" FOR OUTPUT AS #1

DOSHandle& = FILEATTR (1, 2)
Status&     = FILEATTR (1, 1)

PRINT "The file handle is: "DOSHandle&

SELECT CASE Status&

    CASE 1:
        PRINT "File is open for input"

    CASE 2:
        PRINT "File is open for output"

    CASE 4:
        PRINT "File is open for random access"

    CASE 8:
        PRINT "File is open for append"

    CASE 10:
        PRINT "File is a random character device"

END SELECT
END
```

FILES

Command

Purpose

The FILES command displays the names of files residing on a disk. The FILES command in BASIC is similar to the operating system DIR command.

Format

FILES [*filespec*]

Comments

filespec is a string expression for the file specification. It can contain a path and must conform to the rules outlined under "File Names" and "File Specification" in *IBM BASIC Compiler/2 Fundamentals*; otherwise, an error occurs.

If *filespec* is omitted, all the files on the current directory of the default drive are listed.

All files matching the file name are displayed. The file name can contain question marks (?). A question mark matches any character in the name or extension. An asterisk (*) in any character position matches any and all characters from that position on. If a drive is specified as part of *filespec*, files that match the specified file name on the current directory of that drive are listed; otherwise, the default drive is used.

Examples

This command displays all files on the current directory of the default drive:

```
FILES
```

FILES Command

This displays all files with an extension of .BAS on the current directory of the default drive:

```
FILES *.BAS"
```

This displays all files on drive B:

```
FILES "B:"
```

This lists each file on the current directory of the DOS default drive that has a file name beginning with TEST followed by up to two other characters, and an extension of .BAS:

```
FILES "TEST??.BAS"
```

In addition to listing all the files, the current directory name and the number of bytes free are also displayed.

When using tree-structured directories, remember that each subdirectory contains two special entries. They are listed when you use the FILES command to list a subdirectory. The first contains a single period instead of a file name. It identifies this "file" as a subdirectory. The second entry contains two periods instead of a file name. It locates the higher level directory that defines this subdirectory.

See "Input and Output" in *IBM BASIC Compiler/2 Fundamentals* for more information on tree structured directories.

This example lists all files in the current subdirectory called LEVEL1 on drive A:. Note that the directory is empty.

```
FILES "A:\LEVEL1"  
  
    . <DIR>      .. <DIR>  
  
32824 Bytes free
```

The FILES command can also be used to list files in other directories. The example below lists all files in the subdirectory LVL1. The backslash must be used after the directory name.

```
FILES "LVL1\"
```


FILES

Command

This example lists all files in the directory LVL2 with an extension of .BAS:

```
FILES "LVL2\*.BAS"
```

Purpose

The `FIX` statement truncates x to an integer.

Format

$v = \text{FIX}(x)$

Comments

The x can be any numeric expression.

`FIX` strips all digits to the right of the decimal point and returns the value of the digits to the left of the decimal point.

The difference between `FIX` and `INT` is that `FIX` does not return the next lower number when x is negative.

See “`INT`” and “`CINT` Functions”, which also return integers.

Examples

Note in the examples how `FIX` does *not* round the decimal part when it converts to an integer.

```
PRINT FIX(45.67)
```

Results:

```
45
```

```
PRINT FIX(-2.89)
```

Results:

```
-2
```

FOR and NEXT Statements

Purpose

The FOR and NEXT Statements perform a series of instructions in a loop a given number of times.

Format

```
FOR variable = x TO y [STEP z]  
    statements  
    [EXIT FOR]  
    statements  
NEXT [variable [,variable]...]
```

Comments

variable is an integer, long integer, single- or double-precision variable to be used as a counter.

x is a numeric expression that is the initial value of the counter.

y is a numeric expression that is the final value of the counter.

z is a numeric expression to be used as an increment.

statements are any statements that you want BASIC to repeat.

The program lines following the FOR statement are run until the NEXT statement is encountered. Then the counter is increased by the amount specified by the STEP value (*z*). If you do not specify a value for *z*, the increment is assumed to be 1. A check is performed to see if the value of the counter is now greater than the final value (*y*). If it is not greater, BASIC branches back to the statement after the FOR statement and the process is repeated. If it is greater, running continues with the statement following the NEXT statement.

FOR and NEXT Statements

If the value of *z* is negative, the test is reversed. The counter is decreased each time through the loop, and the loop is run until the counter is less than the final value.

The body of the loop is skipped if *x* is already greater than *y* when the STEP value is positive, or *x* is less than *y* when the STEP value is negative. If *z* is 0, an infinite loop is created unless you provide some way to set the counter greater than the final value or use the EXIT FOR statement.

Program performance improves if you use integer counters when possible.

Nested Loops

FOR...NEXT loops can be nested; that is, one FOR...NEXT loop can be placed inside another FOR...NEXT loop. When loops are nested, each loop must have a unique variable name as its counter. FOR...NEXT loops must be nested physically as well as logically; that is, the NEXT statement for the inside loop must appear before that for the outside loop.

Note: If nested loops have the same end point, a single NEXT statement can be used for all of them.

A NEXT statement of the form:

```
NEXT var1, var2, var3 ...
```

is equivalent to the sequence of statements:

```
NEXT var1  
NEXT var2  
NEXT var3  
:
```

The variables in the NEXT statement can be omitted, in which case the NEXT statement matches the most recent FOR statement. It is a good idea to always include the variables to avoid confusion, but it can be necessary if you do any branching out of nested loops. However, using variable names on the NEXT statements causes your program to run somewhat slower.

FOR and NEXT Statements

Active loops should be exited by setting the loop counter out of range or setting a conditional statement with the loop causing the loop to end, so that every iteration of the FOR statement in the loop has a corresponding NEXT.

You can also use the EXIT FOR statement to end a loop. When BASIC encounters an EXIT FOR, control drops to the next line after the loop. For example:

```
FOR I = 1 TO 5
INPUT A$
IF A$="END" THEN EXIT FOR
PRINT A$
NEXT
```

If a NEXT statement is encountered before its corresponding FOR statement, a **NEXT without FOR** error occurs.

Examples

The first example shows a FOR...NEXT loop with a STEP value of 2:

```
10 J=10: K=30
20 FOR I=1 TO J STEP 2
30 PRINT I;
40 K=K+10
50 PRINT K
60 NEXT
```

Results:

```
1 40
3 50
5 60
7 70
9 80
```

FOR and NEXT Statements

In the following example, the loop does not run because the initial value of the loop is more than the final value:

```
10 J=0
20 FOR I=1 TO J
30 PRINT I
40 NEXT I
```

The next program results in an error at compile time. There can be only one NEXT statement for every FOR statement. (This is different from other versions of BASIC that allow a different physical NEXT statement when jumping out of a loop.)

```
10 FOR I=1 TO 5
20 IF I=2 GOTO 50
30 NEXT
40 GOTO 60
50 NEXT
60 END
```

In the following example, the loop runs 10 times. The final value for the loop variable is always set before the initial value is set. (This is different from some other versions of BASIC, which set the initial value of the counter before setting the final value. In another BASIC, the loop in this example might run six times.)

```
10 I=5
20 FOR I=1 TO I+5
30 PRINT I;
40 NEXT
```

Results:

```
1 2 3 4 5 6 7 8 9 10
```

FRE

Function

Purpose

The `FRE` function returns the amount of available memory.

Format

$v = \text{FRE}(-1)$

$v = \text{FRE}(x)$

$v = \text{FRE}(x\$)$

Comments

The `x` and `x$` are dummy arguments.

`FRE` can return these different types of information:

- `FRE(-1)` returns the size of the largest block of free space for dynamic numeric arrays.
- `FRE` with any other numeric argument returns the size of the next free block of string space. Normally, this is also the largest block of string space.
- `FRE` with any string value causes a housecleaning before returning the number of free bytes. `BASIC` collects all its useful data and frees up unused areas of memory once used for strings. The data is compressed so you can continue until you really run out of space.

If a string allocation exceeds the size of the current block, the compiler either finds a free block large enough to hold the string, or it does a housecleaning. This function can be used as an indicator of when a housecleaning may take place.

You can improve the performance of programs that run many string functions by using the `MID$` statement to access substrings imbedded

FRE Function

within one large string. This prevents fragmentation of string space. See the “MID\$ Statement” for an example.

Note: Ctrl + Break cannot be used during housecleaning.

Examples

The actual value returned by FRE on your computer can differ from this example:

```
PRINT FRE(0)
```

Results:

14542

FREEFILE

Function

Purpose

The FREEFILE function returns the next free file number as an integer.

Format

$V = \text{FREEFILE}$

Comments

This function helps you manage file numbers. Because file numbers are global to an entire BASIC program, it may be difficult for you to assign file numbers in separately compiled modules. This function returns the next free file number as an integer.

FREEFILE Function

Examples

```
' This example inputs a file name then uses FREEFILE to allocate a
' "free" file number to the file when it is open. It then reads
' the file and lists it on the screen.

' This subroutine uses FREEFILE to assign a file number to a file.
' Because of this, it can be called from any main program without
' regard for file numbers it may use:
SUB ListFile (FileName$) STATIC

    FileNum = FREEFILE           ' allocate the file number
    OPEN FileName$ FOR INPUT AS FileNum ' open the file

    DO WHILE NOT EOF(FileNum)
        LINE INPUT #FileNum, L$
        PRINT L$
    LOOP

    CLOSE FileNum

END SUB

' -- Main Program --
CLS
INPUT "File to list? ", FileName$

' If file name isn't blank, open it and list it:
IF FileName$ <> "" THEN
    CLS
    ListFile FileName$
END IF
END
```

FUNCTION Statement

Purpose

The FUNCTION statement defines and names a function that you write.

Format

```
FUNCTION name[(parameter [ AS type] [,parameter [ AS type]]...)] STATIC  
  statements  
  [name = expression]  
  [EXIT FUNCTION]  
  statements  
  name = expression  
END FUNCTION
```

Comments

name is the name (up to 40 characters long) that you want to call the FUNCTION. The name of the FUNCTION cannot be the same as any simple variable of the same type or any array variable of the same type, nor can it be the same as the name of a subprogram or user-defined function (DEF FN).

parameter is the name of a simple variable or an array. If the parameter is an array, it must be specified in the form:

parameter (*integer*)

where *integer* is the number of dimensions the array has.

type is one of the following:

- INTEGER
- LONG
- SINGLE
- DOUBLE
- STRING

FUNCTION Statement

- *typename*

typename must have been defined in a previous TYPE statement.

STATIC is required to indicate that the FUNCTION will not be recursive; that is, the FUNCTION will not call itself or a procedure that in turn calls it.

statements are the BASIC statements to be performed when the FUNCTION is called.

expression defines the returned value of the FUNCTION. The type of *expression* (numeric or string) must match the type declared by name.

Results must be assigned to the FUNCTION name prior to leaving or ending the FUNCTION . Otherwise, the results will be lost.

To call a FUNCTION, use its name any place an expression can be used. Follow the name with a list of arguments (enclosed in parentheses) to be passed to the FUNCTION . For example, the following would be valid for a function that returns the largest value of the integer array passed to it.

```
LGST%=LARGEST% (ARRAY%())  
PRINT "LARGEST=";LARGEST%(X%())  
IF LARGEST%(NUM%())>100 GOTO 2000
```

The following is not a valid call for a FUNCTION because there is no place for the FUNCTION's value to be returned.

```
LARGEST% (ARR1%())
```

If you want to call a FUNCTION before it is defined, you must first use the DECLARE statement to describe the FUNCTION to BASIC.

When a function is exited and reentered, the values of its local variables are reset to 0s and null strings. To guarantee that a local vari-

FUNCTION Statement

able retains its assigned value upon reentering the FUNCTION, it should be declared as `STATIC`. See “Scope of Variables” under “Modular Programming” in *IBM BASIC Compiler/2 Fundamentals* for more information.

FUNCTIONS are similar to multi-line functions defined with the `DEF FN` statement, except for the following:

- FUNCTIONS may have local static and dynamic variables. Any simple variables or arrays referred to in the FUNCTION are considered to be local unless they have been explicitly declared to be `SHARED` variables.
- FUNCTIONS are public. They can be called from other modules.
- BASIC does not change FUNCTION parameters. If a FUNCTION is called with a variable that is not the same type as the FUNCTION expects, an error occurs. FUNCTIONS are the same as subprograms in this respect.

See “`DECLARE` Statement” in this book for details on declaring external functions and forward referenced functions

FUNCTION Statement

Examples

This example declares a function and calls it.

```
' Define a function to reverse a string
FUNCTION ReverseString$ (S$) STATIC

    StringLen = LEN(S$)

    ' Put chars in reverse order into new string:
    BackString$ = ""
    FOR I=StringLen TO 1 STEP -1
        BackString$ = BackString$ + MID$(S$,I,1)
    NEXT I

    ' Return reversed string as value of function:
    ReverseString$ = BackString$

END FUNCTION

' -- Main Program --
CLS

' Input some text:
PRINT "Type some text then press <ENTER>:"
LINE INPUT T$

' Use the function to reverse the text and print it:
Backwards$ = ReverseString$(T$)
PRINT
PRINT "The Same thing reversed is:"
PRINT Backwards$

END
```

FUNCTION Statement

This example program defines and uses the ADD function to add five pairs of numbers.

```
FUNCTION ADD (X AS INTEGER, Y AS INTEGER) STATIC
    ADD = X + Y                                ' Assign result to function name
END FUNCTION

'Main Routine
FOR I% = 1 to 5
    READ A%, B%
    C%=ADD(A%,B%)                              ' Invoke the ADD function
    PRINT C%                                    ' Print the returned value
NEXT I%

DATA 1,2,3,4,5,6,7,8,9,10

END
```

Results:

```
3
7
11
15
19
```

GET Statement (Files)

Purpose

The GET statement reads a record from a random file into a random buffer.

Format

```
GET [#]filenum[, [number][,id]]
```

Comments

filenum is the number under which the file was opened.

number is the number of the record to be read, in the range 1 through 2147483647. If *number* is omitted, the next record (after the last GET or PUT) is read into the buffer.

id is any BASIC record variable. You cannot use *id* if a FIELD statement is active on the file.

If field buffers are used after a GET statement, then INPUT #, LINE INPUT #, or references to variables defined in the FIELD statement can be used to read characters from the random file buffer. See "BASIC Disk Input and Output" in *IBM BASIC Compiler/2 Fundamentals* for more information on using GET.

If you specify *id*, GET transfers data from the specified record number to the variable *id*. If *id* is smaller than the *id* size, then BASIC skips to the start of the next record in the file.

Because the operating system blocks as many records as possible in 512-byte sectors, the GET statement does not necessarily perform a physical read from the disk.

GET can also be used for communications files. In this case, *number* is the number of bytes to read from the communications buffer. This

GET Statement (Files)

number cannot exceed the value set by the LEN option on the OPEN "COM... statement.

Random files in BASIC have fixed-length records. The requested record number in a GET or PUT statement is multiplied by this fixed-record length to form a 31-bit product. This value is then used to move the random file pointer by a DOS call to read or write the desired record. Other record-number restrictions are:

- The largest record number possible is 214748364, so the largest record number available is:

$$214748364 / \text{record length}$$

- File size is limited by the available disk space.

Note: The IBM BASIC Compiler/2 stores floating-point data in random files differently than the BASIC Interpreter and previous versions of the BASIC Compiler. See "Floating Point Data in Random Files" under "Disk Data Files – Sequential and Random I/O" in *IBM BASIC Compiler/2 Fundamentals* for more information.

Examples

This example opens the file CUST for random access, with fields defined in line 20. The GET statement on line 30 reads a record into the file buffer. Line 40 displays the information from the record that was read.

```
10 OPEN "A:CUST" AS #1
20 FIELD 1, 30 AS CUSTNAME$, 30 AS ADDR$, 35 AS CITY$
30 GET 1
40 PRINT CUSTNAME$, ADDR$, CITY$
```

GET Statement (Graphics)

Purpose

The GET statement reads points from an area of the screen.

The GET Statement works in Graphics mode only.

Format

GET (x1,y1)-(x2,y2),arrayname [(index)]

Comments

(x1,y1), (x2,y2) are coordinates in either absolute or relative form. Refer to "Specifying Coordinates" under "Graphics Modes" in *IBM BASIC Compiler/2 Fundamentals* for more information on coordinates.

arrayname is the name of the array you want to hold the information.

index describes the starting location of the information within the array. If you do not specify an index, BASIC assumes that the data starts at the beginning of the array.

GET reads the attributes of the points within the specified rectangle into the array. The specified rectangle has points (x1,y1) and (x2,y2) as opposite corners. (This is the same as the rectangle drawn by the LINE statement using the **B** option.)

GET and PUT can be used for high-speed object motion in graphics mode. You might think of GET and PUT as "bit pump" operations that move bits onto (PUT) and off (GET) the screen. Remember that PUT and GET are also used for random access files, but the syntax of these statements is different.

GET Statement (Graphics)

The array is used simply as a place to hold the image and must be numeric; it can be any precision, however. The required size of the array, in bytes, is:

$$4 + \text{INT}((x * \text{bitsperpixel} + 7) / 8) * y$$

where x and y are the lengths of the horizontal and vertical sides of the rectangle, respectively.

The following figure contains the number of bits per pixel for each screen mode:

Mode	Bits per pixel	Formula
SCREEN 1	2	$\text{LOG}_2(4)$
SCREEN 2	1	$\text{LOG}_2(2)$

For example, suppose you want to use the GET statement to get a 10 by 12 image in medium resolution. The number of bytes required is $4 + \text{INT}((10 * 2 + 7) / 8) * 12$, or 40 bytes. The bytes per element of an array are:

- Two for integer string
- Four for single-precision string
- Eight for double-precision string.

Therefore, you could use a static integer array with at least 20 elements. Because dynamic arrays are allocated in 16-byte increments, your array size must be a multiple of 16; otherwise, an error occurs.

The information from the screen is stored in the array as follows:

1. Two bytes giving the x -dimension in bits.
2. Two bytes giving the y -dimension in bits.
3. The data itself.

GET Statement (Graphics)

It is possible to examine the x- and y- dimensions and even the data itself if an integer array is used. The x-dimension is in element 0 of the array, and the y dimension is in element 1.

Keep in mind that integers are stored low byte first, then high byte, but the data is actually transferred high byte first, then low byte.

The data for each row of points in the rectangle is left-justified on a byte boundary. So if less than a multiple of 8 bits is stored, the rest of the byte is filled with 0's.

PUT and GET work significantly faster in medium resolution when $x1 \text{ MOD } 4$ is equal to 0, and in high resolution when $x1 \text{ MOD } 8$ is equal to 0. This is a special case where the rectangle boundaries fall on the byte boundaries.

See the "SCREEN Statement" for detailed information on the various screen modes.

Examples

See "PUT Statement (Graphics)" for an example.

GOSUB Statement

Purpose

The GOSUB statement branches to and returns from a subroutine.

Format

GOSUB *line*|*label*

Comments

line is the line number of the first line of the subroutine.

label is a sequence of 1 through 40 letters, digits, or periods, in any combination.

To distinguish labels from keywords, line numbers, or variables, each label must be followed by a colon (:) when it starts a line.

Numeric labels, alphanumeric labels, and line numbers can be intermixed in the same program.

Note: If you wish to use error-reporting (with the ERL variable) and error-trapping, you *must* use line numbers.

The *line* or *label* must be at the same level as the GOSUB statement. That is, *line* or *label* and GOSUB must either be in the same subprogram or both at the main program level.

A subroutine can be called any number of times in a program, and a subroutine can be called from within another subroutine. Such nesting of calls is limited only by available memory.

The RETURN statement causes BASIC to branch back to the statement following the most recent GOSUB statement. A subroutine can contain more than one RETURN statement, so you can return from different points in the subroutine. Subroutines can appear anywhere in the program.

GOSUB Statement

To prevent your program from accidentally entering a subroutine, you can put a STOP, END, or GOTO statement before the subroutine to direct program control around it.

Use ON...GOSUB to branch to different subroutines based on the result of an expression.

See also "CALL Statement."

Examples

This example shows how a subroutine works. The GOSUB in line 10 calls the subroutine in line 40. The program branches to line 40 and starts running statements there until it sees the RETURN statement in line 70. At that point, the program goes back to the statement after the subroutine call; that is, it returns to line 20. The END statement in line 30 prevents the subroutine from being performed a second time.

```
10 GOSUB 40
20 PRINT "BACK FROM SUBROUTINE"
30 END
40 PRINT "SUBROUTINE";
50 PRINT " IN";
60 PRINT " PROGRESS"
70 RETURN
```

Results:

```
SUBROUTINE IN PROGRESS
BACK FROM SUBROUTINE
```

GOTO Statement

Purpose

The GOTO statement branches unconditionally out of the normal program sequence to a specified line number or label.

Format

GOTO *line*|*label*

Comments

line is the line number of a line in the program.

label is a sequence of 1 through 40 letters, digits, or periods, in any combination.

The *line* and *label* must be at the same level as the GOTO statement. That is, *line* or *label* and GOTO must either be in the same subprogram or both at the main program level.

If *line* is the line number of an executable statement, that statement and those following are run. If *line* refers to a nonexecutable statement (such as REM or DATA), the program continues at the first executable statement encountered after *line*.

Use ON...GOTO to branch to different lines based on the result of an expression.

GOTO Statement

Examples

In this example, the GOTO statement in line 60 puts the program into an infinite loop, which is stopped when the program runs out of data in the DATA statement. (Notice how branching to the DATA statement does not add additional values to the internal data table.)

```
10 DATA 5,7,12
20 READ R
30 PRINT "R = ";R,
40 A = 3.14*R^2
50 PRINT "AREA = ";A
60 GOTO 10
```

Results:

```
R = 5          AREA = 78.5
R = 7          AREA = 153.86
R = 12         AREA = 452.16
Out of DATA in module name at address nnnn:nnnn
Hit any key to return to system
```

The following example illustrates how labels can be used as targets instead of line numbers:

```
PRINT "ENTER WHAT TYPE PET YOU HAVE"
INPUT A$
IF A$="CAT" THEN GOTO FELINE
IF A$="DOG" THEN GOTO CANINE
PRINT "IT MUST BE A LADYBUG THEN"
GOTO 961
FELINE: PRINT " OH, HOW IS HE ?"
GOTO 961
CANINE: PRINT " ARE HIS TEETH SHARP ?"
961: END
```

HEX\$ Function

Purpose

The HEX\$ statement returns a string that represents the hexadecimal value of the decimal argument.

Format

$v\$ = \text{HEX}\(n)

Comments

n is a numeric expression in the range -2147483648 through 2147483647 .

If n is negative, the two's complement form is used.

See "OCT\$ Function" for octal conversion.

Examples

The following example uses the HEX\$ function to figure the hexadecimal representation for the two decimal values that are entered:

```
10 INPUT X
20 A$ = HEX$(X)
30 PRINT X " DECIMAL IS ";A$ " HEXADECEIMAL"
```

Results:

```
? 32
32 DECIMAL IS 20 HEXADECEIMAL
```

```
? 1023
1023 DECIMAL IS 3FF HEXADECEIMAL
```

Purpose

The IF Statement makes a decision regarding program flow based on the result of an expression.

Format

Single-Line Format:

```
IF expression [,] THEN clause [,][ELSE [clause]]
```

```
IF expression [,] GOTO line|label [,][ELSE [clause]]
```

Block Format:

```
IF expression THEN  
    statements  
[ELSEIF expression THEN  
    statements]  
:  
[ELSE  
    statements]  
END IF | ENDIF
```

Comments

expression can be any numeric expression.

clause can be a BASIC statement or a sequence of statements (separated by colons); or it can be simply the number of a line to branch to.

line is the line number of a line existing in the program.

label is a sequence of 1 through 40 letters, digits, or periods, in any combination.

IF

Statement

statements can be any BASIC statements (on one or more lines) that you want BASIC to run depending on how *expression* evaluates.

The *line* or *label* must be at the same level as the IF statement. That is, *line* or *label* and IF must either be in the same subprogram or both at the main program level.

Single-Line Format

If the *expression* is true (not 0), BASIC runs the THEN or GOTO clause. THEN is followed by either a line number for branching or one or more statements to be run. GOTO is always followed by a line number or a label.

If the result of *expression* is false (0), BASIC ignores the THEN or GOTO clause and runs the ELSE clause, if it is present. BASIC then continues with the next executable statement.

Also note that for the single-line format, IF...THEN...ELSE is just *one statement*. Once an IF statement occurs on a line, everything else on that line is part of the IF statement. Because IF...THEN...ELSE is all one statement, the ELSE clause cannot be a separate program line. For example:

```
10 IF A=B THEN X=4
20 ELSE P=Q
```

is incorrect. Instead, it should be:

```
10 IF A=B THEN X=4 ELSE P=Q
```

Nesting of IF Statements: IF statements can be nested. Nesting is limited only by the length of the line. For example:

```
IF X>Y THEN PRINT "GREATER" ELSE IF Y>X THEN PRINT "LESS THAN" ELSE PRINT "EQUAL"
```

is a correct statement. If the statement does not contain the same number of ELSE and THEN clauses, each ELSE is matched with the closest unmatched THEN. For example:

```
IF A=B THEN IF B=C THEN PRINT "A=C" ELSE PRINT "A<>C"
```

does not print "A < > C" when A < > B.

IF Statement

Note: Control constructs must be properly nested. IBM BASIC Compiler/2 does not allow a nested NEXT in an IF statement. The three examples that follow are constructs allowed by the IBM BASIC Compiler 2.00 that are not allowed by IBM BASIC Compiler/2.

- . If expression THEN ... :NEXT
- . If expression THEN NEXT ELSE...
- . If expression THEN ... ELSE NEXT

Block Format

When you use the block format of IF statements, remember these rules:

- The IF, ELSEIF, ELSE, and END IF keywords must be the first words on their program lines.
- Only comments (preceded by an apostrophe) may be on the same line after THEN; otherwise the compiler will interpret this as the single-line format of the IF statement.
- The ELSEIF and ELSE statements are optional.
- You may use as many ELSEIF statements as you wish, but you may use only one ELSE statement and it must come after the last ELSEIF statement.

If the *expression* in the IF statement is true (not 0), BASIC runs the next executable statement or statements then skips to the END IF statement. BASIC then continues with the next executable statement outside of the IF block.

IF

Statement

If the *expression* in the IF statement is false (0), BASIC skips to the next ELSEIF statement and evaluates the *expression* associated with it. If this *expression* is true, BASIC runs the next executable statement or statements and skips to the END IF statement. If this *expression* is false, BASIC proceeds through the remaining ELSEIF statements until it finds a true *expression*.

If none of the *expressions* in the IF or ELSEIF statements are true, BASIC comes to the ELSE statement (if one exists) and runs the executable statements following it. If no ELSE statement exists, BASIC comes to the END IF statement and exits the IF block. Processing continues with the statements after the END IF statement.

Note: When using IF to test equality for a value that is the result of a single-precision or double-precision computation, remember that the internal representation of the value may not be exact. (This is because single-precision and double-precision values are stored internally in floating-point binary format.) Therefore, the test should be against the *range* over which the accuracy of the value can vary.

IF Statement

For example, to test a computed variable A against the value 1.0, use:

```
IF ABS (A-1.0)<1.0E-6 THEN ...
```

This test returns a true result if the value of A is 1.0 with a relative error of less than 1.0E-6.

When the expression in the IF statement is comparing the result of a math operation to a variable it is important to understand that the result of the math operation is stored with more digits of precision than the variable. Therefore, you should assign the result of the math operation to a variable and then use the IF statement to compare two variables.

Instead of:

```
IF A#/B# = C# THEN...
```

Use:

```
D# = A#/B#  
IF D# = C# THEN...
```

Examples

The following program fragments demonstrate the use of block IF, and illustrate the difference between the single-line form and the block form.

The following example computes simple discount prices using the single-line IF form. Note that this is just one *logical* line carried over four *physical* lines by use of the line-continuation character “_”.

```
IF (x >= 10000) THEN PRINT PRICE! = x*.25! ELSE _  
IF (x < 10000) and (x >= 5000) THEN PRINT PRICE! = x*.2! ELSE _  
IF (x < 2500) AND (x >=1500) THEN PRINT PRICE! = x*.1 ELSE _  
PRINT "no discount"  
:
```

IF Statement

The following example uses block IF...THEN...ELSE to make the preceding more readable:

```
IF (x >= 10000) THEN
    PRINT PRICE! = x * .25!
ELSEIF (x < 10000) AND (x >= 5000) THEN
    PRINT PRICE! = x * .02!
ELSEIF (x < 5000) AND ( x >= 2500) THEN
    PRINT PRICE! = x * .01
ELSE PRINT "No discount"
END IF
:
```

This statement gets record I if I is not 0:

```
100 IF I THEN GET #1,I
```

In the next example, if I is between 10 and 20, DB is calculated and the program branches to line 300. If I is not in this range, the message OUT OF RANGE is printed. Note the use of two statements in the THEN clause.

```
100 IF (I>10) AND (I<20) THEN
    DB=1982-I: GOTO 300
ELSE PRINT "OUT OF RANGE"
END IF
```

In the next example, in line 30 everything after the THEN is part of the clause. This means that PRINT I is not executed unless N=I.

```
10 N=15
20 FOR I=1 TO 20
30     IF N=I THEN CLS: PRINT I
40 NEXT I
50 END
```

Results:

15

IF Statement

The following program loops, asking “DONE?”, until the user types “Y.” Notice that both ‘PRINT “YES”’ and ‘DONE=-1’ are part of the IF clause.

```
DONE=0
WHILE NOT DONE
  PRINT "DONE?";
  A$=INPUT$(1)
  IF A$="Y" OR A$="y" THEN
    PRINT "YES":DONE=-1
  ELSE PRINT "NO"
  END IF
WEND
```

Assume that you enter “Y.”

Results:

YES

This example demonstrates the block format:

```
IF A$ = B$ THEN
  PRINT A$ "=" B$
ELSEIF A$ < B$ THEN
  PRINT A$ "<" B$
ELSE
  PRINT A$ ">" B$
END IF
```

INKEY\$

Variable

Purpose

The INKEY\$ variable reads a character from the keyboard.

Format

V\$ = INKEY\$

Comments

INKEY\$ reads only a single character, even if several characters are waiting in the keyboard buffer. The returned value is a zero-, one-, or two-character string.

- A null string (length zero) indicates that no character is pending at the keyboard.
- A one-character string contains the actual character read from the keyboard.
- A two-character string indicates a special extended code. The first character is hex 00. For a complete list of these codes, see Appendix B “ASCII Character Codes”

You must assign the result of INKEY\$ to a string variable before using the character with any BASIC statement or function.

While INKEY\$ is being used, no characters are displayed on the screen and all characters are passed through to the program except:

- Ctrl + Break, which stops the program
- Ctrl + Num Lock, which sends the system into a pause state
- Alt + Ctrl + Del, which does a System Reset
- Shift + PrtSc or PrtSc, which prints the screen.
- Pause, which sends the system into a paused state.

Note: This key is not present on all keyboards.

INKEY\$ Variable

If you press Enter in response to INKEY\$, the carriage return character passes through to the program.

Examples

The following section of a program waits until any key is pressed:

```
100 PRINT "Press any key to continue"  
110 A$=INKEY$: IF A$="" THEN 110
```

The next example shows program lines that can be used to test a two-character code being returned:

```
100 KB$=INKEY$  
110 IF LEN(KB$)=2 THEN KB$=RIGHT$(KB$,1)
```

INP Function

Purpose

The INP function returns the byte read from port n .

This function is not available in OS/2 mode.

Format

$v = \text{INP}(n)$

Comments

The n must be in the range 0 through 65535.

INP is the complementary function to the OUT statement. See "OUT Statement."

INP performs the same function as the IN instruction in assembler language. See also the technical reference book for your computer for a description of valid port numbers (I/O addresses).

Examples

This example turns on the speaker, waits for you to press a key, then turns off the speaker:

```
100 A=INP(&H61)
110 OUT &H61, A OR 3 : REM Speaker on
120 WHILE INKEY$="" : WEND
130 OUT &H61, A AND NOT 3 : REM Speaker off
140 END
```

INPUT Statement

Purpose

The INPUT statement receives input from the keyboard while the program is running.

Format

```
INPUT[;][prompt";|,] variable[,variable]...
```

Comments

prompt is a string constant that prompts for the desired input.

variable is the name of the numeric or string variable or array element that receives the input.

When the program sees an INPUT statement, it pauses and displays a question mark on the screen to indicate that it is waiting for data. If a *prompt* is included, the string is displayed. If the *prompt* is followed by the semicolon (;), a question mark follows the displayed string. If the *prompt* is followed by a comma, the question mark is not displayed. For example, the statement:

```
INPUT "ENTER BIRTHDATE",BS
```

prints the prompt without the question mark.

After the prompt or question mark is displayed, you can enter the required data from the keyboard. The input editor supplied with the IBM BASIC Compiler/2 allows you to easily alter the response to an input statement if a mistake is made. Any corrections must be made before the Enter key is pressed. See "Differences Between the Compiler and the Interpreter" in *IBM BASIC Compiler/2 Compile, Link, and Run* for more information on the input editor.

The data you enter is assigned to the *variables* given in the variable list. The data items you supply must be separated by commas, and

INPUT

Statement

the number of data items must be the same as the number of variables in the list.

The type of data item that you enter must agree with the type specified by the variable name. (Strings entered in response to an INPUT statement need not be surrounded by quotation marks.)

If you respond to INPUT with too many or too few items or with the wrong type of value (letters instead of numbers, and so on.), BASIC displays the message **?Redo from start**. If a single variable is requested, you can simply press Enter to indicate the default values of 0 for numeric input or null for string input. However, if more than one variable is requested, pressing Enter causes the **?Redo from start** message to be printed because too few items were entered. BASIC does not assign any of the input values to variables until you give an acceptable response.

If INPUT is immediately followed by a semicolon, pressing Enter to input data does not produce a carriage return/line feed sequence on the screen. This means that the cursor remains on the same line as your response.

Examples

In this example, the question mark displayed by the computer is a prompt to tell you it wants you to enter something:

```
10 INPUT X
20 PRINT X "SQUARED IS" X^2
30 END
```

Results:

?

Suppose you enter a 5.

The program continues:

```
? 5
5 SQUARED IS 25
```

INPUT Statement

For this second example, a prompt was included in line 20, so this time the computer prompts with "WHAT IS THE RADIUS?"

```
10 PI=3.14
20 INPUT "WHAT IS THE RADIUS";R
30 A=PI*R^2
40 PRINT "THE AREA OF THE CIRCLE IS ";A
50 END
```

Results:

```
WHAT IS THE RADIUS?
```

Assume that you respond with 7.4. The program continues:

```
WHAT IS THE RADIUS? 7.4
THE AREA OF THE CIRCLE IS 171.9464
```

INPUT # Statement

Purpose

The `INPUT #` statement reads data items from a sequential device or file and assigns them to program variables.

Format

`INPUT #filenum, variable [,variable]...`

Comments

filenum is the number used when the file was opened for input.

variable is the name of a variable that will have an item in the file assigned to it. It can be a string or numeric variable or an array element.

The sequential file can reside on disk. It can be a sequential data stream from a communications adapter, or it can be the keyboard (KYBD:).

The type of data in the file must match the type specified by the variable name. Unlike `INPUT`, no question mark is displayed with `INPUT #`.

The data items in the file must appear just as they would if the data were being typed in response to an `INPUT` statement. With numeric values, the leading spaces, carriage returns, and line feeds are ignored. The first character encountered that is not a space, carriage return, or line feed is assumed to be the start of the number. The number ends with a space, carriage return, line feed, or comma.

INPUT # Statement

If BASIC is scanning the data for a string item, the leading spaces, carriage returns, and line feeds are also ignored. The first character encountered that is not a space, carriage return, or line feed is assumed to be the start of the string item. If this first character is a quotation mark ("), the string item consists of all characters read between the first quotation mark and the second. Thus, a quoted string cannot contain a quotation mark as a character. If the first character of the string is not a quotation mark, the string is an unquoted string. It ends after a comma, carriage return, or line feed, or after 255 characters have been read. If end of file is reached when a numeric or string item is being input, the item is cancelled.

Examples

See "BASIC Disk Input and Output" in *IBM BASIC Compiler/2 Fundamentals*.

INPUT\$

Function

Purpose

The INPUT\$ function returns a string of n characters read from the keyboard or from file number *filenum*.

Format

$v\$ = \text{INPUT}\$(n[, [\#]filenum])$

Comments

- n is the number of characters to be read from the file.
- filenum* is the file number used on the OPEN statement. If *filenum* is omitted, the keyboard is read.

If the keyboard is used for input, no characters are displayed on the screen. All characters (including control characters) are passed through except Ctrl + Break, which is used to interrupt the INPUT\$ function. When responding to INPUT\$ from the keyboard, it is not necessary to press Enter.

The INPUT\$ function allows you to read ASCII characters that normally are assigned special control functions, such as Backspace (ASCII code 8). If you want to read these special characters, use INPUT\$ or INKEY\$ (not INPUT or LINE INPUT.)

For communications files, the INPUT\$ function is preferred over the INPUT # and LINE INPUT # statements, because all ASCII characters can be significant in communications. See also "Communications" in *IBM BASIC Compiler/2 Fundamentals*.

INPUT\$ Function

Examples

The following program lists the contents of a sequential file in hexadecimal:

```
10 OPEN "DATA" FOR INPUT AS #1
20 IF EOF(1) THEN 50
30 PRINT HEX$(ASC(INPUT$(1,#1)));
40 GOTO 20
50 PRINT
60 END
```

The next example reads a single character from the keyboard in response to a question:

```
100 PRINT "TYPE P TO PROCEED OR S TO STOP"
110 X$=INPUT$(1)
120 IF X$="P" THEN 500
130 IF X$="S" THEN 700 ELSE 100
```

INSTR

Function

Purpose

The INSTR function searches for the first occurrence of string *y\$* in *x\$* and returns the position at which the match is found. The optional offset *n* sets the position for starting the search in *x\$*.

Format

v = INSTR(*[n,]x\$,y\$*)

Comments

n is a numeric expression in the range 1 through 32767.

x\$, y\$ can be string variables, string expressions, or string constants.

If *n* > LEN(*x\$*), or if *x\$* is null, or if *y\$* cannot be found, INSTR returns 0. If *y\$* is null, INSTR returns *n* (or 1 if *n* is not specified).

If *n* is out of range, an error is returned.

Examples

This example searches for the string "B" within the string "ABCDEB". When the string is searched from the beginning, "B" is found at position 2; when the search starts at position 4, "B" is found at position 6.

```
10 A$ = "ABCDEB"  
20 B$ = "B"  
30 PRINT INSTR(A$,B$);INSTR(4,A$,B$)
```

Results

```
2 6
```

Purpose

The INT function returns the largest integer that is less than or equal to x .

Format

$v = \text{INT}(x)$

Comments

x is any numeric expression.

This is called the “floor” function in some other programming languages.

See also the FIX and the CINT functions. (They also return integer values.)

INT

Function

Examples

These examples show how INT truncates positive integers but rounds negative numbers upward (in a negative direction). This is the first example, truncating a positive, real number:

```
PRINT INT(45.67)
```

The second example shows the truncation of a negative, real number.

```
PRINT INT(-2.89)
```

Results

The result of the first example is:

```
45
```

The result of the second example is:

```
-3
```

Purpose

The IOCTL statement allows BASIC to send a control data string to a device driver anytime *after* the driver has been opened using OPEN.

This statement is not available under the OS/2 mode.

Format

IOCTL [#]*filenum*,*string*

Comments

filenum is the file number of the device driver.

string is a string expression containing the control data.

The file I/O system of BASIC allows you to create and install your own device drivers. The IOCTL statement and the IOCTL\$ function send control data to and read data from your device driver.

An IOCTL command string can be up to 32767 bytes long. Multiple commands within the string can be separated by semicolons:

```
"LF;PL66;LW132"
```

You define the content and format of the control data string. The possible commands are determined by the characteristics of the driver installed.

The IOCTL statement works only if the following conditions are met:

- The device driver is installed.
- The device driver states that it processes IOCTL strings.
- BASIC performs an OPEN on a file on that device.

IOCTL

Statement

Most standard DOS device drivers do not process IOCTL strings. You must determine if the specific driver can handle the command.

Note: For related information, see “IOCTL\$ Function” in this book, “Device Drivers” in *IBM BASIC Compiler/2 Fundamentals*, and the device driver section of the *IBM Disk Operating System Version 3.30 Technical Reference*.

Examples

Initially, character device drivers for LPT1:, LPT2:, and LPT3: are installed, but they can be replaced. If you install a driver called LPT1 (no colon) to replace LPT1: and that driver is able to set page length, an IOCTL command string to set or change the page length might be: "PLn" (where n is the new page length) ,

You can then open the new LPT1 driver and set the page length with:

```
OPEN "LPT1" FOR OUTPUT AS #1
IOCTL #1, "PL60"
```

You could, for instance, write a device driver that controls a monitor and is capable of setting the mode of the screen to color and also capable of setting the width of the screen. For example:

```
OPEN "OPT" FOR OUTPUT AS #2
IOCTL #2, "CL:W40"
```

Assuming that your new driver accepts a command called “CL” to change the screen to color and a command called “Wn” to set the width of the screen, the previous example passes those commands to your driver and causes the screen to respond.

Purpose

The IOCTL\$ reads a control data string from a device driver that is open.

This function is not available under OS/2 mode.

Format

v\$ = IOCTL\$([#]*filenum*)

Comments

filenum is the number of the file open to the device.

The IOCTL\$ function can be used to get acknowledgment that an IOCTL command has succeeded or failed. It can also be used to get device configuration information, such as device width.

Examples

This example checks to see if control data was successfully received:

```
10 OPEN "COMM" AS #1
20 IOCTL #1, "SW132;GW"
30 IF IOCTL$(1) = "132" THEN PRINT "WIDTH SET SUCCESSFULLY"
```

If the device driver "COMM" returns a value not equal to 132 from the IOCTL\$ request, your command was not processed successfully, and you should check for errors. If a device failure occurs, check the system variables of ERDEV and ERDEV\$.

KEY

Statement

Purpose

The KEY statement sets or displays function keys and allows you to define key traps.

Format

KEY ON

KEY OFF

KEY LIST

KEY *n*, *x*\$

KEY *n*, CHR\$(*KBflag*) + CHR\$(*scan code*)

Comments

KEY ON causes the function key values to be displayed on the 25th line. When the width is 40, five of the function keys are displayed. When the width is 80, 10 function keys are displayed. In either width, only the first six characters of each value are displayed.

KEY OFF erases the function key display from the 25th line, making that line available for program use. It does not disable the function keys.

After turning off the function key display with KEY OFF, you can use LOCATE 25,1 followed by PRINT to display anything you want on the bottom line of the screen. Information on line 25 is not scrolled, as are lines 1 through 24.

KEY LIST lists on the screen all function key values that are appropriate for your hardware configuration.

KEY *n*,*x*\$ allows you to set each function key to automatically type any sequence of characters. ON is the default state for the function key display.

KEY Statement

n is the function key number in the range 1 to 10. 30 and 31 are also valid values for *n*.

Note: Assigning values of 30 and 31 only has meaning on keyboards that support function keys 11 and 12 respectively, such as the IBM Enhanced Keyboard.

x\$ is a string expression that is assigned to the key. (Remember to enclose string *constants* in quotation marks.)

The value of a function key *n* is reassigned the value of the string *x\$*. If the value entered for *n* is not valid, an **Illegal function call** error occurs. The previous key string assignment is retained. *x\$* can be 1 to 15 characters in length. If it is longer than 15 characters, only the first 15 characters are assigned.

Assigning a null string to a function key disables the function key.

When a function key is pressed, the `INKEY$` function returns one character of the function key string each time it is called. The first character is binary 0, the second is the key scan code, as listed in Appendix B, "ASCII Character Codes."

There are also eleven definable key traps. With this capability, you can trap any Ctrl, Shift, or super-shift key.

These additional keys are defined by the statement:

```
KEY n,CHR$(KBflag) + CHR$(scan code)
```

n is a numeric expression in the range 15 through 25 .

KBflag is a mask for the shifted keys. The appropriate bit in *KBflag* must be set in order to trap a key that is shifted, Alt-shifted, or Ctrl-shifted. The *KBflag* values in hex are:

Caps Lock &H0 if Caps Lock is not active.

Caps Lock &H40 if Caps Lock is active.

Num Lock &H0 if Num Lock is not active.

Num Lock &H20 if Num Lock is active.

KEY

Statement

- Alt** &H08 if the Alt key is pressed.
- Ctrl** &H04 if the Control key is pressed.
- Left Shift** &H02 if the Left Shift key is pressed.
- Right Shift** &H01 if the Right Shift key is pressed.

Duplicate key

&H80, for keyboards that have two keys with identical functions, if the duplicate key was pressed rather than the primary key. Pairs of Shift, Ctrl, or Alt keys are not considered to be duplicates and cannot be trapped using a KFlag of &H80.

On the IBM Enhanced Keyboard, the duplicate keys are the Enter key on the numeric keypad and the following keys that are found between the typewriter key area and the numeric keypad:

- Insert
- Delete
- Home
- End
- Page Up
- Page Down
- Cursor movement keys

Scan code is the number identifying one of the keys to trap.

See Appendix C, "Scan Codes."

Note that key trapping assumes that the Left and Right Shift keys are the same, so you can use a value of &H01, &H02, or &H03 (the sum of hex 01 and hex 02) to denote a Shift key.

You can also add multiple shift states. For example, the Ctrl and Alt keys can be added together. Shift state values *must* be in hex.

KEY Statement

When trapping a key or key combinations, you must know the state of Num Lock and Caps Lock.

When one of the "new" keys of the IBM Enhanced Keyboard is pressed, the keyboard sends a code to the computer indicating that the key is one of the new keys followed by the scan code for the "old" key from which it was derived. You can distinguish between the old keys and the new (duplicate) keys using the KBflag value &H80, as described above.

The Pause key, however, is handled slightly differently. When the Pause key is pressed, the "new key" code is sent, followed by the Ctrl key scan code and the Num Lock key scan code. Because there is no "new key" code between the Ctrl and Num Lock codes, BASIC cannot distinguish between the new Pause key and the old Num Lock key. Therefore, to trap the Pause key, trap the Num Lock key, as shown in the last example of this section.

Also notice that Ctrl+Break must be handled slightly differently on the IBM Enhanced keyboard than on other IBM keyboards. On the IBM Enhanced keyboard, you must add &H80 to kbflag. For example:

```
KEY 15,CHR$(&H80+&H04)+CHR$ (70) 'Trap for Ctrl+Break
```

When you trap keys, they are processed in the following order:

1. Ctrl + PrtSc, which activates the line printer, is processed first. Even if Ctrl + PrtSc is defined as a trappable key, this does not prevent characters from being echoed to the line printer.
2. Next, the function keys, the numeric keypad cursor control keys—Cursor Up, Cursor Down, Cursor Right, and Cursor Left (1-14)—are processed. Setting scan codes 59 to 68, 72, 75, 77, or 80 as key traps has no effect, because they are considered to be predefined.
3. Last, the keys you define for 15 through 25 are processed.

KEY

Statement

Notes:

1. Trapped keys do not go into the BIOS buffer. Therefore, only BASIC knows that the keys were pressed.
2. Be careful when you trap Ctrl + Break and Ctrl + Alt + Del, because unless you have a test in your trap routine, you will have to turn the power off to stop your program.
3. If you are using the IBM PC Convertible, you can use the FN key to trap certain keys. If you press the FN key, the NUM LOCK KB flag will automatically change states (become active if not active, and vice versa).

The following section, "KEY(n) Statement," explains how to enable and disable function key trapping.

Examples

This example displays the function keys on the 25th line.

```
50 KEY ON
```

This example erases the function key display. The function keys are still active, but not displayed.

```
10 KEY OFF
```

This example assigns the string "FILES" + Enter to function key 1. This is a way to assign a commonly used command to a function key.

```
10 KEY 1,"FILES"+CHR$(13)
```

This example disables function key 1.

```
10 KEY 1,""
```

KEY Statement

This example sets up a key trap for capital P. Note that all three KEY statements — KEY, KEY(n), and ON KEY—are used with key trapping.

```
10 KEY 15, CHR$(&H40)+CHR$(25)
20 ON KEY(15) GOSUB 1000
30 KEY(15) ON
40 GOTO 40
```

This example sets up a key trap for Ctrl + Shift A. Notice that the hex values for Ctrl (&H04) and Shift (&H03) are added together to get the shift state.

```
10 KEY 20, CHR$(&H04+&H03)+CHR$(30)
20 ON KEY(20) GOSUB 2000
30 KEY(20) ON
40 GOTO 40
```

The following example allows you to trap a duplicate key. It only has meaning for keyboards that have duplicate keys with identical functions.

```
10 REM Trap the Enter key on the numeric keypad
20 KEY 15, CHR$(&H80) + CHR$(28)
30 ON KEY(15) GOSUB 1000
40 KEY(15) ON
50 GOTO 50
```

The following example traps the Num Lock key and, on the IBM Enhanced Keyboard, the Pause key.

```
10 KEY 15, CHR$(&H0) + CHR$(&H45)
20 ON KEY(15) GOSUB 1000
30 KEY(15) ON
.
.
.
100 END
.
.
.
1000 PRINT "Num Lock or Pause key pressed."
1010 RETURN
```

KEY(n) Statement

Purpose

The KEY(N) statement activates and deactivates trapping of the specified key in a BASIC program. See "ON KEY(n) Statement."

Format

KEY(*n*) ON

KEY(*n*) OFF

KEY(*n*) STOP

Comments

n is a numeric expression with a value in the range 0 though 25. Values of 30 and 31 are also valid. The value indicates the trapped key:

- 0** All key traps
- 1 – 10** Function keys F1 to F10
- 11** Cursor Up
- 12** Cursor Left
- 13** Cursor Right
- 14** Cursor Down
- 15 – 25** Keys defined by the form:

KEY *n*,CHR\$(*KBflag*) + CHR\$(*scan code*).

- 30** Function key F11
- 31** Function key F12

KEY(*n*) ON must be run to activate trapping of function key or cursor control key activity. After KEY(*n*) ON , if a nonzero line number is specified in the ON KEY(*n*) statement, then every time BASIC starts a new statement or line (depending on whether you compiled using **/V** or **/W**), it checks to see if the specified key was pressed. If so, it performs a GOSUB to the line number or label specified in the ON KEY(*n*)

KEY(*n*) Statement

statement. A KEY(*n*) statement cannot precede an ON KEY(*n*) statement.

If KEY(*n*) is OFF, no trapping takes place and even if the key is pressed, the event is not remembered.

Once a KEY(*n*) STOP statement has been run, no trapping takes place. However, if you press the specified key your action is remembered, so that an immediate trap takes place when KEY(*n*) ON is executed.

Using a value of 0 for *n* allows you to globally change the status of all keys being trapped. You can turn all key traps on by executing:

```
KEY(0) ON
```

This statement is equivalent to:

```
KEY(1) ON  
KEY(2) ON  
KEY(3) ON  
:  
KEY(31) ON
```

Similarly, you can turn all key traps off by executing:

```
KEY(0) OFF
```

and you can stop all key trapping temporarily by executing:

```
KEY(0) STOP
```

KEY(*n*) ON has no effect on whether the function key values are displayed at the bottom of the screen.

See also “KEY Statement.”

Examples

The following example traps both the F1 and P keys. When P is pressed, it temporarily disables (or reenables) the trapping of the F1 key.

KEY(n) Statement

```
KEY 15,CHR$(&H00)+CHR$(25) 'Define trap number
                             'for P
ON KEY(1) GOSUB KEYTRAP
ON KEY(15) GOSUB PAUSE

KEY(1) ON
KEY(15) ON

PRINT "Press any key except F1 or P to quit."
PRINT
WAIT.HERE:
A$ = ""
A$ = INKEY$
IF A$ = "" THEN GOTO WAIT.HERE

END   '*** End of program ***

KEYTRAP:
TIMES& = TIMES& + 1
PRINT "F1 has been pressed ";TIMES&," times."
RETURN

PAUSE:
KEY(1) STOP   'Stop trapping F1, but remember
              'it if it is pressed
ON KEY(15) GOSUB UNPAUSE
RETURN

UNPAUSE:
KEY(1) ON     'Continue trapping F1
ON KEY(15) GOSUB PAUSE
RETURN
```

Purpose

The KILL command deletes a file from a disk. The KILL command in BASIC is similar to the operating system ERASE command.

Format

KILL *filespec*

Comments

filespec is a string expression for the file specification. It can contain a path and must conform to the rules outlined under "File Names" and "File Specification" in *IBM BASIC Compiler/2 Fundamentals*; otherwise, an error occurs.

KILL can be used for all types of disk files. The name must include the extension, if one exists.

If a KILL statement is given for a file that is currently open, a **File already open** error occurs.

Examples

To delete the file named "DATA1" on drive A:, you might use:

```
200 KILL "A:DATA1"
```

To delete the file "PROG.BAS" in the LEVEL2 subdirectory, you might use:

```
KILL "LEVEL1\LEVEL2\PROG.BAS"
```

KILL can be used only to delete files. The RMDIR command must be used to remove directories.

LBOUND

Function

Purpose

The LBOUND function returns the lower boundary (smallest available subscript) for a particular array.

Format

$v = \text{LBOUND}(\text{array}[,\text{dim}])$

Comments

array is the name of the array.

dim is an integer constant that indicates the number of the dimension whose lower bound you are requesting. The default value is 1.

If you used the DIM statement to specify an explicit lower bound for *array*, such as

```
DIM ARRAY(-10 TO 10)
```

then LBOUND returns the explicit lower bound.

If you did not specify an explicit lower bound, LBOUND returns a value of 0 or 1 depending on the setting of the OPTION BASE statement. The default lower bound is 0. LBOUND and UBOUND are particularly useful for determining the size of an array passed to a subprogram.

Note: See also "UBOUND Function."

LBOUND Function

Examples

The following example uses LBOUND and UBOUND to determine the size of the array to be sorted:

```
200 OPTION BASE 1
210 DIM SHARED A(10)
220 CLS
230 PRINT "THE UNSORTED ARRAY"
240 FOR I = LBOUND(A) TO UBOUND(A)
250   READ A(I)
260   PRINT A(I)
270 NEXT I
280 CALL SORT
290 PRINT "THE SORTED ARRAY"
300 FOR I = LBOUND(A) TO UBOUND(A)
310   PRINT A(I)
320 NEXT I
330 DATA 40, 100, 19, 8, 66, 23
340 DATA 83, 6, 54, 120, 25, 98
350 END
360 REM **** EXCHANGE SORT SUBPROGRAM ****
370 SUB SORT STATIC
380 STATIC B
390 REM USE LBOUND TO DETERMINE LOWER
400 REM BOUNDARY OF ARRAY
410 FOR I = LBOUND(A) TO UBOUND(A) - 1
420   FOR J = I + 1 TO UBOUND(A)
430     IF A(I) <= A(J) THEN 470
440     B = A(J)
450     A(J) = A(I)
460     A(I) = B
470   NEXT J
480 NEXT I
490 END SUB
```

LBOUND

Function

Results:

THE UNSORTED ARRAY

40
100
19
8
66
23
83
6
54
120

THE SORTED ARRAY

6
8
19
23
40
54
66
83
100
120

Purpose

The LCASE\$ function converts all the letters in a string to lowercase.

Format

$v\$ = \text{LCASE\$}(m\$)$

Comments

$m\$$ is any string expression. The letters in this string can be uppercase or lowercase.

The LCASE\$ function returns a string containing the characters of an argument string converted to lowercase. You can use this function to increase the speed of programs that use comparisons that are not sensitive to case.

Also see "UCASE\$ Function".

Examples

```
10 MIXED$ = "The LCASE Function."  
20 LOWER$ = LCASE$(MIXED$)  
30 PRINT "Mixed:",MIXED$  
40 PRINT "Lowercase:",LOWER$
```

Results

```
Mixed:      The LCASE Function.  
Lowercase:  the lcase function.
```

LEFT\$ Function

Purpose

The LEFT\$ function returns the leftmost n characters of $x\$$.

Format

$v\$ = \text{LEFT}\$(x\$,n)$

Comments

$x\$$ is any string expression.

n is a numeric expression that must be in the range 0 through 32767. It specifies the number of characters that are to be in the result.

If n is greater than $\text{LEN}(x\$)$, the entire string ($x\$$) is returned. If $n=0$, the null string (length zero) is returned.

See also "MID\$ Function and Statement" and "RIGHT\$ Function."

Examples

In this example, the LEFT\$ function is used to extract the first five characters from the string "BASIC PROGRAM":

```
10 A$ = "BASIC PROGRAM"  
20 B$ = LEFT$(A$,5)  
30 PRINT B$
```

Results:

```
BASIC
```

Purpose

The LEN function returns the number of bytes that a variable requires.

Format

$v = \text{LEN}(\text{variable}[(\text{index})])$

Comments

variable is any variable (scalar, array element, or record variable).

index is the subscript of the variable if it is an array element.

If the variable is a string expression, LEN includes unprintable characters and blanks in the count of the number of characters.

Examples

There are 14 characters in the string "BOCA RATON, FL" because the comma and the two blanks are counted:

```
10 X$ = "BOCA RATON, FL"  
20 PRINT LEN(X$)
```

Results:

14

LEN

Function

This function is also useful with record variables, as the following example illustrates:

```
TYPE X
  A AS INTEGER
  B AS INTEGER
END TYPE
DIM XINSTANCE AS X
PRINT LEN(XINSTANCE)
```

Results:

4

This example illustrates the use of LEN in opening a file:

```
TYPE X
  A AS INTEGER
  B AS INTEGER
END TYPE
DIM XINSTANCE AS X
OPEN "MYFILE" FOR RANDOM LEN=LEN(XINSTANCE) AS 1
GET #1,,XINSTANCE
```

Purpose

The LET statement assigns the value of an expression to a variable.

Format

[LET] *variable* = *expression*

Comments

variable is the name of the variable or array element that is to receive a value. It can be a string or numeric variable or array element.

expression is the expression whose value BASIC assigns to *variable*. The type of the expression (string or numeric) must match the type of the variable, or an error occurs.

The word LET is optional; that is, the equal sign is enough when assigning an expression to a variable name.

Examples

This example assigns the value 40 to the variable HOURS. It then assigns the value 134, which is the value of the expression HOURS * 3.35 to the variable PAY. The example also assigns the string "JOHN" to the variable EMPLOYEE\$:

```
10 LET HOURS=40
20 LET PAY=HOURS*3.35
30 LET EMPLOYEE$=JOHN
```

You can also write the same statements like this:

```
10 HOURS=40
20 PAY=HOURS*3.35
30 EMPLOYEE$=JOHN
```

LINE

Statement

Purpose

The LINE statement draws a line or a box on the screen.

This statement is used in Graphics mode only.

Format

```
LINE [(x1,y1)] -(x2,y2) [, [attribute] [, [B[F]]] [, style]]
```

Comments

(x1,y1), (x2,y2)

are the coordinates of the endpoints of a line or the opposite corners of a box in either absolute or relative form. See “Specifying Coordinates” under “Graphics Modes” in *IBM BASIC Compiler/2 Fundamentals*.

attribute

is an integer expression that specifies a color attribute. In SCREEN 1, (medium resolution), *attribute* can range from 0 through 3. In SCREEN 2, (high resolution), *attribute* can be 0 or 1.

The default color attribute for the foreground is the maximum color attribute for that screen mode. The COLOR statement can change this default.

The default color attribute for the background is always 0. If you draw a line with attribute 0, you cannot see the line.

B

tells LINE to draw a box with the opposite corners at coordinates *(x1,y1)* and *(x2,y2)*.

F

tells LINE to fill the box with color.

LINE Statement

style is a method of telling LINE which points to plot. You specify a 16-bit integer for *style*. BASIC converts this number into a binary number. LINE reads the digits of the number as it plots the points along the line or around the box. If the digit is 1, LINE plots a point. If the digit is 0, LINE does not plot a point. Then LINE moves to the next digit of the *style* number and the next point on the line. This technique is called line styling. You can use the *style* option for normal lines and boxes but not for filled boxes (BF). Using *style* with BF results in an error.

The simplest form of LINE is:

```
LINE -(X2,Y2)
```

This statement draws a line from the last-referred to point to the point (x2,y2) in the foreground attribute.

You can also include a starting point:

```
LINE (0,0)-(319,199) 'diagonal down screen  
LINE (0,100)-(319,100) 'horizontal bar across screen
```

You can indicate the attribute in which to draw the line:

```
LINE (10,10)-(20,20),2 'draw in attribute 2
```

```
10 'draw random lines in random colors  
20 SCREEN 1,0,0,0: CLS  
30 LINE -(RND*319,RND*199),RND*4  
40 GOTO 20
```

```
10 'alternating pattern - line on, line off  
20 SCREEN 1,0,0,0: CLS  
30 FOR X=0 TO 319  
40 LINE (X,0)-(X,199),X AND 1  
50 NEXT
```

The next argument to LINE is **B** (box), or **BF** (filled box). You can leave out *color* and include the argument:

```
LINE (0,0)-(100,100),,B 'box in foreground
```

Or you can include the attribute:

LINE

Statement

LINE (0,0)-(100,100),2,BF ' filled box attribute 2

The **B** tells LINE to draw a rectangle with the points $(x1,y1)$ and $(x2,y2)$ as opposite corners. This avoids having to give the four LINE commands:

```
LINE (X1,Y1)-(X2,Y1)
LINE (X1,Y1)-(X1,Y2)
LINE (X2,Y1)-(X2,Y2)
LINE (X1,Y2)-(X2,Y2)
```

The **BF** means “draw the same rectangle as **B**, but also fill in the interior points with the selected color.”

The last argument to LINE is *style*. LINE uses the current circulating bit in *style* to plot (or store) points on the screen. If the bit is 0, LINE does not plot a point. If the bit is 1, LINE plots a point. After each point, LINE moves to the next bit position in *style*. When LINE reaches the last bit position, LINE wraps around and begins with the first bit again.

A 0 bit tells LINE not to plot a point, but it does not erase the existing point on the screen. If you want a background color beneath a line, you can draw a background line before a styled line to force a known background.

LINE Statement

Examples

You can use the *style* option to draw a dotted line across the screen by plotting (storing) every other point. Because *style* is 16 bits wide, the pattern for a dotted line looks like this:

```
1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0
```

This is equal to AAAA in hexadecimal notation.

To draw a dotted line:

```
10 SCREEN 1,0
20 LINE (0,0)-(319,199),,,&HAAAA
```

To draw a cyan-colored box with dashes:

```
10 SCREEN 1,0
20 LINE (0,0)-(100,100),1,B,&HCCCC
```

The last point referred to after a LINE statement is point (x2,y2). If you use the relative form for the second coordinate, it is relative to the first coordinate. For example:

```
LINE (100,100)-STEP (10,-20)
```

draws a line from (100,100) to (110,80).

This example draws random boxes filled with random colors:

```
10 CLS
20 SCREEN 1,0: COLOR 0,0
30 LINE -(RND*319,RND*199),RND*2+1,BF
40 GOTO 30 'boxes will overlap
```

LINE INPUT

Statement

Purpose

The LINE INPUT statement reads an entire line (up to 32767 characters) from the keyboard into a string variable, ignoring delimiters.

Format

```
LINE INPUT [;][prompt";] stringvar
```

Comments

prompt is a string constant that BASIC displays on the screen before it accepts input. BASIC does not print a question mark unless it is part of the prompt string.

stringvar is the name of the string variable or array element to which BASIC assigns the line. BASIC assigns all input from the end of the prompt to the Enter to *stringvar*. BASIC ignores trailing blanks.

If a semicolon immediately follows LINE INPUT, pressing Enter to end the input line does not produce a carriage return/line feed sequence on the screen. That is, the cursor remains on the same line as your response.

The input editor supplied with the IBM BASIC Compiler/2 allows you to easily alter your response, if you have made a mistake. You must make any corrections before you press Enter.

LINE INPUT Statement

See “Differences between the Compiler and the Interpreter” in *IBM BASIC Compiler/2 Compile, Link, and Run* for more information on the input editor.

Examples

See the example in “LINE INPUT # Statement.”

LINE INPUT # Statement

Purpose

The **LINE INPUT #** statement reads an entire line (up to 32767 characters), ignoring delimiters, from a sequential file into a string variable.

Format

LINE INPUT #*filenum, stringvar*

Comments

filenum is the number under which the file was opened.

stringvar is the name of a string variable or array element to which the line is assigned.

LINE INPUT # reads all characters in the sequential file up to a carriage return. It then skips over the carriage return/line feed sequence, and the next **LINE INPUT #** reads all characters up to the next carriage return. (If a line feed/carriage return sequence is encountered, it is preserved. That is, the line feed/carriage return characters are returned as part of the string.)

LINE INPUT # is especially useful if each line of a file has been broken into fields, or if a BASIC program saved in ASCII mode is being read as data by another program.

See also "BASIC Disk Input and Output," in *IBM BASIC Compiler/2 Fundamentals*.

LINE INPUT # Statement

Examples

The following example uses LINE INPUT to get information from the keyboard, where the information is likely to have commas or other delimiters. The information is then written to a sequential file, and read back out from the file using LINE INPUT #.

```
10 OPEN "LST" FOR OUTPUT AS #1
20 LINE INPUT "Address? ";C$
30 PRINT #1, C$
40 CLOSE 1
50 OPEN "LST" FOR INPUT AS #1
60 LINE INPUT #1, C$
70 PRINT C$
80 CLOSE 1
```

Results:

Address?

Suppose you respond with DELRAY BEACH, FL 33445. The program continues:

```
Address? DELRAY BEACH, FL 33445
```

```
DELRAY BEACH, FL 33445
```

LOC Function

Purpose

The LINE INPUT # function returns the current position in the file.

For a communications file LOC returns the number of characters waiting in the input buffer.

Format

$v = \text{LOC}(\text{filenum})$

Comments

filenum is the file number used when the file was opened.

With random files, LOC returns the record number of the last record read or written to a random file since the file was opened.

With sequential files, LOC returns the number of records read from or written to the file since it was opened. (A record for sequential files is a 128-byte block of data.) When a file is opened for sequential input, BASIC reads the first sector of the file, so LOC returns a 1 even before any input from the file.

For a communications file, LOC returns the number of characters in the input buffer waiting to be read. The default size for the input buffer is 256 characters, but this can be changed by using the **RB** option of the OPEN "COM...Statement or b using the **IC:** option at compile time.

LOC Function

Examples

This example stops the program when the 50th record in the file is passed:

```
100 IF LOC(1)>50 THEN STOP
```

This example could be used to rewrite the record that was just read:

```
100 PUT #1,LOC(1)
```

LOCATE

Statement

Purpose

The LOCATE statement positions the cursor on the active screen. Optional parameters turn the blinking cursor on and off and define the size of the blinking cursor.

Format

```
LOCATE [row][, [col] [, [cursor][, [start] [, stop]]]]
```

Comments

- row* is a numeric expression in the range 1 through 25. It indicates the screen line number where you want to place the cursor.
- col* is a numeric expression in the range 1 through 40 or 1 through 80, depending upon screen width. It indicates the screen column number where you want to place the cursor.
- cursor* is a value indicating whether the cursor is visible or not. A 0 indicates off, a 1 indicates on.
- start* is the cursor-start scan line. It must be a numeric expression in the range 0 through 31.
- stop* is the cursor-stop scan line. It also must be a numeric expression in the range 0 through 31.

The *cursor*, *start*, and *stop* do not apply to graphics mode.

The *start* and *stop* allow you to make the cursor any size you want. You indicate the starting and ending scan lines. The scan lines are numbered from 0 at the top of the character position. The bottom scan line is 7 if you have the Color/Graphics Monitor Adapter, 13 if you have the IBM Monochrome Display and Printer Adapter. If *start* is given and *stop* is omitted, *stop* assumes the value of *start*. If *start* is

LOCATE Statement

greater than *stop*, you get a two-part cursor. The cursor “wraps” from the bottom line back to the top.

For all monitors, using *start* and *stop* values greater than the bottom scan line can cause unpredictable results.

After a LOCATE statement, I/O statements to the screen begin placing characters at the specified location.

When a program is running, the cursor is normally off. You can use LOCATE ,,1 to turn it back on.

Normally, the compiler does not print to line 25. However, if the function key display is turned off by using KEY OFF, you can use:

```
LOCATE 25,1: PRINT...
```

to put data on line 25. Line 25 does not scroll as the rest of the screen does.

Any parameter can be omitted. Omitted parameters assume the current value.

Any values entered outside the ranges indicated results in an **Illegal function call** error. Previous values are retained.

LOCATE

Statement

Examples

This example moves the cursor to the home position in the upper left-hand corner of the screen:

```
10 LOCATE 1,1
```

This example makes the blinking cursor visible; its position remains unchanged:

```
10 LOCATE ,,1
```

In this example, position and cursor visibility remain unchanged. The cursor is set to display at the bottom of the character on the Color/Graphics Monitor Adapter (starting and ending on scan line 7).

```
10 LOCATE ,,7
```

This example moves the cursor to line 5, column 1. It makes the cursor visible, covering the entire character cell on the Color/Graphics Monitor Adapter, starting at scan line 0 and ending on scan line 7.

```
10 LOCATE 5,1,1,0,7
```

Purpose

The LOCK statement restricts access by other processes to all or part of an opened file.

This statement is not available under OS/2.

Format

```
LOCK [#]n [, [recnum] [ TO recnum]]
```

Comments

n is the file number of the opened file.

recnum specifies the beginning and ending record numbers of the range of records to be locked. This is only meaningful if the file is opened for random access.

Under DOS, before you run an application that uses any LOCK or UNLOCK statements, you must first install the SHARE module. This module is on the DOS disk and is installed by entering the command "SHARE" at the DOS prompt or by installing network software.

If the file is opened for sequential access, the entire file is locked regardless of any range that is specified.

If a starting record number is not specified, record number 1 is assumed. If an end record is not specified, only one record is locked. The range of legal record numbers is 1 through 2147483647

Note: If locks are not removed before closing the file or program termination, unpredictable results can occur.

See also "UNLOCK Statement".

LOCK

Statement

Examples

The following example shows how LOCK and UNLOCK are used with a file that is opened for sequential access:

```
10 OPEN "DATA" FOR INPUT AS #1
20 ' OPENS DATA AS A SEQUENTIAL FILE
30 LOCK #1,1 TO 2
40 ' LOCKS ENTIRE FILE
50 UNLOCK #1,1 TO 2
60 ' UNLOCKS ENTIRE FILE
70 LOCK #1,2
80 ' ALSO LOCKS ENTIRE FILE
90 UNLOCK #1,2
100 ' UNLOCKS ENTIRE FILE
110 CLOSE #1
120 END
```

File access controls can be applied to files opened for sequential access or random access. When writing an application to be used in a networking environment, care should be taken that file access is properly controlled. Data loss or corruption can occur when file access control is not provided.

The following example shows how LOCK and UNLOCK are used with a file that is opened for random access:

```
10 OPEN "DATA2" AS #1
20 ' OPEN DATA2 AS RANDOM ACCESS FILE
30 LOCK #1,3
40 ' LOCKS RECORD #3 ONLY
50 LOCK #1,4 TO 10
60 ' LOCKS RECORDS #4 - #10
70 UNLOCK #1,4 TO 10
80 ' UNLOCKS RECORDS #4 - #10
90 UNLOCK #1,3
100 ' UNLOCKS RECORD #3
110 CLOSE #1
120 END
```

Purpose

The LOF function returns the number of bytes allocated to the file (length of the file).

For a communications file, LOF returns the amount of free space in the input buffer.

Format

$v = \text{LOF}(\text{filenum})$

Comments

filenum is the file number used when the file was opened.

LOF returns the actual number of bytes allocated to the file. If the disk file was created by BASIC Compiler 1.00, LOF returns the number of bytes allocated to the file in a multiple of 128. For example, if the actual data in the file is 257 bytes, LOF returns the number 384.

For communications, LOF returns the amount of free space in the input buffer. You can calculate this with the following formula:

$$\text{size} - \text{LOC}(\text{filenum})$$

where *size* is the size of the communications buffer, which defaults to 256 but can be changed by using the **RB** option of the OPEN "COM... statement or by using the **IC:** option at compile time. LOF can be used to detect when the input buffer is getting full. In practicality, LOC is adequate for this purpose, as demonstrated in the example in "Communications" in *IBM BASIC Compiler/2 Fundamentals*.

LOF

Function

Examples

This example gets the last record of the file named BIG (BIG was created with a record length of 128 bytes):

```
10 OPEN "BIG" AS #1  
20 GET #1,LOF(1)/128
```

Purpose

The LOG function returns the natural logarithm of x .

Format

$v = \text{LOG}(x)$

Comments

x must be a numeric expression greater than 0.

The natural logarithm is the logarithm to the base e .

Examples

The first example calculates the logarithm of the expression 45/7:

```
PRINT LOG(45/7)
```

Results:

1.860752

LOG Function

The second example calculates the logarithm of e :

```
E=2.718282  
PRINT LOG(E)
```

Results:

1

The following example calculates the logarithm of e^2 :

```
E=2.718282  
PRINT LOG(E*E)
```

Results:

2

Purpose

The LPOS function returns the current position of the print head within the printer buffer for LPT1, LPT2, or LPT3.

Format

$v = \text{LPOS}(n)$

Comments

n is a numeric expression that indicates the printer being tested, as follows:

0 or 1	LPT1:
2	LPT2:
3	LPT3:

Note: The colon is part of the device name and must be included when specifying a device.

The LPOS function does not necessarily give the physical position of the print head on the printer.

Examples

In this example, if the line length is more than 60 characters, a carriage return character is sent to the printer so that it skips to the next line:

```
100 IF LPOS(0)>60 THEN LPRINT CHR$(13)
```

LPRINT and LPRINT USING Statements

Purpose

The LPRINT AND LPRINT USING statements Print data on the printer (LPT1).

Format

LPRINT [*list of expressions* [;]]

LPRINT USING *v\$*; *list of expressions* [;]

Comments

list of expressions

is a list of the numeric and/or string expressions to be printed.
The expressions must be separated by commas or semicolons.

v\$ is a string constant or variable that identifies the format to be used for printing. This is explained in detail under the PRINT statement.

These statements function like PRINT and PRINT USING, except output goes to the printer. See "PRINT Statement" and "PRINT USING Statement".

LPRINT assumes an 80-character-wide printer. That is, BASIC automatically inserts a carriage return/line feed after printing 80 characters. This means that two lines are skipped when you print exactly 80 characters, unless you end the statement with a semicolon. You can change the width value with a WIDTH "LPT1:" statement.

LPRINT and LPRINT USING Statements

If you do a form feed (LPRINT CHR\$(12);) followed by another LPRINT and the printer takes more than 10 seconds to do the form feed, you can get a **Device timeout** error on the second LPRINT. To avoid this problem, enter the following:

```
1 ON ERROR GOTO 65000
.
.
.
65000 IF ERR = 24 THEN RESUME '24=timeout
```

You may want to test the ERL variable to make sure the timeout was caused by an LPRINT statement.

Examples

This is an example of sending special control characters to the IBM Graphics Printer using LPRINT and CHR\$. The printer control characters are listed in the technical documentation for your printer.

```
10 LPRINT CHR$(14);" Title Line"
20 FOR I=2 TO 4
30 LPRINT "Report line";I
40 NEXT I
50 LPRINT CHR$(15);"Condensed print;132 char/line"
60 LPRINT CHR$(18);"Return to normal"
70 LPRINT CHR$(27);"E"
80 LPRINT "This is emphasized print"
90 LPRINT CHR$(27);"F"
100 LPRINT "Back to normal again"
```

The output produced by this program looks like this:

```
  T i t l e   L i n e
Report line 2
Report line 3
Report line 4
Condensed print;132 char/line
Return to normal

This is emphasized print

Back to normal again
```

LSET and RSET Statements

Purpose

The LSET AND RSET statements Move data into a random file buffer in preparation for a PUT (file) statement.

Format

LSET *stringvar* = x\$

RSET *stringvar* = x\$

Comments

stringvar is the name of a variable defined in a FIELD statement.

x\$ is a string expression to place the information into the field identified by *stringvar*.

If x\$ requires fewer bytes than were specified for *stringvar* in the FIELD statement, LSET left-justifies the string in the field, and RSET right-justifies the string. (Spaces are used to pad the extra positions.) If x\$ is longer than *stringvar*, characters are dropped from the right.

Numeric values must be converted to strings before they are LSET or RSET. See MKI\$, MKL, MKS\$, and MKD\$ Functions."

See also "BASIC Disk Input and Output" in *IBM BASIC Compiler/2 Fundamentals* for a complete explanation of using random files.

LSET and RSET Statements

Note: LSET or RSET can also be used with a string variable that was not defined in a FIELD statement to left-justify or right-justify a string in a given field. For example, the following program lines right-justify the string N\$ in a 20-character field. This can be useful for formatting printed output.

```
10 A$=SPACE$(20)
20 RSET A$=N$
```

Examples

This example converts the numeric value AMT into a string, and left-justifies it in the field A\$ in preparation for a PUT (file) statement:

```
10 LSET A$=MK$(AMT)
```

LTRIM\$ Function

Purpose

The LTRIM\$ function Removes leading spaces from string expressions.

Format

$v\$ = \text{LTRIM\$(x\$)}$

Comments

$x\$$ is the name of the string you want to trim.

The LTRIM\$ function examines $x\$$, removes any spaces that pad the beginning of the string, and returns a new string, $v\$$, without the spaces. $x\$$ remains unchanged.

Also see "RTRIM\$ Function."

LTRIM\$ Function

Examples

This example demonstrates LTRIM\$ and RTRIM\$

```
DIM FixedString AS STRING * 10      ' FixedString = 10 character string
DIM NormalString$                ' NormalString= a dynamic string

FixedString = "Test"
NormalString$ = "Test"

' RTRIM$ must be used when comparing a fixed string with a normal
' one to trim off any default trailing blanks:
IF RTRIM$(FixedString) = NormalString$ THEN
    PRINT "The two strings are equal"

' If this happens, something's wrong:
ELSE
    PRINT "The two strings are not equal"
END IF

' Try a string with leading blanks:
NormalString$ = " Test"
IF RTRIM$(FixedString) = NormalString$ THEN
    PRINT "The two strings are still equal"

' LTRIM Removes the leading blanks so the comparison will work:
ELSEIF RTRIM$(FixedString) = LTRIM$(NormalString$) THEN
    PRINT "The two strings are equal if leading blanks are removed"

' If this happens, something's wrong:
ELSE
    PRINT "The two strings aren't equal"
END IF
END
```

MID\$ Function and Statement

Purpose

The MID\$ statement returns the requested part of a given string. When used as a statement, as in the second format, replaces a portion of one string with another string.

Format

As a function:

$$v\$ = \text{MID\$}(x\$,n[,m])$$

As a statement:

$$\text{MID\$}(v\$,n[,m]) = y\$$$

Comments

For the function ($v\$ = \text{MID\$}...$):

$x\$$ is any string expression.

n is an integer expression in the range 1 through 32767.

m is an integer expression in the range 0 through 32767.

The function returns a string of length m characters from $x\$$ beginning with the n th character. If m is omitted or if fewer than m characters are to the right of the n th character, all rightmost characters beginning with the n th character are returned. If m is equal to 0 or if n is greater than $\text{LEN}(x\$)$, MID\$ returns a null string.

See also "LEFT\$ Function" and "RIGHT\$ Function."

For the statement ($\text{MID\$}... = y\$$):

MID\$ Function and Statement

v\$ is a string variable or array element that will have its characters replaced.

n is an integer expression in the range 1 through 32767.

m is an integer expression in the range 0 through 32767.

y\$ is a string expression.

The characters in *v\$*, beginning at position *n*, are replaced by the characters in *y\$*. The optional *m* refers to the number of characters from *y\$* used in the replacement. If *m* is omitted, all of *y\$* is used.

However, regardless of whether *m* is omitted or included, the length of *v\$* does not change. For example, if *v\$* is four characters long and *y\$* is five characters long, then after the replacement *v\$* contains only the first four characters of *y\$*.

Note: If either *n* or *m* is out of range, an **Illegal function call** error is returned.

MID\$ Function and Statement

Examples

The first example uses the MID\$ function to select the middle portion of the string B\$:

```
10 A$="GOOD "  
20 B$="MORNING EVENING AFTERNOON"  
30 PRINT A$;MID$(B$,9,7)
```

Results:

```
GOOD EVENING
```

The next example uses the MID\$ statement to access substrings imbedded within one large string. This technique reduces fragmentation of string space.

```
10 RECORD$ = STRING$(255,0)  
20 PART1.OFF = 1  
30 PART1.LEN = 5  
40 PART2.OFF = 6  
50 PART2.LEN = 15  
.  
.  
.  
100 MID$(RECORD$,PART1.OFF,PART1.LEN) = "STRNG"
```

Purpose

The MKDIR command creates a directory on the specified disk.

Format

MKDIR *path*

Comments

path is a string expression, not exceeding 63 characters, that identifies the new directory to be created. For more information about paths, refer to "File Specification" and "Tree-Structured Directories" in *IBM BASIC Compiler/2 Fundamentals*.

Examples

This example creates from the root directory a subdirectory called APPS:

```
MKDIR "APPS"
```

This example creates from the root directory a subdirectory called FIN under the directory APPS:

```
MKDIR "APPS\FIN"
```

This example creates from the root directory a subdirectory called WP under the directory APP:

```
MKDIR "APPS\WP"
```

This example creates from the root directory a subdirectory called LANG:

```
MKDIR "LANG"
```

This example makes LANG the current directory, and then creates two subdirectories called BASIC and FORTRAN:

MKDIR

Command

```
CHDIR "LANG"  
MKDIR "BASIC"  
MKDIR "FORTRAN"
```

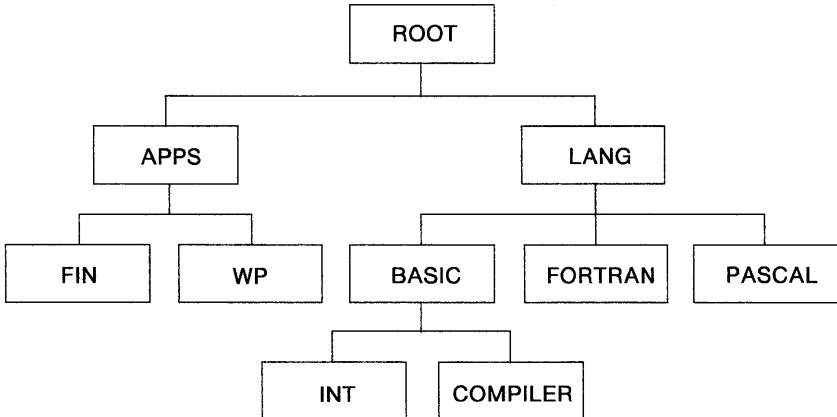
The same structure can be created from the root by entering:

```
MKDIR "LANG\BASIC"  
MKDIR "LANG\FORTRAN"
```

The following example creates from the root directory subdirectories called **COMPILER** and **INT** under the subdirectory **LANG\BASIC**:

```
MKDIR "LANG\BASIC\COMPILER"  
MKDIR "LANG\BASIC\INT"
```

By following the preceding examples, you create a tree structure that looks like this:



MKI\$, MKL\$, MKS\$, MKD\$ Functions

Purpose

The MKI\$, MKL\$, MKS\$, MKD\$ functions convert numeric type values to string type values for placement in random files.

Format

$v\$ = \text{MKI}\$(\textit{integer expression})$

$v\$ = \text{MKL}\$(\textit{long integer expression})$

$v\$ = \text{MKS}\$(\textit{single-precision expression})$

$v\$ = \text{MKD}\$(\textit{double-precision expression})$

Comments

Any numeric value that is placed in a random file buffer with an LSET or RSET statement must be converted to a string. MKI\$ converts a long integer to a 2-byte string. MKL\$ converts an integer to a 4-byte string. MKS\$ converts a single-precision number to a 4-byte string. MKD\$ converts a double-precision number to an 8-byte string.

These functions differ from STR\$ because they do not really change the bytes of the data. They change the way BASIC interprets those bytes.

If you want to store floating-point numbers in a random file so they can be read by programs created with the BASIC Interpreter or a previous version of the BASIC Compiler, you must use the MKSMBF\$ and MKDMBF\$ functions instead of the MKS\$ and MKD\$ functions. See the next section for details on these functions.

See also the CVI, CVL, CVS, and CVD functions in this book and "BASIC Disk Input and Output" in *IBM BASIC Compiler/2 Fundamentals*.

MKIS\$, MKL\$, MKS\$, MKD\$

Functions

Examples

This example uses a random file (#1) that has previously been opened with the fields defined in line 100. The first field, D\$, is intended to hold a numeric value, AMT. Line 110 converts AMT to a string value using MKS\$ and uses LSET to place what is really the value of AMT into the random file buffer. Line 120 places a string into the buffer (it is not necessary to convert a string). Line 130 writes the data from the random file buffer to the file.

```
100 FIELD #1, 4 AS D$, 20 AS N$
110 LSET D$ = MKS$(AMT)
120 LSET N$ = AS
130 PUT #1
```

MKSMBF\$, MKDMBF\$ Functions

Purpose

The MKSMBF\$ and MKDMBF\$ functions translate a number in IEEE format into Microsoft Binary Format (MBF) and return it as a string.

Format

v\$ = MKSMBF\$(*single*)

v\$ = MKDMBF\$(*double*)

Comments

single is a single-precision number in IEEE format.

double is a double-precision number in IEEE format.

Numeric values written to a random file that will be read with an earlier version of the BASIC Compiler or with the BASIC Interpreter must be converted to MBF format. You can use the MKSMBF\$ and MKDMBF\$ functions to convert these numbers from IEEE format back to Microsoft Binary Format.

You can also use the **/CV** parameter on the IBM BASIC Compiler/2 command line to make the conversion automatic.

Also see "CVSMBF, CVDMBF Functions."

MKSMBF\$, MKDMBF\$ Functions

Examples

This example reads a record from an old format random access file and allows new values to be entered in its fields. It uses CVSMBF, CVDMBF, MKSMBF\$ and MKDMBF\$ to convert from the old Microsoft Binary format to the current IEEE number format.

```
' Define the record structure for file record:
TYPE OldRecord
  ID   AS STRING * 10
  Cost AS STRING * 4   ' Single precision number
  Amt  AS STRING * 8   ' Double precision number
END TYPE

' Define a variable of the above structure:
DIM Buff AS OldRecord

' Open file:
OPEN "OLD.DAT" FOR RANDOM AS #1

' Get the first record:
GET #1, 1, Buff

' Decode values:
CostVal = CVSMBF(Buff.Cost)   ' Single precision value
AmtVal# = CVDMBF(Buff.Amt )  ' Double precision value

' Get updated values:
PRINT "Current: "Buff.ID", "CostVal", "AmtVal#"
INPUT "New?   : ", NewID$, CostVal, AmtVal#

' Encode the new values for writing to the file:
Buff.Cost = MKSMBF$(CostVal)
Buff.Amt  = MKDMBF$(AmtVal#)
Buff.ID   = NewID$

' Write the updated record to the file:
PUT #1, 1, Buff

END
```

NAME Command

Purpose

The NAME command changes the name of a disk file. The NAME command in BASIC is similar to the operating system RENAME command.

Format

NAME *oldspec* AS *newspec*

Comments

oldspec is a string expression containing the file specification of an existing file. If the drive or path is not specified, BASIC assumes the current drive or directory.

newspec is a string expression containing the new file specification for the file. If no path is specified, BASIC assumes the current directory.

newspec can include a drive name only if it is the same disk drive as the one specified by *oldspec*. Different logical drives, defined using the operating system's ASSIGN command, are allowed as long as they both specify the same physical drive.

If either the file specified by *oldspec* does not exist or the file specified by *newspec* already exists, then an error occurs.

If the path in *newspec* is different from the path in *oldspec*, then the file will be "moved" to the new directory. The file itself is not moved, but its directory entry is.

See "Filenames" and "File Specification" in *IBM BASIC Compiler/2 Fundamentals* for more information on file specifications.

NAME

Command

Examples

In this example, the file that was formerly named ACCTS.BAS on the disk in drive A: is now named LEDGER.BAS:

```
NAME "A:ACCTS.BAS" AS "LEDGER.BAS"
```

Purpose

The OCT\$ function returns a string that represents the octal value of the decimal argument.

Format

$v\$ = \text{OCT}\(n)

Comments

n is a numeric expression in the range -2147483648 through 2147483647 .

If n is negative, the two's complement form is used.

See "HEX\$ Function" for hexadecimal conversion.

Examples

This example shows that 24 in decimal is 30 in octal:

```
PRINT OCT$(24)
```

Results:

```
30
```

ON COM(*n*) Statement

Purpose

The ON COM(*N*) statement sets up a line number or label for BASIC to branch to when there is information coming into the communications buffer.

This statement is only supported under DOS and OS/2 mode.

Format

ON COM(*n*) GOSUB *line|label*

Comments

n is the number of the communications adapter (1 or 2).

line is the line number of the beginning of the trap routine. Setting *line* equal to 0 disables trapping of communications activity for the specified adapter.

label is a sequence of 1 through 40 letters, digits, or periods, in any combination.

The *line* and *label* must be at the main program level; they cannot be in a subprogram or function.

A COM(*n*) ON statement must be run to activate this statement for adapter *n*. After COM(*n*) ON, if a non-0 line number is specified in the ON COM(*n*) statement, every time the program starts a new statement or line, (depending on whether you compiled using /V or /W), BASIC checks to see if any characters have come in to the specified communications adapter. If so, BASIC performs a GOSUB to the specified *line* or *label*.

ON COM(*n*) Statement

Notes:

1. If your program contains any event-trapping statements, such as ON COM, for example, you need to compile your program using the **/V** or **/W** switch.
2. If you compile your program with the **/O** switch, you must link the IBMCOM.OBJ module.

If COM(*n*) OFF is run, no trapping takes place for the adapter. Even if communications activity does take place, the event is not remembered.

If a COM(*n*) STOP statement is run, no trapping takes place for the adapter. However, any characters being received are remembered so an immediate trap takes place when COM(*n*) ON is run.

When the trap occurs, an automatic COM(*n*) STOP is run so that recursive traps never take place. The RETURN from the trap routine automatically does a COM(*n*) ON unless an explicit COM(*n*) OFF was performed inside the trap routine.

When an error trap takes place, all trapping is automatically disabled (including ON COM, ON ERROR, ON PEN, ON PLAY, ON STRIG, and ON TIMER).

Typically, the communications trap routine reads an entire message from the communications line before returning. It is not recommended that you use the communications trap for single character messages. Several characters may arrive within a short time interval, thereby causing only one event to occur. For example, characters may arrive during the communications trap routine.

You can use RETURN *line|label* to go back to the BASIC program at a fixed location. Use this nonlocal return with care, however, because any other GOSUBS, WHILES, or FORS active at the time of the trap remain active.

You can exit an active loop by setting the loop counter variable out of range or setting a conditional statement within the loop to end it.

ON COM(n) Statement

This insures that every repetition of a FOR has a corresponding NEXT and every repetition of a WHILE has a corresponding WEND.

Examples

This example sets up a trap routine for the first communication adapter at line 500:

```
150 ON COM(1) GOSUB 500
160 COM(1) ON
.
.
.
500   'incoming characters
.
.
.
590 RETURN
```

ON ERROR Statement

Purpose

The ON ERROR statement sets up a line number or label for BASIC to branch to when an error occurs.

Format

ON ERROR GOTO *line|label*

Comments

line is the line number of the first line of the error-trapping routine. If the line number does not exist, an **Undefined line number** error results.

label is a sequence of 1 through 40 letters, digits, or periods, in any combination.

line or *label* must be at the main program level; they cannot be in a subprogram or function.

Once error-trapping has been enabled, all errors detected cause a jump to the specified error-handling subroutine.

Note: If your program contains any ON ERROR or RESUME statements, you may need to compile using the /X or /E parameter. See “Compiler Parameters” in *IBM BASIC Compiler/2 Compile, Link, and Run* for more information.

To disable error-trapping, run an ON ERROR GOTO 0. Subsequent errors print an error message and halt the program. An ON ERROR GOTO 0 statement that appears in an error trapping subroutine causes BASIC to stop and print the error message for the error that caused the trap. It is recommended that all error-trapping subroutines run an ON ERROR GOTO 0 if an error is encountered for which there is no recovery action.

ON ERROR Statement

BASIC considers itself to be within the error-trapping routine from the time an error occurs. It branches to the line specified by the ON ERROR statement until a RESUME statement is encountered. You must use the RESUME statement to exit from the error trapping routine. See also "RESUME Statement."

Because error-trapping does not occur within the error-trapping routine, an ON ERROR GOTO *line* (within the error-trapping routine), where *line* is anything other than 0, does not work.

Note: If an error occurs while an error-handling subroutine is running, the BASIC error message is printed and the program ends. Error-trapping does not occur within the error-handling subroutine.

Examples

This example tests to see if the drive door is open when the program needs to open a file:

```
10 ON ERROR GOTO 100
20 OPEN "DATA" FOR INPUT AS #1
30 END
.
.
.
100 IF ERR=71 THEN LOCATE 23,1:
    PRINT "DISK IS NOT READY"
110 RESUME NEXT
```

ON...GOSUB and ON...GOTO Statements

Purpose

The ON...GOSUB AND ON...GOTO statements branch to one of several specified line numbers or labels depending on the value of an expression.

Format

ON *n* GOTO *line|label* [,*line|label*]...

ON *n* GOSUB *line|label* [,*line|label*]...

Comments

n is a numeric expression, rounded to an integer, if necessary. It must be in the range 0 through 255, or an **Illegal function call** error occurs.

line is the number of the line to which the program branches.

label is a sequence of 1 through 40 letters, digits, or periods, in any combination.

The *line* and *label* must be at the same level as the GOSUB or GOTO statement. That is, the *line* or *label* and the GOSUB or GOTO must all be in the same subprogram or all at the main program level.

The value of *n* determines which line number in the list the program uses for branching. For example, if the value of *n* is 3, the third line number in the list is the point at which the program branches.

In the ON...GOSUB statement, each line number in the list must be the first line number of a subroutine. Eventually you must have a RETURN statement to bring you back to the line following the ON...GOSUB.

ON...GOSUB and ON...GOTO Statements

If the value of n is 0, or greater than the number of items in the list (but less than or equal to 255), BASIC continues with the next executable statement.

Examples

The first example branches to line 150 if L-1 equals 1, to line 300 if L-1 equals 2, to line 320 if L-1 equals 3, and to line 390 if L-1 equals 4. If L-1 is equal to 0, or is greater than 4, the program goes to the next statement.

```
100 ON L-1 GOTO 150,300,320,390
```

The next example shows how to use an ON...GOSUB statement:

```
100 REM display menu
110 PRINT "1. Routine 1"
120 PRINT "2. Routine 2"
130 PRINT "3. Routine 3"
140 PRINT "4. Routine 4"
150 INPUT "Your choice?"; CHOICE
160 ON CHOICE GOSUB 200, 300, 400, 500
170 GOTO 100 ' redisplay menu after routine is done
200 REM start of first routine
.
.
.
290 RETURN
300 REM start of second routine
.
.
.
```

ON KEY(n) Statement

Purpose

The ON KEY(n) statement sets up a line number or label for BASIC to branch to when the specified function key or cursor control key is pressed.

Format

ON KEY(n) GOSUB *line|label*

Comments

n is a numeric expression in the range 1 through 25, 30, or 31 indicating the key to be trapped, as follows:

1 – 10 Function keys F1 to F10

11 Cursor Up

12 Cursor Left

13 Cursor Right

14 Cursor Down

15 – 25 keys defined by the form:

KEY *n*,CHR\$(*KBflag*) + CHR\$(*scan code*).

See “KEY(n) Statement” for more information.

30 Function key 11 on the IBM Enhanced Keyboard

31 Function key 12 on the IBM Enhanced Keyboard.

line is the line number of the beginning of the trapping routine for the specified key. Setting *line* equal to 0 stops trapping of the key.

label is a sequence of 1 through 40 letters, digits, or periods, in any combination.

ON KEY(*n*)

Statement

The *line* and *label* must be at the main program level; they cannot be in a subprogram or function.

A KEY(*n*) ON statement must be run to activate this statement. After KEY(*n*) ON, if a non-0 line number is specified in the ON KEY(*n*) statement, then every time the program starts a new statement or line number (depending on whether the program was compiled with /V or /W), BASIC checks to see if the specified key was pressed. If so, BASIC performs a GOSUB to the specified *line* or *label*.

Note: If your program contains any event-trapping statements, such as ON KEY, for example, you need to compile your program using the /V or /W switch.

If a KEY(*n*) OFF statement is run, no trapping takes place for the specified key. Even if the key is pressed, the event is not remembered.

If a KEY(*n*) STOP statement is run, no trapping takes place for the specified key. However, if the key is pressed, the event is remembered, and an immediate trap takes place when KEY(*n*) ON is run.

When the trap occurs, an automatic KEY(*n*) STOP is run so that recursive traps never take place. The RETURN from the trap routine automatically does a KEY(*n*) ON unless an explicit KEY(*n*) OFF was performed inside the trap routine.

Event trapping does not take place when BASIC is not running a program. When an error trap (resulting from an ON ERROR statement) takes place, all trapping is automatically disabled (including ON COM, ON ERROR, ON PEN, ON PLAY, ON STRIG, and ON TIMER).

Key trapping may not work if you press other keys before the specified key. The key that caused the trap cannot be tested using INPUT\$ or INKEY\$, so the trap routine for each key must be different if a different function is desired.

You can use RETURN *line*||*label* to go back to the BASIC program at a fixed line number. Use this nonlocal return with care, however, because any other GOSUBS, WHILEs, or FORs active at the time of the trap remain active.

ON KEY(n) Statement

You can exit an active loop by setting the loop counter variable out of range or setting a conditional statement within the loop to end it. This ensures that every repetition of a FOR has a corresponding NEXT and every repetition of a WHILE has a corresponding WEND.

KEY(n) ON has no effect on whether the soft key values are displayed at the bottom of the screen.

Special Considerations for DOS National Diskettes

The DOS national diskette keyboard programs have a feature that allows you to change between the United States and national keyboard at any time. Use the F1 or F2 key while holding down Alt and Ctrl to perform the switch. (See *IBM DOS* for more information on the DOS keyboard programs.) If your BASIC program traps either of these keys, it will not pass the information to the DOS keyboard program and the keyboard change will not take place. If your program needs to provide this ability to change keyboard formats, avoid trapping the F1 and F2 keys.

Note: The shift state you use when trapping either of these keys makes no difference when considering the DOS keyboard programs. Any shift of base state trapping of F1 and F2 prevents the keystroke from being passed to the DOS program.

ON KEY(n) Statement

Examples

The following is an example of a trap routine for function key 5:

```
100 .ON KEY(5) GOSUB 200
110 KEY(5) ON
.
.
.
200 'function key 5 pressed
.
.
.
290 RETURN 140
```

This example traps Ctrl + Break and Ctrl + Alt + Del. It assumes that Caps Lock, and Num Lock are currently disabled.

```
10 KEY 15,CHR$(&H04)+CHR$(70) 'Trap Ctrl+Break
20 KEY 16,CHR$(&H04+&H08)+CHR$(83) 'Trap Ctrl+Alt+Del
30 ON KEY(15) GOSUB 1000
40 ON KEY(16) GOSUB 2000
50 KEY(15) ON: KEY(16) ON
.
.
.
1000 PRINT "Trapping for Ctrl+Break"
1010 RETURN
2000 TRAPS=TRAPS+1
2010 ON TRAPS GOTO 2100,2200,2300,2400,2500
2020 '
2100 PRINT "First trap of System Reset":RETURN
2200 PRINT "Second trap of System Reset":RETURN
2300 PRINT "Third trap of System Reset":RETURN
2400 PRINT "Fourth trap of System Reset":RETURN
2500 KEY(16) OFF 'Disable trap of System Reset
```

Note: When specifying scan codes, you can use either hexadecimal or decimal notation.

Purpose

The ON PEN statement sets up a line number or label for BASIC to branch to when the light pen is activated.

This statement is not available under the OS/2 mode.

Format

ON PEN GOSUB *line|label*

Comments

line is the line number of the beginning of the trap routine for the light pen. Using a line number of 0 disables trapping of the light pen.

label is a sequence of 1 through 40 letters, digits, or periods, in any combination, followed by a colon.

The *line* and *label* must be at the main program level; they cannot be in a subprogram or function.

A PEN ON statement must be run to activate this statement. After PEN ON, if a non-0 line number is specified in the ON PEN statement, then every time the program starts a new statement BASIC checks to see if the pen was activated. If so, BASIC performs a GOSUB *line|label*.

If PEN OFF is run, no trapping takes place. Even if the light pen is activated, the event is not remembered.

If a PEN STOP statement is run, no trapping takes place, but pen activity is remembered so that an immediate trap takes place when PEN ON is run.

ON PEN

Statement

When the trap occurs, an automatic PEN STOP is run so recursive traps never take place. The RETURN from the trap routine automatically does a PEN ON unless an explicit PEN OFF was performed inside the trap routine.

PEN(0) is not set when pen activity causes a trap.

You can use RETURN *line|label* to go back to the BASIC program at a fixed line number. Use this nonlocal return with care, however, because any other GOSUBS, WHILES, or FORS active at the time of the trap remain active.

You can exit an active loop by setting the loop counter variable out of range or setting a conditional statement within the loop to end it. This ensures that every repetition of a FOR has a corresponding NEXT and every repetition of a WHILE has a corresponding WEND.

Examples

This example sets up a trap routine for the light pen:

```
10 ON PEN GOSUB 500
20 PEN ON
.
.
.
500 'subroutine for pen
.
.
.
650 RETURN 30
```

ON PLAY(*n*) Statement

Purpose

The ON PLAY(*n*) statement sets up a line number or label for BASIC to branch to when the music background buffer has gone from *n* to *n*-1 while the program is running.

This statement is not available in OS/2 mode.

Format

ON PLAY(*n*) GOSUB *line*|*label*

Comments

n is an integer expression in the range 1 through 32 indicating the notes to be trapped. Values entered outside this range result in an **Illegal function call** error.

line is the beginning line number of the trap routine for PLAY. A line number of 0 stops the trapping of PLAY.

label is a sequence of 1 through 40 letters, digits, or periods, in any combination.

The *line* and *label* must be at the main program level; they cannot be in a subprogram or function.

A PLAY ON statement must be used to start the ON PLAY(*n*) statement. You can then play continuous background music while the program is running. (See "PLAY Statement.") After PLAY ON, if a non-0 line number is specified in the PLAY(*n*) statement, each time the program starts a new statement or line number (depending on whether you compiled using /V or /W), BASIC checks to see if the music buffer has gone from *n* to *n*-1 notes. If so, BASIC performs a GOSUB to the specified line or label.

ON PLAY(*n*)

Statement

Note: If your program contains any event-trapping statements, such as ON PLAY, for example, you need to compile your program using the /V or /W switch.

If PLAY OFF is used, no trapping takes place. Even if a play activity takes place, the event is not remembered.

If a PLAY STOP statement is used, no trapping takes place, but play activity is remembered so that an immediate trap takes place when PLAY ON is run.

When the trap occurs, an automatic PLAY STOP is run so recursive traps never take place. The RETURN from the trap routine automatically does a PLAY ON unless an explicit PLAY OFF was performed inside the trap routine.

You can use RETURN *line|label* to go back to the BASIC program at a fixed line number. Use this nonlocal return with care, because any other GOSUBS, WHILES, or FORS active at the time of the trap remain active.

You can exit an active loop by setting the loop counter variable out of range or setting a conditional statement within the loop to end it. This ensures that every repetition of a FOR has a corresponding NEXT and every repetition of a WHILE has a corresponding WEND.

Notes:

1. A PLAY event trap is issued only when PLAY is in the Music Background mode (PLAY "MB..."). An event trap is not issued when PLAY is in the Music Foreground mode (PLAY "MF...").
2. A PLAY event trap is not issued if the Music Background buffer is already empty when a PLAY ON statement is performed.
3. Be careful choosing values for *n*. For example: ON PLAY(32) causes so many event traps that little time remains to run the rest of the program.

See also "PLAY(*n*) Function" for additional information.

ON PLAY(n) Statement

Examples

This example sets up a trap routine that is called when five notes are left in the background music buffer. This example works only in DOS mode.

```
10 ON PLAY(5) GOSUB 500
20 PLAY ON
.
.
.
500 'subroutine for background music
.
.
.
650 RETURN
```

ON SIGNAL Statement

Purpose

The ON SIGNAL statement sets up a line number or label for BASIC to branch to when the program receives an interprocess communication signal.

This statement is not available in DOS mode.

Format

```
ON SIGNAL(n) GOSUB line|label
```

Comments

n is the number of a signal that the program in the OS/2 mode sends. For a list of the valid signal numbers, see *IBM Operating System/2 Programmer's Guide*.

If you specify a signal number other than those which the OS/2 mode supports, BASIC returns an **Illegal function call** error message.

line is the line number of the beginning of the trap routine for SIGNAL(*n*). A line number of 0 stops trapping of SIGNAL.

label is a sequence of 1 through 40 letters, digits, or periods, in any combination.

The *line* and *label* must be at the main program level; they cannot be in a subprogram or function.

You must run a SIGNAL(*n*) ON statement to activate this statement for the OS/2 mode signals. If you run a SIGNAL(*n*) ON statement and specify a non-0 line number in the ON SIGNAL(*n*) statement, then every time the program starts a new statement BASIC checks to see if signal *n* has been received. If so, BASIC performs a GOSUB to the line or label that you specified.

ON SIGNAL Statement

Note: If your program contains any event-trapping statements, such as ON SIGNAL for example, you need to compile your program using the /V or /W switch.

If you run a SIGNAL(*n*) OFF statement, BASIC does not trap the OS/2 mode *n* signal. Even if signal *n* occurs, the program does not remember the event.

If you run a SIGNAL(*n*) STOP statement, BASIC does not trap signal *n*, but BASIC remembers the event and traps signal *n* as soon as you run a SIGNAL(*n*) ON statement.

When the trap occurs, BASIC automatically runs a SIGNAL(*n*) STOP so that recursive traps never take place. The RETURN from the trap routine automatically does a SIGNAL(*n*) ON, unless you ran an explicit SIGNAL(*n*) OFF inside the trap routine.

You can use RETURN *line|label* to go back to the BASIC program at a fixed line number. Use this nonlocal return with care, because any other GOSUBS, WHILES, or FORS active at the time of the trap remain active.

ON SIGNAL

Statement

Examples

In the following example, ON SIGNAL is used to count the number of times that signal 5 is received.

```
' Prepare to trap signal 5
ON SIGNAL(5) GOSUB RECEIVED.5
SIGNAL(5) ON

' Wait until signal 5 is received
PRINT "Press any key to stop the program."

DO
LOOP UNTIL INKEY$ <> ""

' Print the result and end
PRINT "Signal 5 was received ";COUNT%;" times."
END

' Trap for signal 5
RECEIVED.5:
COUNT% = COUNT% + 1
RETURN
```

ON STRIG(*n*) Statement

Purpose

The ON STRIG(*n*) statement sets up a line number or label for BASIC to branch to when one of the joystick buttons (triggers) is pressed.

This statement is not available under the OS/2 mode.

Format

ON STRIG(*n*) GOSUB *line|label*

Comments

n can be 0, 2, 4, or 6, and indicates the button to be trapped as, follows:

- 0** button A1
- 2** button B1
- 4** button A2
- 6** button B2

line is the line number of the beginning of the trap routine for STRIG. A line number of 0 stops trapping of the joystick button.

label is a sequence of 1 through 40 letters, digits, or periods, in any combination.

The *line* and *label* must be at the main program level; they cannot be in a subprogram or function.

A STRIG(*n*) ON statement must be run to activate this statement for button *n*. If STRIG(*n*) ON is run and a non-0 line number is specified in the ON STRIG(*n*) statement, then every time the program starts a new statement BASIC checks to see if the specified button has been pressed. If so, BASIC performs a GOSUB to the specified line or label.

ON STRIG(*n*)

Statement

Note: If your program contains any event-trapping statements, such as ON STRIG, for example, you may need to compile your program using the /V or /W switch.

If STRIG(*n*) OFF is run, no trapping takes place for button *n*. Even if the button is pressed, the event is not remembered.

If a STRIG(*n*) STOP statement is run, no trapping takes place for button *n*, but the button being pressed is remembered so that an immediate trap takes place when STRIG(*n*) ON is run.

When the trap occurs, an automatic STRIG(*n*) STOP is run so that recursive traps never take place. The RETURN from the trap routine automatically does a STRIG(*n*) ON unless an explicit STRIG(*n*) OFF was performed inside the trap routine.

Using STRIG(*n*) ON activates the interrupt routine that checks the button status for the specified joystick button. Downstrokes that cause trapping do not set functions STRIG(0), STRIG(2), STRIG(4), or STRIG(6).

You can use RETURN *line|label* to go back to the BASIC program at a fixed line number. Use this nonlocal return with care, because any other GOSUBS, WHILES, or FORS active at the time of the trap remain active.

You can exit an active loop by setting the loop counter variable out of range or setting a conditional statement within the loop to terminate it. This ensures that every iteration of a FOR has a corresponding NEXT and every iteration of a WHILE has a corresponding WEND.

ON STRIG(n) Statement

Examples

This is an example of a trapping routine for the button on the first joystick. This example is for DOS mode.

```
10 ON STRIG(0) GOSUB 500
20 STRIG(0) ON
.
.
.
500 'subroutine for 1st button
.
.
.
650 RETURN
```

ON TIMER

Statement

Purpose

The ON TIMER statement branches to a given line number or label in a BASIC program when a defined period of time has elapsed.

Format

```
ON TIMER(n) GOSUB line|label
```

Comments

n is a numeric expression in the range 1 through 86400 (1 second through 24 hours). Values entered that are outside this range result in an **Illegal function call** error.

line is the beginning line number of the trap routine for TIMER. A line number of 0 stops timer trapping.

label is a sequence of 1 through 40 letters, digits, or periods, in any combination.

The *line* and *label* must be at the main program level; they cannot be in a subprogram or function.

A TIMER ON statement must be used to start the ON TIMER statement. (See "TIMER Function and Statement.") After TIMER ON, if a non-0 line number is specified in the ON TIMER statement, then every time the program starts a new statement or line number, (depending on whether you compiled using **/V** or **/W**), BASIC checks to see if the specified number of seconds have passed. When *n* seconds have elapsed, BASIC performs a GOSUB to the specified line. The event trap occurs, and BASIC starts counting again from 0.

Note: If your program contains any event- statements, such as ON TIMER, for example, you need to compile your program using the **/V** or **/W** switch.

ON TIMER Statement

If `TIMER OFF` is used, no trapping takes place. Even if `TIMER` activity takes place, the event is not remembered.

If a `TIMER STOP` statement is used, no trapping takes place, but `TIMER` activity is remembered so that an immediate trap occurs when `TIMER ON` is used.

When the trap occurs, an automatic `TIMER STOP` is run so that recursive traps never take place. The `RETURN` from the trap routine automatically does a `TIMER ON` unless an explicit `TIMER OFF` was performed inside the trap routine.

You can use `RETURN line|label` to go back to the `BASIC` program at a fixed line number. Use this nonlocal return with care, because any other `GOSUBS`, `WHILES`, or `FORS` active at the time of the trap remain active.

You can exit an active loop by setting the loop counter variable out of range or setting a conditional statement within the loop to terminate it. This ensures that every iteration of a `FOR` has a corresponding `NEXT` and every iteration of a `WHILE` has a corresponding `WEND`.

Examples

`ON TIMER` is useful in programs that need an interval timer. This example displays the time of day on line 1 every minute:

```
10 CLS
20 ON TIMER(60) GOSUB 10000
30 TIMER ON
.
.
.
10000 OLDROW=CSRLIN 'save current row
10010 OLDCOL=POS(0) 'save current column
10020 LOCATE 1,1:PRINT TIME$;
10030 LOCATE OLDROW,OLDCOL 'restore row & col
10040 RETURN
```

OPEN Statement

Purpose

The OPEN statement allows input or output to a file or device.

Any reference to networking is for DOS only.

Format

```
OPEN filespec [FOR mode] [ACCESS access]  
[locking] AS [#]filenum [LEN=recl]
```

Alternate form:

```
OPEN mode2, [#] filenum, filespec [,recl]
```

Comments

filespec is a string expression for the file specification. It can contain a path and must conform to the rules outlined under "File Names" and "File Specification" in *IBM BASIC Compiler/2 Fundamentals*; otherwise, an error occurs.

mode is one of the following:

OUTPUT specifies sequential output mode.

Warning: Opening a file for sequential output destroys the contents of the file.

INPUT specifies sequential input mode.

APPEND specifies sequential output mode where the file is positioned to the end of data on the file when it is opened.

RANDOM specifies random input or output mode.

OPEN Statement

Note that *mode* must be a string constant, *not* enclosed in quotation marks. If *mode* is omitted, random access is assumed.

mode2 (alternate form) is a string expression with the first character being one of the following:

O specifies sequential output mode.

Warning: Opening a file for sequential output destroys the contents of the file.

I specifies sequential input mode.

A specifies sequential append mode.

R specifies random input/output mode.

access is one of the following:

READ allows file access for input only.

WRITE allows file access for output only.

READ WRITE allows file access for input and output.

locking is one of the following:

SHARED Any process on any machine may read or write the file.

LOCK READ

No other process is to be granted read access to this file. This access is granted only if no other process has locked read access to the file.

LOCK WRITE

No other process is to be granted write access to this file. This access is granted only if no other process has locked write access to the file.

LOCK READ WRITE

No other process is to be granted read or write access to this file. This access is

OPEN Statement

granted only if no other process has locked read or write access to the file.

If *locking* is not specified, the file may be opened for reading and writing any number of times by the process, but other processes are denied access to this file while it is opened.

File access control is provided to support a networking environment. If you are not familiar with this type of environment, you should understand that file access must be strictly controlled to protect the integrity of data files that are accessible by several people. Data loss can occur when two or more programs attempt to access the same file simultaneously. By using the file access and locking parameters, control can be established and data loss or corruption can be minimized. See *IBM Disk Operating System Version 3.30 Technical Reference* for more information on file sharing.

Note: You must use DOS mode and SHARE if you want to use the new OPEN access and *locking* parameters.

filenum is an integer expression whose value is from 1 through 255.

The *filenum* is the number associated with the file or device for as long as it is open and is used by other I/O statements to refer to the file or device.

Note: See "Differences Between the Compiler and the Interpreter" in *IBM BASIC Compiler/2 Compile, Link, and Run* for detailed information on the number of files allowed.

recl is an integer expression which, if included, sets the record length for random files. It can range from 1 through 32767. The default record length is 128 bytes.

OPEN allocates a buffer for I/O to the file or device and determines the mode of access that is used with the buffer.

OPEN Statement

Under DOS mode, the FILES=XX command in your CONFIG.SYS should be set to the number of files you plan to have open simultaneously, plus 3, which BASIC uses.

An OPEN must be run before any I/O can be done to a device or file using any of the following statements, or any statement or function requiring a file number:

PRINT #
PRINT # USING
INPUT #
LINE INPUT #
IOCTL #
WRITE #
INPUT\$
GET #
PUT #

GET and PUT are valid for random files or communications files. A disk file can be either random or sequential, and a printer can be opened in either random or sequential mode; however, all other standard devices can be opened only for sequential operations. See "OPEN "COM... Statement."

BASIC normally adds a line feed after each carriage return (CHR\$(13)) sent to a printer. However, if you open a printer (LPT1:,LPT2:, or LPT3:) as a random file with width 255, this line feed is suppressed.

APPEND initially sets the file pointer to the end of the file, and the record number is set to the last record of the file. PRINT # or WRITE # then extends the file.

A file cannot be opened for sequential output or append if the file is already open.

If a file opened for input does not exist, an error occurs. If a file that does not exist is opened for output, append, or random access, a file is created.

Any values given outside the ranges indicated result in an **Illegal function call** error. The file is not opened.

OPEN

Statement

See the sections on device drivers and Disk I/O in *IBM BASIC Compiler/2 Fundamentals* for a complete explanation of using disk files. See "OPEN "COM... Statement" for information on opening communications files.

Examples

Either of these statements opens the file called "DATA" for sequential output on the default device in the directory called LVL2.

```
10 OPEN "LVL1\LVL2\DATA" FOR OUTPUT AS #1
   or
10 OPEN "0",#1,"LVL1\LVL2\DATA"
```

Either of the next two statements opens the file named "RRFILE" in the LVL1 directory on the disk in drive B: for random input and output. The record length is 256.

```
20 OPEN "B:LVL1\RRFILE" AS 1 LEN=256
   or
20 OPEN "R",B:LVL1\RRFILE:",256
```

Either of the following statements opens the file named "DATA" for sequential output on the default device:

```
10 OPEN "DATA" FOR OUTPUT AS #1
   or
10 OPEN "0",#1,"DATA"
```

In the preceding example, opening for output destroys any existing data in the file. If you do not wish to destroy data, you must open for APPEND.

Either of the following two statements opens the file named "SSFILE" on the disk in drive B: for random input and output. The record length is 256.

```
10 OPEN "B:SSFILE" AS 1 LEN=256
   or
10 OPEN "R",1,"B:SSFILE",256
```

This example opens the file "DATA.ART" on the disk in drive A: and positions the file pointers and that any output to the file is placed at the end of existing data in the file:

OPEN Statement

```
10 FILE$ = "A:DATA.ART"  
20 OPEN FILE$ FOR APPEND AS 3
```

Line 10 in the next example opens the printer in random mode. Because the default width is 80, the lines printed by lines 20 and 30 end with a carriage return/line feed. Line 40 changes the printer width to 255, so the line feed after the carriage return is suppressed. Therefore, the line printed by line 50 ends only with a carriage return and not a line feed. This causes the line printed by line 70 to overprint "This line is underlined", causing the line to be underlined. Line 60 changes the width back to 80 so the underlines and following lines end with a line feed.

```
10 OPEN "LPT1:" AS #1  
20 PRINT #1,"Printing width 80"  
30 PRINT #1,"Now change to width 255"  
40 WIDTH #1,255  
50 PRINT #1,"This line will be underlined"  
60 WIDTH #1,80  
70 PRINT #1, STRING$(28,"_")  
80 PRINT #1,"Printing width 80 with CR/LF"
```

Results:

```
Printing width 80  
Now change to width 255  
This line will be underlined  
Printing width 80 with CR/LF
```

The following example opens a file for input and denies write access to all other processes:

```
10 OPEN "TEST.DAT" FOR INPUT LOCK WRITE AS #1  
20 INPUT #1,A$  
30 CLOSE #1
```

OPEN "COM. . . Statement

Purpose

The OPEN "COM. . . statement opens a communications file for RS-232 asynchronous communication with other computers and peripherals.

Valid only with an Asynchronous Communications Adapter.

This statement is only supported under DOS and OS/2 mode.

Format

```
OPEN "COMn:[speed] [, [parity] [, [data] [, [stop] [,RB[n]] [,TB[n]][,OP[n]]  
[,RS] [,CS[n]] [,DS[n]] [,CD[n]] [,LF] [,PE]]]" AS [#]filenum  
[LEN=number]
```

Comments

n is 1 or 2, indicating the number of the Asynchronous Communications Adapter.

speed is an integer constant specifying the transmit/receive bit rate in bits per second (bps). Valid speeds are 75, 110, 150, 300, 600, 1200, 1800, 2400, 4800, and 9600. The default is 300 bps.

parity is a 1-character constant specifying the parity for transmit and receive as follows:

- S SPACE: Parity bit always transmitted and received a a space (0-bit).
- O ODD: Odd transmit parity; odd receive parity checking.
- M MARK: Parity bit always transmitted and received a a mark (1-bit).
- E EVEN: Even transmit parity, even receive parity checking.

OPEN "COM. . . Statement

- N NONE:** No transmit parity, no receive parity checking
The default is **EVEN (E)**.
- data** is an integer constant indicating the number of transmit/receive data bits. Valid values are: 5, 6, 7, and 8. The default is 7.
- stop** is an integer constant indicating the number of stop bits. Valid values are 1 and 2. The default is 2 stop bits for 75 and 110 bps; **ONE** stop bit for all others. If you use 5 for *data*, a 2 here means 1-1/2 stop bits.
- filenum** is an integer expression that evaluates to a valid file number. The number is then associated with the file for as long as it is open and is used by other communications I/O statements to refer to the file.
- number** is the maximum number of bytes that can be read from the communications buffer when using **GET** or **PUT**. The default is 128 bytes.

The **RB**, **TB**, **OP**, **RS**, **CS**, **DS**, **CD**, **LF**, and **PE** options affect the line communications as follows:

- RB[n]** Controls the size of the receive buffer.
- TB[n]** Controls the size of the transmit buffer.
- OP[n]** Controls the timeout value during the opening of the file
- RS** suppresses **RTS** (Request To Send).
- CS[n]** Controls checking of **CTS** (Clear To Send) signal.
- DS[n]** Controls checking of **DSR** (Data Set Ready) signal.
- CD[n]** Controls checking of **CD** (Carrier Detect) signal.
- LF** sends a line feed following each carrier return.
- PE** enables parity checking for the data that is received.

The **CD** (Carrier Detect) is also known as the **RLSD** (Received Line Signal Detect).

OPEN "COM. . . Statement

Note: The *speed*, *parity*, *data*, and *stop* parameters are positional, but **RB**, **TB**, **OP**, **RS**, **CS**, **DS**, **CD**, **LF**, and **PE** are not.

The **RB** and **TB** options set the sizes of the buffers for the data being received and transmitted, respectively. The argument *n* is the number of bytes to be reserved for the buffer and can range from 0 through 65535. The default for **RB** is the value specified with the */c* switch when compiling. The default for **TB** is 128 bytes.

The **OP** option allows you to specify the time the OPEN "COM... statement should allow for the 'data set ready' **DSR** and /or the **CD** 'Carrier Detect' lines to become active when opening the communications file. *n* is the number of milliseconds (between 1 and 65535) to wait or 0 to specify that no time limit should be enforced. If the **OP** option is specified without *n*, the default is 0. If no **OP** option is specified, the default is ten times the maximum value of the **DS**[*n*] and **CD**[*n*] options.

Note: **DSR** and **CD** are only checked if they are set to be checked during actual communications.

The 'request to send' **RTS** line is turned on when you run an OPEN "COM... statement unless you include the **RS** option.

The *n* argument in the **CS**, **DS**, and **CD** options specifies the number of milliseconds to wait for the signal during communications before returning a **Device timeout** error. The *n* can range from 0 through 65535. If *n* is omitted or is equal to 0, the line status is not checked. The defaults are **CS1000**, **DS1000**, and **CD0**. If **RS** was specified, **CS0** is the default.

Normally, I/O statements to a communications file fail if the 'clear to send' **CTS** or 'Data Set Ready' **DSR** signals are off. The system waits 1 second before returning a **Device timeout**. The **CS** and **DS** options allow you to ignore these lines or to specify the amount of time to wait before the time-out.

Normally 'carrier detect' (**CD** or **RLSD**) is ignored. The **CD** option allows you to test this line by including a non-0 *n* parameter. If *n* is

OPEN "COM. . . Statement

omitted or is equal to 0, carrier detect is not checked at all (which is the same as omitting the **CD** option).

The **LF** parameter is intended for those using communications files to print to a serial line printer. When you specify **LF**, a line feed character (hex 0A) is automatically sent after each carriage return character (hex 0C). (This includes the carriage return sent as a result of the width setting.) **INPUT #** and **LINE INPUT #**, when used to read from a communications file that was opened with the **LF** option, stop when they see a carriage return. The line feed is always ignored.

The **PE** option enables parity checking. The default is no parity checking. The **PE** option causes a **Device I/O error** on parity errors and turns on the high-order bit for 7 or fewer data bits. The **PE** option does *not* affect framing and overrun errors. These errors always turn on the high-order bit and cause a **Device I/O error**.

Any coding errors within the string expression starting with *speed* result in an error. No indication is given as to which parameter is in error.

See "Communications" in *IBM BASIC Compiler/2 Fundamentals* for more information on control of output signals and other communications support.

If you specify eight data bits, you must specify parity **N**. BASIC uses all eight bits in a byte to store numbers, so if you are transmitting or receiving numeric data (for example, by using **PUT**), you must specify eight data bits. (This is not necessary if you are sending numeric data as *text*.)

A communications device can be open to only one number at a time.

Note: If you are using asynchronous communications and want to compile your program with the **/O** switch, you must link the **IBMCOMC.OBJ** module (when running under DOS 3.30) or the **IBMCOMP.OBJ** module (when running under the OS/2 mode) to your program.

OPEN "COM. . . Statement

See also "OPEN Statement" for information on opening devices other than communications devices.

Examples

In this example, file 1 is opened for communication with all defaults. The speed is 300 bps with even parity. There are seven data bits and one stop bit.

```
10 OPEN "COM1:" AS #1
```

In this example, file 2 is opened for asynchronous I/O at 1200 bps; no parity is to be produced or checked; eight-bit bytes are sent and received; and one stop bit is transmitted:

```
10 OPEN "COM2:1200,N,8" AS #1
```

This example opens COM1: at 9600 bps with no parity and eight data bits. CTS, DSR, and CD are not checked.

```
10 OPEN "COM1:9600,N,8,,CS,DS,CD" AS #1
```

This example opens COM1: at 1200 bps with the defaults of even parity, seven data bits, and one stop bit. RTS is sent, CTS is not checked, and **Device timeout** is given if DSR is not seen within 2 seconds when needed during communications. Since no **OP** parameter is specified, OPEN "COM..." will wait 20 seconds to open the communications line before timing out. The commas are required to indicate the position of the *parity*, *start*, and *stop* parameters, even though a value is not specified.

```
10 OPEN "COM1:1200,,,,CS,DS2000" AS #1
```

OPEN "COM. . . Statement

OPEN "COM... can be used with the ON ERROR statement to extend the time allowed for a communications connection to be made. For example, the following program waits for three minutes for the 'carrier detect' line to become active. (Line 20 is set to time-out after 60 seconds on the OPEN and TRIES\$ is set to 3.) After the line is open, 'carrier detect' must be seen within 2 seconds of requesting communications activity. 'clear to send' (CTS) and Data Set Ready (DSR) are not checked at all.

```
10 TRIES=3:ON ERROR GOTO 5000
20 OPEN "COM1:300,N,8,2,0P10000, CS, DS, CD2000" AS #1
30 ON ERROR GOTO 0
.
.
.
5000 TRIES=TRIES-1
5010 IF TRIES=0 THEN
5020     ON ERROR GOTO 0 'give up
5030     PRINT "OPEN failed."
5035 END IF
5040 RESUME
```

The next example shows a typical way to use a communication file to control a serial line printer. The **LF** parameter in the OPEN "COM... statement ensures that lines do not print on top of each other.

```
10 WIDTH "COM1:", 132
20 OPEN "COM1:1200,N,8,,CS10000, DS10000,CD10000,LF" AS #1
```

OPEN "PIPE... Statement

Purpose

The OPEN "PIPE... statement allows you to run a child process and read and write its standard input and output.

This statement is not available in DOS mode.

Format

OPEN "PIPE:*command string*" AS #*filenum*

Comments

command string

is a string expression containing the name of a program or OS/2 command to run, and, optionally, any parameters you are passing to the child process.

filenum is an integer expression whose value is from 1 through 255.

Note: See "Differences Between the Compiler and the Interpreter" in *IBM BASIC Compiler/2 Compile, Link, and Run* for detailed information on the number of files allowed to be open at one time.

A program that runs under a BASIC program is referred to as a child process. See "SHELL Function" and "SHELL Statement" for more information about child processes.

The program you specify in the *command string* (the child process) can write to its standard output as if it were writing to the screen. The BASIC program can then read this data using the INPUT # statement, specifying the *filenum* from the OPEN "PIPE... statement.

The child process can read the data that BASIC prints to the PIPE as if the data came from the keyboard.

OPEN "PIPE... Statement

Examples

The following example prints the operating system DIR command output on the screen.

```
OPEN "PIPE:DIR" AS 1
WHILE NOT EOF(1)
  LINE INPUT #1, A$
  PRINT A$
WEND
```

OPTION BASE Statement

Purpose

The OPTION BASE Statement declares the minimum value for array subscripts.

Format

```
OPTION BASE n
```

Comments

n is 1 or 0.

The OPTION BASE statement must be coded *before* you define or use any arrays. An error occurs if you change the base value when arrays exist. The default base is 0. If the statement:

```
OPTION BASE 1
```

is executed, the lowest value an array subscript can have is 1.

You can use only one OPTION BASE statement per module. Also, the OPTION BASE statement must appear at the module level (that is, it cannot appear within a SUB or FUNCTION definition).

Note: The OPTION BASE statement does not affect arrays whose lower bound is set with the “*min* TO *max*” form of the DIM statement.

OPTION BASE Statement

Examples

```
OPTION BASE 1
DIM A(20)
PRINT LBOUND(A), UBOUND(A)
```

Results:

```
1      20
```

```
OPTION BASE 1
DIM A(15), B(-5 TO 12)

PRINT "The lower bound of A= "; LBOUND(A)
PRINT "The lower bound of B= "; LBOUND(B)

END
```

Results:

```
The lower bound of A = 1
The lower bound of B = -5
```

OUT Statement

Purpose

The OUT Statement sends a byte to a machine output port.

This statement is not available in OS/2 mode.

Format

OUT *n,m*

Comments

n is a numeric expression for the port number, in the range 0 through 65535.

m is a numeric expression for the data to be transmitted, in the range 0 through 255.

See the technical reference book for your computer for a description of valid port numbers (I/O addresses).

OUT is the complementary statement to the INP function. See also "INP Function."

Examples

The following example turns the speaker on and off:

```
100 A=INP(&H61)
110 OUT &H61, A OR 3 : REM Speaker on
120 WHILE INKEY$="" : WEND
130 OUT &H61, A AND NOT 3 : REM Speaker off
140 END
```

Purpose

The PAINT statement fills an area on the screen with the selected color.

This statement is used in graphics mode only.

Format

PAINT (x,y) [, [*paint*] [, [*boundary*] [, [*background*]]]

Comments

(x,y) are the coordinates of a point within the area to be filled in. The coordinates can be given in absolute or relative form. For more information on specifying coordinates, see "Graphics Modes" under "Input and Output" in *IBM BASIC Compiler/2 Fundamentals*. This point is used as a starting point. Points specified outside the limits of the screen are not plotted, and no error occurs.

paint can be a numeric or string expression. It is used to fill a color or pattern in or around a bounded area. When *paint* is a numeric expression, it chooses an attribute from the legal attribute range for the current screen mode.

In SCREEN 1, (medium resolution), *attribute* can range from 0 through 3. In SCREEN 2, (high resolution), *attribute* can be 0 or 1.

The default color attribute for the foreground is the maximum color attribute for that screen mode.

The default color attribute for the background is always 0.

boundary is an integer expression in the legal attribute range of the current screen mode. It defines the attribute for the edges of the figure to be painted.

PAINT

Statement

background is a 1-byte string expression used in paint tiling.

In medium resolution, you can fill inside or around a defined area with any one of four colors from the current palette defined by the `COLOR` statement. Examples of this are filling a red circle with green or surrounding a red circle with green.

`PAINT` begins at the specified starting point and covers an area until it meets the specified boundary attribute. Therefore, `PAINT` must always begin inside the area to be painted. If the specified starting point already has the same attribute as *boundary*, painting stops at that point and appears not to occur. An example of this is plotting a point with `PSET` that has the same attribute as *boundary*, and using the coordinates of that point with the `PAINT` statement.

`PAINT` fills any designated area no matter what the shape of the area; however, the more complex the edges of a figure (jagged edges, for instance), the more stack space `BASIC` uses. Under these circumstances you may want to use the `CLEAR` statement at the beginning of your program to increase the stack space.

The `PAINT` statement allows scenes to be displayed with very few statements.

In the example that follows, the `PAINT` statement in line 30 fills in the box drawn in line 20 with the color represented by the attribute in the current palette:

```
10 SCREEN 1
20 LINE (0,0)-(100,150),2,B
30 PAINT (50,50),1,2
```

The following discussion deals with paint tiling only.

To use paint tiling, the *paint* attribute must be a string expression in the form:

```
CHR$(&Hnn)+CHR$(&Hnn)+CHR$(&Hnn)+...
```

The `CHR$` sequence specifies a bit mask that is one byte wide. When the mask is plotted all the way across and down the designated area

PAINT Statement

defined by *boundary*, a pattern is created rather than a solid color. You design the pattern. The two hexadecimal digits in the CHR\$ expression correspond to eight bits, or one byte. The string expression can contain up to 64 bytes.

The design created by the string expression can be mapped as follows:

```
      x increases -->
      7 6 5 4 3 2 1 0
0,0  x x x x x x x x  Tile byte 0
0,1  x x x x x x x x  Tile byte 1
0,2  x x x x x x x x  Tile byte 2
.
.
.
0,63 x x x x x x x x  Tile byte 63
      (maximum allowed)
```

The tile pattern is repeated uniformly over the area defined by *boundary*. If you do not define an area with *boundary*, the whole screen is your designated area. Each byte of the tile string masks 8 bits along the **x**-axis when plotting points. Each byte of the tile string is rotated as required to align the pattern along the **y** axis. BASIC chooses the particular byte of the pattern to plot, using the formula **y mod tile length**.

PAINT

Statement

The method of designing patterns in each screen varies depending on the number of color attributes available in each screen mode. This is so because the number of bits per pixel is directly related to the number of color attributes available in each screen mode. In any screen, where x is the total number of color attributes for that screen:

$$\text{LOG}_2(X)=Y$$

where Y is the number of bits per pixel. In high resolution, each byte of the string is able to plot eight points across the screen (1 bit per pixel), because $\text{LOG}_2(2)=1$.

In SCREEN 1, one medium-resolution tile byte describes four pixels, because medium resolution has two bits per pixel: that is, $\text{LOG}_2(4)=\text{two bits per pixel}$. Every two bits of the tile byte describes one of four possible color attributes associated with each of the four pixels to be plotted.

Mode	Bits per pixel	Formula
SCREEN 1	2	$\text{LOG}_2(4)$
SCREEN 2	1	$\text{LOG}_2(2)$

Because there is only one bit per pixel in high resolution, a point is plotted at every position in the bit mask that has a value of 1. In high resolution, the screen can be painted with X's using the following example:

```
5 REM Without background tile
10 CLS: SCREEN 1: COLOR 1: KEY OFF
20 LOCATE 12,7:PRINT "I LOVE MY IBM COMPUTER"
30 PAINT(320,100),CHR$(&H81)+CHR$(&H42)+
  CHR$(&H24)+CHR$(&H18)+CHR$(&H18)+
  CHR$(&H24)+CHR$(&H42)+CHR$(&H81)
```

PAINT Statement

The length of this mask is 8, indexed 0 through 7. In this case, PAINT at coordinates (320,100) begins by plotting byte 4. This is calculated using the **y mod tile length** formula by substituting 100 for **y** and 8 for **tile length**. This pattern appears on the screen as:

```
      x increases -->
      7 6 5 4 3 2 1 0
Tile byte 0  1 0 0 0 0 0 1  CHR$(&H81)
Tile byte 1  0 1 0 0 0 0 1  CHR$(&H42)
Tile byte 2  0 0 1 0 0 1 0 0  CHR$(&H24)
Tile byte 3  0 0 0 1 1 0 0 0  CHR$(&H18)
Tile byte 4  0 0 0 1 1 0 0 0  CHR$(&H18)
Tile byte 5  0 0 1 0 0 1 0 0  CHR$(&H24)
Tile byte 6  0 1 0 0 0 0 1 0  CHR$(&H42)
Tile byte 7  1 0 0 0 0 0 0 1  CHR$(&H81)
```

PAINT

Statement

The following figure shows the binary and hexadecimal values associated with each attribute in medium resolution:

Color palette 0	Attrib. in binary	Pattern to draw solid line in binary	Pattern to draw solid line in hexadecimal
green	01	01010101	&H55
red	10	10101010	&HAA
brown	11	11111111	&HFF

Color palette 1	Attrib. in binary	Pattern to draw solid line in binary	Pattern to draw solid line in hexadecimal
cyan	01	01010101	&H55
magenta	10	10101010	&HAA
white	11	11111111	&HFF

In medium resolution, the following example plots a pattern of boxes with a border color of red in palette 0 and magenta in palette 1:

```

10 CLS: SCREEN 1: COLOR 1: KEY OFF
20 LOCATE 12,7:PRINT "I LOVE MY IBM COMPUTER"
30 PAINT (320,100),CHR$(&HAA)+CHR$(&H82)+
  CHR$(&H82)+CHR$(&H82)+CHR$(&H82)+
  CHR$(&H82)+CHR$(&H82)+CHR$(&HAA)

```

Occasionally, you may want to tile over an already painted area that is the same color or pattern as two consecutive bytes in the tile mask. Normally, this constitutes a terminating condition because your point is bounded by points of the same bit pattern. (An example follows on the use of *background*.)

PAINT Statement

You can use the *background* attribute to skip this terminating condition. You cannot specify more than two consecutive lines in the tile pattern that matches this *background* attribute. Doing so causes an **Illegal function call error**.

Examples

The following program demonstrates how to tile an area with three lines of red, two lines of green, and one line of red. The palette then changes to show that the same tile mask yields the same pattern with different colors.

```
10 CLS: SCREEN 1,0:KEY OFF
20 TIL$=CHR$(&HAA)+CHR$(&HAA)+CHR$(&HAA)+CHR$(&H55)+CHR$(&H55)+CHR$(&HFF)
30 COLOR 0,0 'choose palette 0
40 VIEW (1,1)-(150,100),0,2
50 GOSUB 1000
60 COLOR 0,1 'choose palette 1
70 GOTO 1020
1000 PAINT (125,50),TIL$,2
1010 RETURN
```


PAINT Statement

The following example uses paint tiling with the *background* attribute:

```
10 CLS: SCREEN 1:COLOR 0,1:KEY OFF
20 TIL$=CHR$(&H5F)+CHR$(&H5F)+CHR$(&H27)+CHR$(&H81)
30 VIEW (1,1)-(150,100),0,2
40 LOCATE 3,22:PRINT "<---Without back-"
50 LOCATE 4,22:PRINT " ground tile "
60 PAINT (125,50),CHR$(&H5F)
70 PAINT (125,50),TIL$,2
80 '
90 'with background tile'
100 '
110 VIEW (160,100)-(310,198),0,2
120 LOCATE 16,1:PRINT "With background-->"
130 LOCATE 17,1:PRINT "tile chr$(&H5F)"
140 PAINT (125,50),CHR$(&H5F)
150 PAINT (125,50),TIL$,2,CHR$(&H5F)
160 LINE (1,100)-(319,100),3
170 FOR I=1 TO 2500:NEXT I
```

Purpose

The PEEK function returns the byte read from the indicated memory position.

Format

$v = \text{PEEK}(n)$

Comments

n is an integer value in the range 0 through 65535. This is the offset from the current segment as defined by the DEF SEG statement. See "DEF SEG Statement."

The returned value is an integer in the range 0 through 255.

PEEK is the complementary function to the POKE statement. Because the allocation of memory is different, PEEKS and POKES designed for the interpreter *do not* work with the compiler. See "POKE Statement."

For OS/2 users:

In the OS/2 mode, PEEK changes single and double precision address values to 2-byte integers. PEEK treats 2-byte integer address arguments as offsets from the segment described by the current DEF SEG.

PEEK treats the segment as a selector. Illegal memory references may cause exceptions or return a **Permission denied** error.

PEEK

Function

Examples

The following example in a program tests which display adapter is on the system. After line 30 is run, the variable IBMMONO has a value of 0 if the IBM Color/Graphics Monitor Adapter is used, or 1 if the IBM Monochrome Display and Printer Adapter is used.

```
10 'test display adapter
20 DEF SEG=0
30 IF (PEEK(&H410) AND &H30)=&H30 THEN IBMMONO=1 ELSE IBMMONO=0
```

PEN Statement and Function

Purpose

The PEN statement and function Reads the light pen.

This statement is not available in OS/2 mode.

Format

As a statement:

PEN ON

PEN OFF

PEN STOP

As a function:

$v = \text{PEN}(n)$

Comments

The PEN function, $v = \text{PEN}(n)$, reads the light pen coordinates.

n is a numeric expression in the range 0 through 9 and affects the value returned by the function as follows:

- 0** A flag indicating if the pen was down since the last poll. It returns -1 if down, 0 if not.
- 1** Returns the x-coordinate where the pen was last activated.

The range is 0 through 319 in medium resolution, and 0 to 639 in high resolution.

PEN

Statement and Function

- 2 Returns the y-coordinate where the pen was last activated. Range is 0 through 199.
- 3 Returns the current pen switch value. It returns -1 if down, 0 if up.
- 4 Returns the last known valid x-coordinate.
The range is 0 through 319 in medium resolution, and 0 through 639 in high resolution.
- 5 Returns the last known valid y coordinate. The range is 0 through 199.
- 6 Returns the character row position where the pen was last activated. The range is 1 through 24.
- 7 Returns the character column position where the pen was last activated. The range is 1 through 40 or 1 through 80, depending on WIDTH.
- 8 Returns the last known valid character row. The range is 1 through 24.
- 9 Returns the last known valid character column position. The range is 1 through 40 or 1 through 80, depending on WIDTH.

PEN ON enables the PEN read function. The PEN function is initially off. A PEN ON statement must be run before any pen read function calls can be made. A call to the PEN function while the PEN function is off results in an **illegal function call** error.

To improve running speed, turn off the pen with a PEN OFF statement when you are not using the light pen.

Running PEN ON also allows trapping to take place with the ON PEN statement. After PEN ON, if a non-0 line number was specified in the ON PEN statement, every time the program starts a new statement or line number, (depending on whether you compiled using /V or /W), BASIC checks to see if the pen was activated. See "ON PEN Statement."

PEN

Statement and Function

PEN OFF disables the **PEN** read function. No trapping of the pen takes place. Action by the light pen is not remembered even if it does take place.

PEN STOP disables trapping of light pen activity. If activity does occur, however, it is remembered, and an immediate trap occurs when a **PEN ON** is run.

When the pen is down in the border area of the screen, the values returned are inaccurate.

Examples

This example prints the pen value since the last poll, and the current value:

```
10 PEN ON
20 FOR I=1 TO 500
30 X=PEN(0): X1=PEN(3)
40 PRINT X, X1
50 NEXT
60 PEN OFF
```

PLAY Statement

Purpose

The PLAY statement plays music as specified by *string*.

This statement is not available in OS/2 mode.

Format

PLAY *string*

Comments

PLAY implements a concept similar to DRAW by imbedding a “tune definition language” into a character string.

Note: In a compiled program, if the program ends before the buffer is empty, the music stops playing. This is different from the interpreter. If your program is sensitive to this, you can place a dummy FOR..NEXT loop prior to exiting your program.

string is a string expression consisting of single-character or double-character music commands.

The commands in PLAY are:

A to G with optional #, +, or -

Plays the indicated note in the current octave. A number sign (#) or plus sign (+) afterward indicates a sharp; a minus sign (-) indicates a flat. A #, +, or - is not allowed unless it corresponds to a black key on a piano. For example, B# is an invalid note.

O *n* Octave. Sets the current octave for the notes that follow. There are seven octaves, numbered 0 through 6. Each octave goes from C through B. Octave 2 starts with middle C. Octave 4 is the default octave.

PLAY Statement

- > n** Go up to the next higher octave and play note *n*. Each time note *n* is played, the octave goes up, until it reaches octave 6. For example, PLAY ">A" raises the octave and plays note A. Each time PLAY ">A" is run, the octave goes up until it reaches octave 6. Then each time PLAY ">A" is run, note A plays at octave 6.
- < n** Go down one octave and play note *n*. Each time note *n* is played, the octave goes down, until it reaches octave 0. For example, PLAY "<A" lowers the octave and plays note A. Each time PLAY "<A" is run, the octave goes down until it reaches octave 0. Then each time PLAY "<A" is run, note A plays at octave 0.
- N n** Plays note *n*, which can range from 0 through 84. In the 7 possible octaves, there are 84 notes. An *n* 0 means "rest." This is an alternative way of selecting notes besides specifying the octave (O *n*) and the note name (A–G).
- L n** Sets the length of the notes that follow. The actual length of the note is $1/n$. *n* can range from 1 through 64.

Length Equivalent

L1	whole note
L2	half note
L3	one of a triplet of three half notes (1/3 of a four-beat measure)
L4	quarter note
L5	one of a quintuplet (1/5 of a measure)
L6	one of a quarter-note triplet
	.
	.
	.
L64	sixty-fourth note

The length can also follow the note when you want to change only the length of the note. For example, A16 is equivalent to L16A.

PLAY

Statement

P *n* Pause (rest). An *n* can range from 1 through 64, and figures the length of the pause in the same way as L (length).

- Dot or period. When placed after a note, causes the note to be played as a dotted note. A dot increases the duration of a note by half the duration of the note. A note can have more than one dot. Each dot increases the total value of the note by 1/2 the value of the previous dot. For example, a double-dotted halfnote is equivalent in duration to a half note plus a quarter note plus an eighth note. Dots can also appear after a pause (P) to scale the pause length in the same way.

T *n* Tempo. Sets the number of quarter notes in a minute. The *n* can range from 32 through 255. The default is 120. Under "SOUND Statement" is a table listing common tempos and the equivalent beats per minute.

MF Music foreground. Music (created by SOUND or PLAY) runs in foreground. Each subsequent note or sound does not start until the previous note or sound is finished. You can press Ctrl+Break to exit PLAY. Music foreground is the default state.

MB Music background. Music (created by SOUND or PLAY) runs in background instead of in foreground. Each note or sound is placed in a buffer, allowing the BASIC program to continue running while music plays in the background. The music background buffer can hold up to 32 notes at one time.

MN Music normal. Each note plays 7/8 of the time specified by L (length). This is the default setting of **MN**, **ML**, and **MS**.

ML Music legato. Each note plays the full period set by L (length).

MS Music staccato. Each note plays 3/4 of the time specified by L.

XVARPTRS(*variable*)

Run specified string.

In all these commands, the *n* argument can be any integer. It can be a constant such as **12**, or it can be VARPTRS(*variable*), where *variable* is the name of a variable. Any blanks in *string* are ignored.

PLAY Statement

The `VARPTR$` form is the only one that can be used in compiled programs. For example:

```
PLAY "X"+VARPTR$(A$)
PLAY "O="+VARPTR$(I)
```

You can use **X** to store a “subtune” in one string and call it repetitively with different tempos or octaves from another string.

There are two ways you can specify variables in a `PLAY` string for the compiler:

- Use the “X” variable form, concatenated (“+”) with the `VARPTR$` of the variable itself.
- Use the `PLAY` macro, followed by an equal sign (=) and concatenated (“+”) with the `VARPTR$` of the variable itself.

The examples on the following page demonstrate both methods.

Examples

The following example plays a tune:

```
10 'little lamb
20 MARY$="GFE-FGGG"
30 PLAY "MB T100 03 L8 X"+VARPTR$(MARY$) +"P8FFF4"
40 PLAY "GB-B-4 X"+VARPTR$(MARY$)+ "GFFGFE-."
```

The following example plays the scale from octave 0 to octave 6:

```
10 ' Play the scale using > octave
20 SCALE$="CDEFGAB"
30 PLAY "O0 X"+VARPTR$(SCALE$)
40 FOR I=1 TO 6
50 PLAY ">X"+VARPTR$(SCALE$)
60 NEXT
70 ' Play the scale using < octave
80 PLAY "O6 X"+VARPTR$(SCALE$)
90 FOR I=1 TO 6
100 PLAY "<X"+VARPTR$(SCALE$)
110 NEXT
```

PLAY(n) Function

Purpose

The PLAY(N) function returns the number of notes currently in the music background buffer.

This function is not available in OS/2 mode.

Format

$V=PLAY(n)$

Comments

n is a dummy argument that can have any value.

PLAY(n) returns a 0 when the program is running in Music Foreground mode. The maximum value that can be returned is 32, which is the maximum number of notes held in the buffer.

PLAY(n) returns notes in the buffer only when you are using Music Background (MB) mode.

Examples

The following example begins playing a new tune when there are five notes left in the background music buffer:

```
10 'When 5 notes in background buffer
20 'go to line 1000 and play another tune
30 PLAY "MB CDEFGAB"
40 IF PLAY(1)=5 GOTO 1000
50 GOTO 2000
:
1000 PLAY "MB 04 T200 L4 MS GG#GE"
2000 END
```

Purpose

The PMAP function maps physical coordinates to world coordinates or world coordinates to physical coordinates.

This function is used in graphics mode only.

Format

$v = \text{PMAP}(x, n)$

Comments

x coordinate of the point that is to be mapped

n can be a value in the range 0 through 3 such that:

- 0 maps the world coordinate x to the physical coordinate x .
- 1 maps the world coordinate y to the physical coordinate y .
- 2 maps the physical coordinate x to the world coordinate x .
- 3 maps the physical coordinate y to the world coordinate y .

PMAP is used to translate coordinates between the world system as defined by the WINDOW statement and the physical coordinate system.

PMAP($x, 0$) and PMAP($x, 1$) are used to map values from the world coordinate system to the physical coordinate system.

PMAP($x, 2$) and PMAP($x, 3$) are used to map values from the physical coordinate system to the world coordinate system.

PMAP

Function

For example, if the statement

```
SCREEN 1: WINDOW (-1,-1)-(1,1)
```

is in effect you can use PMAP to map the world coordinate points of $(-1,-1)$ and $(1,1)$ to their corresponding physical points on the screen.

PMAP $(-1,0)$ returns the physical x coordinate value of 0.

PMAP $(-1,1)$ returns the physical y coordinate value of 199.

PMAP $(1,0)$ returns the physical x coordinate value of 319.

PMAP $(1,1)$ returns the physical y coordinate value of 0.

The above information tells you that the point $(-1,-1)$, which is in the lower left corner of the screen, corresponds to the physical point $(0,199)$. You also know that the point $(1,1)$, which is in the upper right corner, corresponds to the physical point $(319,0)$.

Examples

The coordinates of the upper-left corner of WINDOW defined in the following statement are $(80,100)$; the coordinates of the lower-right corner are $200,200$.

```
SCREEN 2  
WINDOW SCREEN (80,100) - (200,200)
```

If the physical screen coordinates are $(0,0)$ in the upper-left corner and $(639,199)$ in the lower-right corner, then the following statements return the screen coordinates equivalent to the window coordinates $80,100$.

```
X = PMAP(80,0)           'X = 0  
Y = PMAP(100,1)         'Y = 0
```

PMAP Function

The following statement returns the screen coordinates equivalent to the window coordinates 200,200.

```
X = PMAP(200,0)      'X = 639  
Y = PMAP(200,1)      'Y = 199
```

The following statement returns the window coordinates equivalent to the screen coordinates 639,199.

```
X = PMAP(639,2)      'X = 200  
Y = PMAP(199,3)      'Y = 200
```

POINT Function

Purpose

The first form of the POINT function returns the color attribute of the specified point on the screen. The second form returns the value of the current x or y graphics coordinate.

This function is used in graphics mode only.

Format

$v = \text{POINT}(x,y)$

$v = \text{POINT}(n)$

Comments

(x,y) are the coordinates of the point to be used. They must be in absolute form as explained in "Specifying Coordinates" under "Graphics Modes" in *IBM BASIC Compiler/2 Fundamentals*.

If the point given is out of range, the value -1 is returned.

In medium resolution, valid returns are 0, 1, 2, and 3. In high resolution, they are 0 and 1.

n returns the value of the current x or y graphics coordinate. n can have a value from 0 through 3 where:

0 returns the current physical x-coordinate.

1 returns the current physical y-coordinate.

2 returns the current world x-coordinate if WINDOW is active. If WINDOW is not active, it returns the current physical x-coordinate.

POINT Function

- 3** returns the current world y-coordinate if WINDOW is active. If WINDOW is not active, it returns the current physical y.

See also "WINDOW Statement."

Examples

The following example inverts the current setting of point (I,I):

```
10 SCREEN 2
20 IF POINT(I,I)<>0 THEN PRESET(I,I) ELSE PSET(I,I)
   or
20 PSET(I,I),1-POINT(I,I)
```

This example illustrates values returned by the POINT function. Note the change in the values, depending upon WINDOW.

```
10 CLS: SCREEN 1,0:KEY OFF:DEFINT A-Z
20 PRINT "POINT(n) with WINDOW inactive"
30 GOSUB 110
40 WINDOW (0,0)-(319,199)
50 PRINT "POINT(n) with WINDOW active"
60 GOSUB 110
70 PRINT "POINT(n) with WINDOW and SCREEN active"
80 WINDOW SCREEN (0,0)-(319,199)
90 GOSUB 110
100 END
110 PSET (5,15)
120 FOR I=0 TO 3
130 PRINT INT(POINT (I));
140 NEXT
150 PRINT:PRINT
160 RETURN
```

Results:

```
POINT(n) with WINDOW inactive
5 15 5 15
POINT(n) with WINDOW active
5 184 5 15
POINT(n) with WINDOW and SCREEN active
5 15 5 15
```


POINT

Function

The following example redraws the ellipse drawn with the CIRCLE statement, tilted the specified number of degrees.

```
DEFINT X,Y
INPUT "Angle of tilt in degrees (0 to 90): ",ANG
'Medium resolution screen
SCREEN 1
'Convert degrees to radians
ANG = (3.1415926#/180)*ANG
CS = COS(ANG) : SN = SIN(ANG)
'Draw ellipse
CIRCLE (45,70),50,2,,2
'Paint interior of ellipse
PAINT (45,70),2
FOR Y = 20 TO 120
  FOR X = 20 TO 70
    'Check each point in rectangle enclosing ellipse:
    IF POINT(X,Y) <> 0 THEN
      'If the point is in the ellipse, plot a corresponding
      'point in the "tilted" ellipse:
      XNEW = (X*CS - Y*SN) + 200 : YNEW = (X*SN + Y*CS)
      PSET(XNEW,YNEW),2
    END IF
  NEXT
NEXT
NEXT
END
```

Purpose

The POKE statement writes a byte into a memory location.

Format

POKE *n,m*

Comments

n is an integer value in the range 0 through 65535. It indicates the offset into the current segment where that data is to be written. The current segment is defined by the DEF SEG statement. See “DEF SEG Statement.”

m *m* is the data to be written to the specified location. It must be in the range 0 through 255.

The complementary function to POKE is PEEK. POKE and PEEK are useful for efficient data storage and loading assembly language subprograms. See also “PEEK Function.”

Warning: BASIC does not check the offset specified; therefore, do not poke around in BASIC’s stack, BASIC’s variable area, or your BASIC program.

Because the allocation of memory is different, PEEKS and POKES designed for the interpreter *do not* work with the compiler.

POKE

Statement

For OS/2 users:

In OS/2 mode, POKE changes single and double precision address values to 2-byte integers. POKE treats 2-byte integer address arguments as offsets from the segment described by the current DEF SEG.

POKE treats the segment as a selector. Illegal memory references may cause exceptions or return a **Permission denied** error.

Examples

See “Calling Assembly Language Subprograms” in *IBM BASIC Compiler/2 Fundamentals*.

Purpose

The POS function returns the current cursor column position.

Format

$v = \text{POS}(n)$

Comments

The n is a dummy argument.

The current horizontal (column) position of the cursor is returned. The returned value is in the range 1 through 40 or 1 through 80, depending on the current WIDTH setting.

CSRLIN can be used to find the vertical (row) position of the cursor. See "CSRLIN Variable."

See also "LPOS Function."

Examples

This example prints a carriage return (moves the cursor to the beginning of the next line) if the cursor is beyond position 60 on the screen:

```
IF POS(0)>60 THEN PRINT CHR$(13)
```

PRINT Statement

Purpose

The PRINT statement displays data on the screen.

Format

PRINT [*list of expressions*] [;|,]

Comments

list of expressions

is a list of numeric and/or string expressions, separated by commas, blanks, or semicolons. Any string constants in the list must be enclosed in quotation marks.

If the list of expressions is omitted, a blank line is displayed. If the list of expressions is included, the values of the expressions are displayed on the screen.

Print Positions

The position of each printed item is determined by the punctuation used to separate the items in the list. BASIC divides the line into print zones of 14 spaces each.

In the list of expressions:

- Typing a comma between expressions causes the next value to be printed at the beginning of the next zone.

Note: If the last character of an expression ends at the print zone boundary and the expression is followed by a comma, BASIC skips the next print zone and prints the next expression in the following print zone, thus leaving an empty print zone between the two expressions.

PRINT Statement

- Typing a semicolon causes the next value to be printed immediately after the last value.
- Typing one or more spaces between expressions has the same effect as typing a semicolon.

If a comma, semicolon, or SPC or TAB function ends the list of expressions, the next PRINT statement begins printing on the same line, spacing accordingly. If the list of expressions ends without a comma, semicolon, SPC or TAB function, a carriage return is printed at the end of the line; that is, BASIC moves the cursor to the beginning of the next line.

If the length of the value to be printed exceeds the number of character positions remaining on the current line, the value is printed at the beginning of the next line. If the value to be printed is longer than the defined WIDTH, BASIC prints as much as it can on the current line and continues printing the rest of the value on the next physical line.

Scrolling occurs as described under “Text Mode” in *IBM BASIC Compiler/2 Fundamentals*.

Printed numbers are always followed by a space. Positive numbers are preceded by a space. Negative numbers are preceded by a minus sign. When single-precision numbers can be represented with seven or fewer digits in fixed–point format as accurately as in floating-point format, they are returned in fixed-point or integer format. For example, 10^{-7} is printed as .0000001 and 10^{-8} is printed as 1E–8.

BASIC automatically inserts a carriage return/line feed after printing *width* characters, where *width* is 40 or 80, as defined by the WIDTH statement. This causes two lines to be skipped when you print exactly 40 (or 80) characters, unless the PRINT statement ends in a semicolon (;).

LPRINT is used to print information on the printer. See “LPRINT and LPRINT USING Statements.”

PRINT Statement

Examples

In this example, the commas in the PRINT statement cause each value to be printed at the beginning of the next print zone:

```
10 X=5
20 PRINT X+5, X-5, X*(-5)
```

Results:

```
10          0          -25
```

Here, the semicolon at the end of line 20 causes both PRINT statements to be printed on the same line:

```
10 INPUT X
20 PRINT X;"SQUARED IS";X^2;"AND";
30 PRINT X;"CUBED IS";X^3
```

Assume that you input 9.

Results:

```
9 SQUARED IS 81 AND 9 CUBED IS 729
```

PRINT USING Statement

Purpose

The PRINT USING statement prints strings or numbers using a specified format.

Format

PRINT USING *v\$*; *list of expressions* [;|,]

Comments

v\$ is a string constant or variable that consists of special formatting characters. These formatting characters determine the field and the format of the printed strings or numbers.

list of expressions

consists of the string or numeric expressions that are to be printed, separated by semicolons or commas.

With PRINT USING, many common formatting tasks are simplified, such as aligning decimal points in numeric output. There are separate formatting characters for string and numeric data. Descriptions of each special formatting character and examples of their use are contained in the following pages.

String Fields

When PRINT USING is used to print strings, one of three formatting characters can be used to format the string field:

! Specifies that only the first character in the given string is to be printed.

PRINT USING Statement

- `\n spaces\` Specifies that $2 + n$ characters from the string are to be printed. If the backslashes are typed with no spaces, two characters are printed; with one space, three characters are printed, and so on. If the string is longer than the field, the extra characters are ignored. If the string is shorter than the field, the string is left-justified in the field and padded with spaces on the right.
- `&` Specifies a variable-length string field. When the field is specified with `&`, the string is output exactly as input.

Examples

This example shows how to use `!` and `\ \` to print string fields:

```
10 A$="LOOK":B$="OUT"  
20 PRINT USING "!" ;A$;B$  
30 PRINT USING "\ \ " ;A$;B$
```

Results:

```
LO  
LOOKOUT
```

This example demonstrates the use of `&` in conjunction with `!`:

```
10 A$="LOOK": B$="OUT"  
20 PRINT USING "!" ;A$;  
30 PRINT USING "&" ;B$
```

Results:

```
LOUT
```

Numeric Fields

When `PRINT USING` is used to print numbers, the following special characters can be used to format the numeric field:

- `#` A number sign is used to represent each digit position. Digit positions are always filled. If the number to be printed has fewer digits than positions specified, the number is right-justified (preceded by spaces) in the field.

PRINT USING Statement

- . A decimal point can be inserted at any position in the field. If the format string specifies that a digit is to precede the decimal point, the digit is always printed (as 0 if necessary). Numbers are rounded as necessary.
- + A plus sign at the beginning or end of the format string causes the sign of the number (plus or minus) to be printed before or after the number.
- A minus sign at the end of the format field causes negative numbers to be printed with a trailing minus sign.
- ** A double asterisk at the beginning of the format string causes leading spaces in the numeric field to be filled with asterisks. The ** also specifies positions for two more digits.
- \$\$ A double dollar sign causes a dollar sign to be printed to the immediate left of the formatted number. The \$\$ specifies two more digit positions, one of which is the dollar sign. The exponential format cannot be used with \$\$.
- **\$ The **\$ at the beginning of a format string combines the effects of the above two symbols. Leading spaces are filled with asterisks, and a dollar sign is printed before the number. The **\$ specifies three more digit positions, one of which is the dollar sign.
- ,
- ^ A comma at the left of the decimal point in a formatting string prints a comma left of every third digit left of the decimal point. A comma at the end of the format string is printed as part of the string. A comma specifies another digit position. The comma has no effect if used with the exponential (^ ^ ^ ^) format.
- ^ You can place four or five carets after the digit position characters to specify exponential format. Four carets allow space for $E\pm nn$ or $D\pm nn$ to be printed. Five carets allow space for numbers in IEEE format, which can have a 3-digit exponent: $E\pm nnn$ or $D\pm nnn$. Any decimal point position can be specified. The significant digits are left-justified, and the exponent is adjusted. Unless a leading + or trailing + or - is specified, one digit position is used

PRINT USING

Statement

to the left of the decimal point to print a space or a minus sign.

– An underscore in the format string causes the next character to be output as a literal character.

The literal character itself can be an underscore by placing two underscores “__” in the format string.

If the number to be printed is larger than the specified numeric field, a percent sign (%) is printed in front of the number. If rounding causes the number to exceed the field, the percent sign is printed in front of the rounded number.

If the number of digits specified exceeds 24, an **Illegal function call** error occurs.

Examples

Example 1: Using spaces

In this example, three spaces are inserted at the end of the format string to separate the printed values on the line:

```
PRINT USING "###.##  ";10.2,5.3,66.789,.234
```

Results:

```
10.20    5.30    66.79    0.23
```

Example 2: Using a +.

```
PRINT USING "+##.##  ";-68.95,2.4,55.6,-.9
```

Results:

```
-68.95    +2.40    +55.60    -0.90
```

Example 3: Using a –.

```
PRINT USING "##.##-  ";-68.95,22.449,-7.01
```

PRINT USING Statement

Results:

68.95- 22.45 7.01-

Example 4: Using a **.

PRINT USING "***#.## ";12.39,-0.9,765.1

Results:

*12.4 *-0.9 765.1

Example 5: Using a \$\$.

PRINT USING "\$\$###.## ";456.78,0.9,-765.1

Results:

\$456.78 \$0.90 -\$765.10

Example 6: Using a **\$.

PRINT USING "***\$#.##";2.34

Results:

***\$2.34

PRINT USING Statement

Example 7: Using a comma.

```
PRINT USING "####,##";1234.5
```

Results:

```
1,234.50
```

```
PRINT USING "####.##,";1234.5
```

Results:

```
1234.50,
```

Example 8: Using a ^.

```
PRINT USING ".###^~~~";-88888
```

Results:

```
0.889E+05-
```

Example 9: Using a _.

```
PRINT USING "_!##.##_!";12.34
```

Results:

```
!12.34!
```

PRINT USING Statement

Example 10: Using a number too large for the field.

```
PRINT USING "##.##";111.22
```

Results:

```
%111.22
```

Example 11 Using a number too large for the field.

```
PRINT USING ".##";.999
```

Results:

```
%1.00
```

Example 12: Using a string constant with a numeric field.

```
PRINT USING "THIS IS EXAMPLE _###"; 12
```

Results:

```
THIS IS EXAMPLE #12
```

PRINT # and PRINT # USING Statements

Purpose

The PRINT # AND PRINT # USING statements write data sequentially to a file.

Format

PRINT #*filenum*,[USING *x\$*;*list of expressions*

Comments

filenum is the number used when the file was opened for output.

x\$ is a string expression comprised of formatting characters as described in the PRINT USING statement.

list of expressions is a list of the numeric and/or string expressions that are written to the file.

PRINT # does not compress data in the file. An image of the data is written to the file just as it would be displayed on the screen with a PRINT statement. For this reason, care should be taken to delimit the data in the file, so that it is input correctly from the file.

In the list of expressions, numeric expressions should be delimited by semicolons. For example:

```
PRINT #1,A;B;C;X;Y;Z
```

(If commas are used as delimiters, the extra blanks inserted between print fields are also written to the file.)

String expressions must be separated by semicolons in the list. To format the string expressions correctly in the file, use explicit delimiters in the list of expressions.

PRINT # and PRINT # USING Statements

For example, let A\$="CAMERA" and B\$="93604-1". The statement:

```
PRINT #1,A$;B$
```

writes CAMERA93604-1 to the file. Because there are no delimiters, this could not be input as two separate strings. To correct the problem, insert explicit delimiters into the PRINT # statement as follows:

```
PRINT #1,A$;" ";B$
```

The image written to the file is:

```
CAMERA,93604-1
```

which can be read back into two string variables.

If the strings themselves contain commas, semicolons, significant leading blanks, carriage returns, or line feeds, write them to the file surrounded by explicit quotation marks using CHR\$(34).

For example, let A\$="CAMERA, AUTOMATIC" and B\$="93604-1". The statement:

```
PRINT #1,A$;B$
```

writes the following image to the file:

```
CAMERA, AUTOMATIC93604-1
```

and the statement:

```
INPUT #1,A$,B$
```

puts the string "CAMERA" into A\$ and "AUTOMATIC93604-1" into B\$.

PRINT # and PRINT # USING Statements

To separate these strings properly in the file, write double quotes to the file image using CHR\$(34). The statement:

```
PRINT #1,CHR$(34);A$;CHR$(34);CHR$(34);B$;CHR$(34)
```

writes the following image to the file:

```
"CAMERA, AUTOMATIC" "93604-1"
```

and the statement:

```
INPUT #1,A$,B$
```

inputs "CAMERA, AUTOMATIC" to A\$ and "93604-1" to B\$.

The PRINT # statement can also be used with the USING option to control the format of the file. For example:

```
PRINT #1,USING"$###.##,";J;K;L
```

Examples

Because data written to the file contains a dollar sign, use string variables to read them back, as in this example:

```
10 A=123
20 B=6789
30 C=22.33
40 OPEN "DATA" FOR OUTPUT AS #1
50 PRINT #1,USING "$###.##,";A;B;C
60 CLOSE
70 OPEN "DATA" FOR INPUT AS #1
80 INPUT #1,A$,B$,C$
90 CLOSE
100 PRINT A$,B$,C$
```

Results:

```
$123.00      $6789.00      $22.33
```

PSET and PRESET Statements

Purpose

The PSET AND PRESET statements draw a point at the specified position on the screen.

Graphics mode only.

Format

PSET (*x,y*) [,*attribute*]

PRESET (*x,y*) [,*attribute*]

Comments

(*x,y*) are the coordinates of the point to be set. They can be in absolute or relative form, as explained in "Specifying Coordinates" under "Graphics Modes" in *IBM BASIC Compiler/2 Fundamentals*.

attribute is an integer expression that chooses an attribute from the attribute range for the current screen mode. In SCREEN 1, (medium resolution), *attribute* can range from 0 through 3. In SCREEN 2, (high resolution), *attribute* can be 0 or 1.

The default color attribute for the foreground is the maximum color attribute for that screen mode.

The default color attribute for the background is always 0.

PRESET is almost identical to PSET. The only difference is that if no *attribute* parameter is given to PRESET, the background attribute (0) is selected. If *attribute* is included, PRESET is identical to PSET. Line 70 in the example shown can be:

```
70 PSET(I,I),0
```

PSET and PRESET Statements

Out-of-range coordinates are clipped, and no error occurs.

Examples

Lines 20 – 40 of this example draw a diagonal line from the point (0,0) to the point (100,100). Lines 60 – 80 erase the line by setting each point to a color of 0.

```
10 CLS: SCREEN 1:KEY OFF
20 FOR I=0 TO 100
30 PSET (I,I)
40 NEXT
50 'erase line
60 FOR I=100 TO 0 STEP -1
70 PRESET(I,I)
80 NEXT
```

PUT Statement (Files)

Purpose

The PUT statement writes a record from a random buffer to a random file.

Format

```
PUT [#]filenum [, [number][,id]]
```

Comments

filenum is the number under which the file was opened.

number is the record number for the record to be written, in the range 1 through 2147483647.

id is any BASIC record variable. You cannot use *id* if a FIELD statement is active on the file.

If you do not specify *number*, the record has the next available record number (after the last PUT).

If you specify *id*, PUT transfers data from the specified record number to the variable *id*. If *id* is smaller than the *id* size, then BASIC skips to the start of the next record in the file before transferring any other data.

PRINT #, PRINT # USING, WRITE #, LSET, and RSET can be used to put characters in the random file buffer before a PUT statement. In the case of WRITE #, BASIC pads the buffer with spaces up to the carriage return.

Any attempt to read or write past the end of the buffer causes a **Field overflow** error. See also "BASIC Disk Input and Output" in *IBM BASIC Compiler/2 Fundamentals*.

PUT

Statement (Files)

Because DOS blocks as many records as possible in 512-byte sectors, the PUT statement does not necessarily perform a physical write to the disk for each record. See *IBM BASIC Compiler/2 Language Reference* for more information on buffers and the CONFIG.SYS file.

PUT can be used for a communications file. In that case *number* is the number of bytes to write to the communications file. This number must be less than or equal to the value set by the LEN option on the OPEN "COM..." statement.

Random files in BASIC have fixed-length records. The requested record number in a GET or PUT statement is multiplied by this fixed record length to form a 31-bit product. This value is then used to move the random file pointer and then write the desired record. Other record-number restrictions are:

- The largest record number possible is 214748364716, so the largest record number available is:

$$2147483647 / \text{record_length}$$

- File size is limited by the available disk space.

Note: The IBM BASIC Compiler/2 stores floating-point data in random files differently than the BASIC Interpreter and previous versions of the BASIC Compiler. See "Floating Point Data in Random Files" under "Disk Data Files-Sequential and Random I/O" for more information.

Examples

See "BASIC Disk Input and Output" in *IBM BASIC Compiler/2 Fundamentals*.

PUT Statement (Graphics)

Purpose

The PUT statement plots images on a specified area of the screen.

Graphics mode only.

Format

PUT (*x,y*), *arrayname* [(*index*)] [,*action*]

Comments

(x,y) are the coordinates of the top left corner of the image to be transferred.

arrayname is the name of a numeric array containing the information to be transferred. For more information on this array, see also "GET Statement (Graphics)."

index describes the starting location of the information stored within the array. If you do not specify an index, BASIC assumes that the data starts at the beginning of the array.

action is one of:

PSET
PRESET
XOR
OR
AND

XOR is the default.

PUT is the opposite of GET in the sense that it takes data out of the array and puts it on the screen. However, it also provides the option of interacting with the data already on the screen.

PUT

Statement (Graphics)

PSET stores the data from the array onto the screen, so this is the true opposite of GET.

PRESET is the same as PSET except that a complementary image is produced. For example, in medium resolution, which has a maximum attribute of 3, an attribute of 0 in the array causes the corresponding point to be plotted with an attribute of 3, and vice versa; an attribute of 1 in the array causes the corresponding point to be plotted with an attribute of 2, and vice versa.

AND, OR, and XOR specify the logical operations on the bits of each image.

AND is used to display selected parts of an image. Only those areas where some image already exists are shown.

OR is used to superimpose the transferred image onto the existing image.

XOR is a special mode that can be used for animation. Its unique property is that when an image is PUT against a complex background *twice*, the background is restored unchanged. This allows you to move an object around without obliterating the background.

In medium resolution mode, AND, OR, and XOR have the following effects on color:

PUT Statement (Graphics)

AND

Screen Color	Array Value			
	0	1	2	3
0	0	0	0	0
1	0	1	0	1
2	0	0	2	2
3	0	1	2	3

PUT
Statement (Graphics)

OR

Screen Color	Array Value			
	0	1	2	3
0	0	1	2	3
1	1	1	3	3
2	2	3	2	3
3	3	3	3	3

XOR

Screen Color	Array Value			
	0	1	2	3
0	0	1	2	3
1	1	0	3	2
2	2	3	0	1
3	3	2	1	0

PUT Statement (Graphics)

Animation of an object can be performed as follows:

1. PUT the object on the screen (with XOR).
2. Recalculate the new position of the object.
3. PUT the object on the screen (with XOR) a second time at the old location to remove the old image.
4. Repeat step 1, this time putting the object at the new location.

Movement done this way leaves the background unchanged. Flicker can be reduced by minimizing the time between steps 4 and 1, and making sure there is enough time delay between steps 1 and 3. If more than one object is being animated, each object should be processed individually, one step at a time.

If it is not important to preserve the background, animation can be performed using the PSET action verb. But you should remember to have an image area that contains the “before” and “after” images of the object. This way the extra area erases the old image. This method can be somewhat faster than the method using XOR described above, since only one PUT is required to move an object (although you must PUT a larger image).

If the image to be transferred is too large to fit on the screen, an **Illegal function call** error occurs.

PUT

Statement (Graphics)

Examples

This example shows how to move a circle across the screen with XOR:

```
10 CLS:DEFINT A-Z: SCREEN 1:KEY OFF
20 DIM A(404)
30 CIRCLE (160,100), 20,3
40 PAINT (160,100),2,3
50 GET (140,80)-(180,120),A:CLS
60 X=30: Y=50
70 FOR I=1 TO 20
80 PUT (X,Y),A,XOR
90 PUT (X,Y),A,XOR
100 X=X + 10
110 NEXT
```

RANDOMIZE Statement

Purpose

The RANDOMIZE statement reseeds the random number generator.

Format

RANDOMIZE [*n*]

RANDOMIZE TIMER

Comments

n is an integer, single-precision, or double-precision expression that is used as the random-number seed.

If *n* is omitted, BASIC suspends the program and asks for a value by displaying:

Random-number seed (-32768 to 32767)?

before executing RANDOMIZE.

If the random-number generator is not reseeded, the RND function returns the same sequence of random numbers each time the program is run. To change the sequence of random numbers every time the program is run, place a RANDOMIZE statement at the beginning of the program and change the seed with each run.

The internal clock can be a useful way to get a random-number seed. You can use VAL to change the last two digits of TIME\$ to a number, and then use that number for *n*.

You can get a new random-number seed without being prompted. To do this, use the TIMER function in the expression.

RANDOMIZE

Statement

Examples

This example uses `TIMER`. Each time the program is run you see a different sequence of numbers.

Note: The values you receive may be different.

```
10 RANDOMIZE TIMER
20 FOR I=1 TO 4
30 PRINT RND;
40 NEXT
```

Results:

First time

```
.9590051 .1036786 .1464037 .7754918
```

Second time

```
.8261163 .17422 .9191545 .5041142
```

The following example demonstrates how different values for the random-number seed produce different number sequences:

```
10 RANDOMIZE
20 FOR i=1 TO 4
30 PRINT RND;
40 NEXT I
```

RANDOMIZE Statement

Results:

Random-number seed (-32768 to 32767)?

Assume that you respond with 3.

Random-number seed (-32768 to 32767)? 3
.5074723 .209742 .3667878 .2915855

Assume that you run the program again and respond with 4.

Random-number seed (-32768 to 32767)? 4
.7730515 .6161293 .5382508 .6053215

If you try 3 again, you'll get the same sequence as the first run:

Random-number seed (-32768 to 32767)? 3
.5074723 .209742 .3667878 .2915855

READ Statement

Purpose

The READ statement reads values from a DATA statement and assigns them to variables. See the DATA statement.

Format

READ *variable* [,*variable*]...

Comments

variable is a numeric or string variable or array element that is to receive the value read from the DATA table.

A READ statement must always be used with a DATA statement. READ statements assign DATA statement values to the variables in the READ statement on a one-to-one basis.

READ statement variables can be numeric or string, and the values that are read must agree with the variable types specified. If they do not agree, an error results.

A single READ statement can access one or more DATA statements (they are accessed in order), or several READ statements can access the same DATA statement. If the number of variables in the list of variables exceeds the number of elements in the DATA statement(s), an **Out of data** error occurs. If the number of variables specified is fewer than the number of elements in the DATA statement(s), subsequent READ statements begin reading data at the first unread element. If there are no subsequent READ statements, the extra data is ignored.

To reread data from any line in the list of DATA statements, use the RESTORE statement. See "RESTORE Statement."

READ Statement

Examples

This program segment reads the values from the DATA statements into the array A. After running, the value of A(1) is 3.08, the value of A(2) is 5.19 and so on.

```
10 FOR I=1 TO 10
20 READ A(I)
30 NEXT I
40 DATA 3.08,5.19,3.12,3.98,4.24
50 DATA 5.08,5.55,4.00,3.16,3.37
```

This program reads string and numeric data from the DATA statement in line 30. Note that you do not need quotation marks around "COLORADO" because it does not have commas, semicolons, or significant leading or trailing blanks. However, you do need the quotation marks around "DENVER," because of the comma.

```
10 PRINT "CITY", "STATE", " ZIP"
20 READ C$,S$,Z
30 DATA "DENVER,", COLORADO, 80211
40 PRINT C$,S$,Z
```

Results:

CITY	STATE	ZIP
DENVER,	COLORADO	80211

REDIM

Statement

Purpose

The REDIM statement changes the space allocated to an array that has been declared dynamic.

Format

REDIM [SHARED] *variable(subscripts)* [AS *type*] [,*variable(subscripts)*][AS *type*]]...

Comments

- SHARED** allows you to share simple variables and arrays among all subprograms in a module.
- variable*** is the name of an array that you want to redimension. It can be up to 40 characters in length.
- subscripts*** define the new dimensions of the array. The *subscripts* can be in two forms:
- (*exp*[,*exp*]...), where *exp* is a numeric expression that defines the upper bound of the dimension. The lower bound is implicitly defined by the OPTION BASE statement.
 - (*min* TO *max*[,*min* TO *max*]...), where *min* and *max* are numeric expressions that you use to explicitly define the upper and lower bounds of each dimension in the array.
- type*** is one of the following:
- INTEGER
 - LONG
 - SINGLE
 - DOUBLE
 - STRING [**bytecount*]

REDIM Statement

- *typename*
typename must have been defined in a previous TYPE statement.

The REDIM statement changes the space allocated to an array that has been declared dynamic. Static arrays *cannot* appear in a REDIM statement.

The SHARED attribute allows arrays to be shared globally by the main program and all subprograms within a module. To declare all variables as global variables, place the word "SHARED" directly after the word "REDIM." This differs from the SHARED statement. The SHARED statement only affects variables within a *single* subprogram. For more information, see "SHARED Statement."

You can specify the range of subscripts in two ways. The first way is to specify the range of each subscript with a single integer, variable, or expression. In this case, the maximum value of the subscript is the number you specify. BASIC assumes that the minimum value of the subscript is 0, unless you use the OPTION BASE statement. See the OPTION BASE statement.

For example, you could redimension an array with the following statement:

```
REDIM A(4,I+2)
```

The second way to specify the range of subscripts is to specify explicitly both the minimum and maximum values for the subscripts. For example, you could redimension an array with the following statement:

```
REDIM B(-2 TO 2,10 TO 13)
```

The number of subscripts must be the same in the redimensioned array as in the original array. The following example demonstrates an illegal REDIM:

```
5 REM $DYNAMIC  
10 DIM MONTH$(12), TME(12,60)  
20 REDIM MONTH$(12), TME(12,60,60)
```

REDIM Statement

The preceding example is illegal because the array TME in line 20 has three dimensions declared, but the original array has two dimensions declared.

When a REDIM statement is compiled, all the array declarations in the statement are treated as dynamic arrays. At runtime, when a REDIM statement is run, the array is deallocated if it is currently allocated and is then reallocated with the new dimensions. Old array element values are lost.

Examples

This example demonstrates the use of static and dynamic arrays within the same program:

```
120 ' $STATIC
130 DIM C(5,5)
140 C(5,5)=1
145 C=5
150 ERASE C
160 PRINT C,C(5,5)
180 ' $DYNAMIC
190 DIM A(20,20,20)
200 I = 40
210 DIM B(I,1)
220 ' ASSIGN VALUES INTO A AND B
223 A(1,1,1)=3
225 B(1,1) = 17
227 ' ERASE AND REDIMENSION A
230 ERASE A
240 REDIM A(5,5,5)
260 PRINT B(1,1),A(1,1,1)
270 END
```

Results:

```
5      0
17     0
```

The following program fragment shows a subroutine that deletes a record from a random-access file. This subroutine uses REDIM to allocate a temporary string array (STORE\$) to hold the records from STOCK.DAT.

REDIM Statement

After the records are stored in STORE\$, STOCK.DAT is closed, deleted, then reopened, and the values in STORE\$ are put back into the file. The ERASE statement then deallocates STORE\$.

```
GOSUB OPENFILE
TOTAL% = LOF(1)/50
MAIN:
  PRINT "1) Add a record"
  PRINT "2) Update a record"
  PRINT "3) Delete a record"
  PRINT "4) End program"
  PRINT : INPUT "What is your choice"; BRANCH
  ON BRANCH GOSUB ADDAREC, UPDATEREC, DELETEREC, ENDPROG
  GOTO MAIN

'Open STOCK.DAT and allocate random-file buffers
OPENFILE:
  OPEN "STOCK.DAT" FOR RANDOM AS #1 LEN=50
'Multiply-defined field
  FIELD #1, 50 AS RECORD$
  FIELD #1, 10 AS PART$, 35 AS DESC$, 5 AS QTY$
RETURN
DELETEREC:
  INPUT "Record number to delete: ",REC%
  GET #1, REC% : PRINT DESC$
  INPUT "Is this the correct record"; CH$
  IF CH$ <> "y" THEN GOTO DELETEREC
'Put the last record where the deleted record was
  GET #1, TOTAL%
  PUT #1, REC%
  TOTAL% = TOTAL% - 1
'Allocate temporary array
  REDIM STORE$(TOTAL%)
'Store STOCK.DAT in array
  FOR I% = 1 TO TOTAL%
    GET #1, I%
    STORE$(I%) = RECORD$
  NEXT
'Erase STOCK.DAT
  CLOSE #1 : KILL "STOCK.DAT"
'Reopen STOCK.DAT
  GOSUB OPENFILE
'Put STORE$ array values in STOCK.DAT
  FOR I% = 1 TO TOTAL%
    LSET RECORD$ = STORE$(I%)
    PUT #1, I%
  NEXT
'Erase array STORE$
  ERASE STORE$
RETURN
```

REM Statement

Purpose

The REM statement inserts explanatory remarks in a program, or inserts compiler metacommands.

Format

REM *remark*|*metacommand*

Comments

remark can be any sequence of characters.

metacommand

is the name of one of the special commands that control the operation of the compiler. All compiler metacommands begin with the \$ (dollar sign) character. See "Compiler Metacommands" for more information.

REM statements that do not contain compiler metacommands are not run but are displayed when the program is listed exactly as they were entered.

REM statements can be branched into (from a GOTO or GOSUB statement), and the program continues with the first executable statement after the REM statement.

Remarks can be added by preceding the remark with a single quotation mark instead of :REM. If you put a remark on a line with other BASIC statements, the remark must be the *last* statement on the line.

You cannot use the single quote (') to add comments at the end of a DATA statement. If you do, BASIC treats it as part of a string. You can, however, use REM to add a remark.

REM Statement

Examples

This example shows the two ways to insert remarks in a program:

```
10 'calculate average velocity
20 SUM=0: REM initialize SUM
30 FOR I=1 TO 20
40 SUM=SUM + V(I)
:
```

Line 20 might also be written:

```
20 SUM=0 ' initialize SUM
```

The following example includes a compiler metacommand:

```
1000 REM $LINESIZE: 120
```

RESET

Command

Purpose

The RESET command closes all disk files and clears the system buffer.

Format

RESET

Comments

If all open files are on disk, RESET is the same as CLOSE with no file numbers after it.

RESTORE Statement

Purpose

The RESTORE statement allows DATA statements to be reread from a specified line or label.

Format

```
RESTORE [line|label]
```

Comments

line is the line number of a DATA statement in the program.

label is a sequence of 1 through 40 letters, digits, or periods, in any combination.

After a RESTORE statement is run, the next READ statement accesses the first item in the first DATA statement in the program. If *line* or *label* is specified, the next READ statement accesses the first item in the specified DATA statement.

Examples

In this example, the RESTORE statement in line 20 resets the DATA pointer to the beginning so that the values that are read in line 30 are 57, 68, and 79:

```
10 READ A,B,C
20 RESTORE
30 READ D,E,F
40 DATA 57, 68, 79
50 PRINT A;B;C;D;E;F
```

Results:

```
57 68 79 57 68 79
```

RESUME

Statement

Purpose

The RESUME statement continues running a program after an error recovery procedure.

Format

RESUME [0]

RESUME NEXT

RESUME *line*|*label*

Comments

Any of the formats shown previously can be used, depending on where running is to resume:

RESUME or RESUME 0

The program resumes at the statement that caused the error.

RESUME NEXT

The program resumes at the statement immediately following the one that caused the error.

RESUME *line*

The program resumes at the specified line number.

RESUME *label*

The program resumes at the specified label. A label is a sequence of 1 through 40 letters, digits, or periods, in any combination.

The *line* and *label* must be at the main program level; they cannot be in a subprogram or function.

RESUME Statement

A RESUME statement that is not in an error trap routine causes a **RESUME without error** message to occur.

Note: If your program contains any ON ERROR or RESUME statements, you need to compile using the /X or /E parameter. See “Compiler Parameters” in *IBM BASIC Compiler/2 Compile, Link, and Run* for more information.

Examples

In this example, line 1000 is the beginning of the error trapping routine. The RESUME statement causes the program to return to line 80 when error 230 occurs in line 90.

```
10 ON ERROR GOTO 1000
.
.
.
1000 IF (ERR=230)AND(ERL=90) THEN PRINT "TRY AGAIN": RESUME 80
```

RETURN

Statement

Purpose

The RETURN statement stops a subroutine and returns to the main program. See also "GOSUB Statement."

Format

RETURN [*line*|*label*]

Comments

line is the number of the program line you wish to return to.

label is the label you wish to return to. A label is a sequence of 1 through 40 letters, digits, or periods, in any combination.

The *line* and *label* must be at the main program level; they cannot be in a subprogram or function.

Although you can use RETURN *line*|*label* to return from any subroutine, this enhancement was added to allow nonlocal returns from the event trapping routines. From one of these routines you will often want to go back to a specific line; while eliminating the GOSUB entry the trap created. The nonlocal RETURN must be used with care, however, since any other GOSUB, WHILE, or FOR statements active at the time of the trap remain active.

Examples

```
10 PRINT "Calling the subroutine..."
20 GOSUB 50
30 PRINT: PRINT "We're back." :END
40 '
50 'Subroutine
60 PRINT: PRINT "We're in the subroutine now."
70 PRINT "Returning from the subroutine..."
80 RETURN
90 PRINT "This line will never be executed."
```

RIGHT\$ Function

Purpose

The RIGHT\$ function returns the rightmost n characters of string x .

Format

$v = \text{RIGHT}\$(x,n)$

Comments

x is any string expression.

n is an integer expression in the range 0 through 32767 that specifies the number of characters to be in the result.

If n is greater than or equal to $\text{LEN}(x)$, x is returned. If n is 0, the null string (length zero) is returned.

See also "MID\$ Function and Statement" and "LEFT\$ Function."

Examples

In this example, the rightmost seven characters of the string A are returned:

```
10 A$="BOCA RATON, FLORIDA"  
20 PRINT RIGHT$(A$,7)
```

Results:

```
FLORIDA
```

RMDIR

Command

Purpose

The RMDIR command removes a directory from the specified disk.

Format

RMDIR *path*

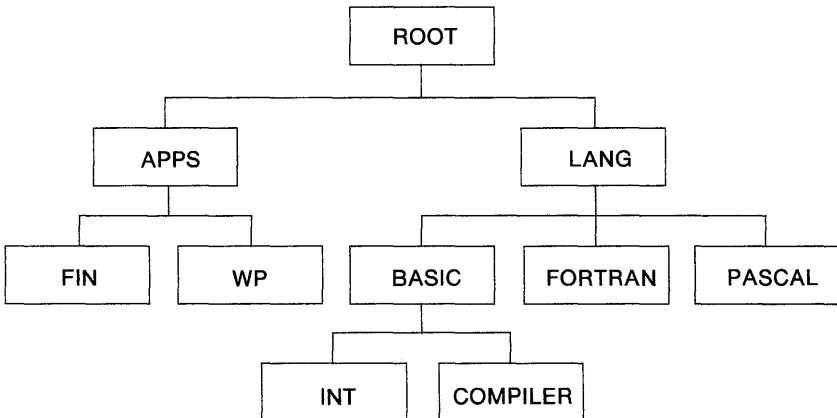
Comments

path is a string expression, not exceeding 63 characters, that identifies the subdirectory to be removed from the existing directory. See also "File Specification" and "Tree-Structured Directories" in *IBM BASIC Compiler/2 Fundamentals* for more information.

The directory must be empty of all files and subdirectories, with the exception of "." and "..", before it can be removed. Otherwise, a **Path/file access** error occurs.

Examples

The following examples refer to the tree structure shown here:



RMDIR Command

If you are in the root directory and you want to remove the directory called WP, use:

```
RMDIR "APPS\WP"
```

If you want to make LANG the current directory and remove the directory called FORTRAN, use:

```
CHDIR "LANG"  
RMDIR "FORTRAN"
```

Another way to remove the directory FORTRAN is to make the root the current directory and then remove FORTRAN:

```
CHDIR "\"  
RMDIR "LANG\FORTRAN"
```

The directory preceding the current directory cannot be removed. Using the same tree structure, suppose that FIN is the current directory. If you try to remove the APPS directory, you get a **Path/file access** error.

RND

Function

Purpose

The RND function returns a random number between 0 and 1.

Format

$v = \text{RND}[(x)]$

Comments

x is a numeric expression that affects the returned value as described here.

The same sequence of random numbers is generated each time the program is run unless the random-number generator is reseeded. Reseeding is most easily done by using the RANDOMIZE statement. See "RANDOMIZE Statement."

You can also reseed the generator when you call the RND function by using x where x is negative. This always generates the particular sequence for the given x . This sequence is not affected by RANDOMIZE, so if you want it to generate a different sequence each time the program is run, you must use a different value for x each time.

If x is positive or not included, RND(x) generates the next random number in the sequence.

RND(0) repeats the last number generated.

To get random numbers in the range 0 through n , use the formula:

$\text{INT}(\text{RND} * (n+1))$

RND Function

Examples

In this example, the first horizontal line of results shows three random numbers, generated using a positive x .

In line 40, a negative number is used to reseed the random number generator. The random numbers produced after this reseeding are in the second row of results.

In line 80, the random-number generator is reseeded using the `RANDOMIZE` statement. In line 90, it is reseeded again by calling `RND` with the same negative value as in line 40. This cancels the effect of the `RANDOMIZE` statement; the third line of results is identical to the second line.

In line 130, `RND` is called with an argument of 0, so the last number printed is the same as the preceding number.

```
10 FOR I=1 TO 3
20 PRINT RND(I);      ' x>0
30 NEXT I
40 PRINT: X=RND(-6)  ' x<0
50 FOR I=1 TO 3
60 PRINT RND(I);      ' x>0
70 NEXT I
80 RANDOMIZE 853      ' randomize
90 PRINT: X=RND(-6)  ' x<0
100 FOR I=1 TO 3
110 PRINT RND;        ' same as x>0
120 NEXT I
130 PRINT: PRINT RND(0)
```


RND

Function

Results:

```
.7107346 .99058 .8523988
.4124333 .4854596 .9428332
.4124333 .4854596 .9428332
.9438332
```

Reseeding with the RND (negative number) function reseeds through permutation of the last floating-point temporary. Because no floating-point calculations are done in this example, the new seed is always the same:

```
10 DEFINT A-Z
20 FOR J=1 TO 5
30 X=RND(-J)
40 FOR I=1 TO 5:PRINT RND;NEXT:PRINT
50 NEXT J
```

If line 20 is changed to:

```
20 FOR J=1 TO 2 STEP .1
```

a new seed is generated each time.

Purpose

The RTRIM\$ function removes trailing spaces from string expressions.

Format

$v\$ = \text{RTRIM}\$(x\$)$

Comments

$x\$$ is the name of the string you want to trim.

The RTRIM\$ function examines $x\$$, removes any spaces that pad the end of the string, and returns a new string, $v\$$, without the spaces. $x\$$ remains unchanged.

See also "LTRIM\$ Function.

RTRIM\$

Function

Examples

This example demonstrates LTRIM\$ AND RTRIM\$.

```
DIM FixedString AS STRING * 10 ' FixedString = 10 character string
DIM NormalString$ ' NormalString= a dynamic string
```

```
FixedString = "Test"
NormalString$ = "Test"
```

```
' RTRIM$ must be used when comparing a fixed string with a normal
' one to trim off any default trailing blanks:
```

```
IF RTRIM$(FixedString) = NormalString$ THEN
    PRINT "The two strings are equal"
```

```
' If this happens, something's wrong:
```

```
ELSE
    PRINT "The two strings are not equal"
END IF
```

```
' Try a string with leading blanks:
```

```
NormalString$ = " Test"
IF RTRIM$(FixedString) = NormalString$ THEN
    PRINT "The two strings are still equal"
```

```
' LTRIM Removes the leading blanks so the comparison will work:
```

```
ELSEIF RTRIM$(FixedString) = LTRIM$(NormalString$) THEN
    PRINT "The two strings are equal if leading blanks are removed"
```

```
' If this happens, something's wrong:
```

```
ELSE
    PRINT "The two strings aren't equal"
END IF
END
```

Purpose

The RUN command begins a program.

Format

RUN [*line*]

RUN *filespec*

Comments

line is the line number of the program where you want to begin.

filespec is a string expression for the file specification. It can contain a path. *Filespec* must conform to the rules outlined under “File Names” and “File Specification” in *IBM BASIC Compiler/2 Fundamentals*; otherwise, an error occurs.

RUN or RUN *line* begins the program. If *line* is specified, running begins with the specified line number. Otherwise, running begins at the lowest line number.

RUN *filespec* loads a file from disk into memory and runs it. It closes all open files and deletes the current contents of memory before loading the designated program. See also “BASIC Disk Input and Output” in *IBM BASIC Compiler/2 Fundamentals*.

Running a RUN command turns off any sound that is running and resets to Music Foreground. Also, PEN and STRIG are reset to OFF.

Note: All variables and strings are reset to 0 or nulls when the RUN statement is run.

RUN Command

Examples

This is an example of RUN *line*:

```
10 A=1
20 B=2
30 C=3
40 D=4
50 PRINT A,B,C,D
60 IF D=0 THEN END
70 RUN 50
80 END
```

Results:

1	2	3	4
0	0	0	0

As an example of RUN *filespec*, change line 70 to read:

```
70 RUN "NEXTPGM.EXE"
```

The program would look like:

```
10 A=1
20 B=2
30 C=3
40 D=4
50 PRINT A,B,C,D
60 IF D=0 THEN END
70 RUN "NEXTPGM.EXE"
80 END
```

NEXTPGM.BAS looks like:

```
10 PRINT A,B,C,D
20 END
```

Results:

1	2	3	4
0	0	0	0

RUN Command

Note: You must first compile and link NEXTPGM.BAS to create NEXTPGM.EXE. You cannot RUN a BASIC interpreter application with this statement. For instance, the following program line:

```
70 RUN "NEXTPGM.BAS"
```

is legal in the BASIC Interpreter environment. However, unpredictable errors result in the BASIC Compiler environment because .BAS is not recognized as a executable filetype. The file extensions recognized as executable by the compiler are .BAT, .CMD, .COM and .EXE.

SADD

Function

Purpose

The SADD function returns the address of the specified string expression.

Format

v = SADD (*string expression*)

Comments

Use this function with care, because strings in string space can move at any time. Three possible causes of string-space rearrangement are:

- A string-space compaction as a result of FRE
- Opening or closing a file
- Allocating a string variable.

See also the FRE, PEEK, POKE, VARPTR, and VARPTR\$ functions.

Examples

```
A$ = "Late arrivals"           'Store string in variable A$
ADD = SADD(A$)                 'Get address of A$
LN = LEN(A$) - 1               'Get length of A$
GOSUB PRINTOUT                 'Print A$
POKE ADD,ASC("F")              'Change first, third, and
                                'fourth letters of A$

POKE ADD+2,ASC("s")
POKE ADD+3,ASC("t")
GOSUB PRINTOUT                 'Print new value of A$
END

PRINTOUT:
  FOR I = 0 TO LN
    ASCII = PEEK(ADD + I) 'Get value at address
    PRINT CHR$(ASCII);    'Convert value to character, and print
  NEXT
  PRINT                   'Print new line
  RETURN
```

SADD Function

Results:

```
Late arrivals  
Fast arrivals
```

This example uses SADD to access the text of a string rather than the variables string descriptor (which is what would happen using VARPTR).

```
' Set a string variable to some text:  
A$ = "That was the week that was"  
  
' Get address of the string text:  
StringText = SADD (A$)  
  
' Use PEEK to get the ASCII of each char and print it out  
' as a character:  
FOR I=1 TO LEN(A$)  
    ASCIIVal = PEEK(StringText+I-1)  
    PRINT CHR$(ASCIIVal);  
NEXT I  
END
```

SCREEN

Function

Purpose

The SCREEN function returns the ASCII code (0–255) for the character on the active screen at the specified row (line) and column.

Format

$v = \text{SCREEN}(\text{row}, \text{col}[, z])$

Comments

row is a numeric expression in the range 1 through 25. If the soft key display is turned on, however, only 24 rows are available.

col is a numeric expression in the range 1 through 40 or 1 through 80, depending on the WIDTH setting.

z is a numeric expression that evaluates to a true or false value. *z* is valid only in text mode.

See Appendix B, “ASCII Character Codes,” for a list of ASCII codes.

In text mode, if *z* is included and is true (non-0), the color attribute for the character is returned instead of the code for the character. The color attribute is a number in the range 0 through 255. This number, *v*, is deciphered as follows:

$(v \text{ MOD } 16)$ is the foreground attribute.

$((v \setminus 16) \text{ MOD } 8)$ is the background attribute, where *foreground* is calculated as above.

If $(v > 127)$, character is blinking; otherwise it is not.

For a list of colors and their associated attributes, see “COLOR” statement.

SCREEN Function

In graphics mode, if the specified location contains graphic information (points or lines, not just a character), the SCREEN function returns 0.

Any values entered outside the ranges indicated result in an **Illegal function call** error.

The SCREEN statement is explained in the next entry.

Examples

In this example, if the character at 10,10 is A, X is 65:

```
100 X = SCREEN (10,10)
```

This example returns the color attribute of the character in the upper left corner of the screen:

```
110 X = SCREEN (1,1,1)
```

SCREEN

Statement

Purpose

The SCREEN statement sets the screen attributes to be used by subsequent statements.

This statement is not meaningful with the Monochrome Display and printer adapter.

Format

```
SCREEN [mode] [, [burst] [, [apage] [, [vpage]]]
```

Comments

mode is a numeric expression resulting in an integer value of 0, 1, or 2. Valid modes are:

- 0** Text mode at current width (40 or 80).
- 1** Medium-resolution graphics mode (320x200).
- 2** High-resolution graphics mode (640x200).

burst is a numeric expression resulting in a true or false value. It enables or disables color. On an RGB monitor, color burst is always on. On a composite monitor, color burst can be on or off.

In text mode (*mode*=0), a false (0) value disables color (only the monochrome images are displayed); a true (non-0) value enables color (color images are displayed). In medium resolution graphics mode (*mode*=1), a true (non-0) value disables color, and a false (0) value enables color.

Because high resolution graphics are only two colors (black and white), this parameter has no effect in high resolution.

SCREEN Statement

apage (active page) is an integer expression in the range 0 through 7 for width 40; 0 through 3 for width 80. It selects the page to be written to by output statements to the screen, and is valid only in text mode (*mode*=0). *Apage* is always 0 under OS/2 mode.

vpage (visual page) selects the page to be displayed on the screen, in the same way as *apage* above. The visual page can be different from the active page. *vpage* is valid only in text mode (*mode*=0). If omitted, *vpage* defaults to *apage*. *Vpage* is always 0 under OS/2 mode.

Mode	Description	Width	Psize	Colors
0	Alpha	40,80	2k,4k	16
1	320x400	40	16	4
2	640x200	80	16k	2

If all parameters are valid, the new screen mode is stored; or the screen is erased; or the foreground color is set to white; or and the background and border colors are set to black.

If the new screen mode is the same as the previous mode, and color burst does not change, nothing is changed.

If only *apage* and *vpage* are specified, display pages are changed for viewing. Initially, both active and visual pages default to 0 (zero). By manipulating active and visual pages, you can display one page while building another. You can then switch visual pages instantaneously.

If you mix text and graphics in the 40-or 80-column graphics mode and are not using a U.S. keyboard, refer to the *GRAFTABL* command in *IBM Disk Operating System Version 3.30 Reference* and the *IBM Operating System/2 User's Reference* for more information regarding additional character support with the Color/Graphics monitor.

SCREEN

Statement

Note: Only one cursor is shared among all the pages. If you are going to switch active pages back and forth, save the cursor position on the current active page, using `POS(0)` and `CSRLIN`, before changing to another active page. When you return to the original page, you can restore the cursor position using the `LOCATE` statement.

Any parameter can be omitted. Omitted parameters, except *vpage*, assume the old value. Any values entered outside the ranges indicated result in an **illegal function call** error. Previous values are retained.

Examples

This example selects text mode with color burst enabled, and sets active and visual page to 0:

```
10 SCREEN 0,1,0,0
```

In this example, mode and color burst remain unchanged. Active page is set to 1 and display page to 2.

```
10 SCREEN ,,1,2
```

This example switches to high resolution graphics mode:

```
10 SCREEN 2,,0,0
```

This example switches to medium-resolution color graphics, color burst enabled:

```
10 SCREEN 1,0
```

Purpose

The SETMEM function is used to alter the amount of system memory allocated for all BASIC data.

Format

$v = \text{SETMEM}(x)$

Comments

The x is a signed integer number of bytes representing the change to the current memory allocation.

The value returned by the SETMEM function represents the current memory allocation as adjusted by SETMEM, both the near and far heaps.

If you specify SETMEM(0), the value returned is the current allocation.

If x is negative, the memory is released by SETMEM and becomes available for memory allocation through operating system memory allocation calls from other languages. BASIC data includes static data, common blocks, file buffers, strings, and all arrays. COM buffers are not included.

If SETMEM can not adjust the BASIC data allocation by the amount specified by x , then SETMEM simply returns the current allocation. No adjustment is made.

Note to Operating System/2 users

SETMEM is available in the OS/2 mode, but performs no function. SETMEM returns the cumulative effect on a starting "memory size" of 640K for compatibility with BASIC programs written for DOS mode.

SETMEM

Function

To the BASIC application, it looks like 640K was available, and all further adjustments are also successful.

Examples

```
' This is an example of SETMEM expanding and contracting  
' the "far heap" data space.
```

```
' Check current memory size:  
NewSize& = SETMEM(0&)  
PRINT "Original memory= "NewSize&
```

```
' Add 2k to BASIC memory allocation:  
NewSize& = SETMEM(2048&)  
PRINT "New memory size= "NewSize&
```

```
' Perhaps call a C routine here ...
```

```
' Reduce memory allocation by 2k:  
NewSize& = SETMEM(-2048&)  
PRINT "New memory size= "NewSize&
```

```
END
```

Purpose

The SGN function returns the sign of x .

Format

$v = \text{SGN}(x)$

Comments

The x is any numeric expression.

$\text{SGN}(x)$ is the mathematical signum function:

- If x is positive, $\text{SGN}(x)$ returns 1.
- If x is 0, $\text{SGN}(x)$ returns 0.
- If x is negative, $\text{SGN}(x)$ returns -1 .

Examples

This example branches to 100 if x is negative; 200 if x is 0; and 300 if x is positive:

```
ON SGN(X)+2 GOTO 100,200,300
```

SHARED Statement

Purpose

The SHARED statement allows a subprogram to access variables declared in the main program without passing them as parameters.

Format

SHARED *variable*[(*)*] [AS *type*] [,*variable*[(*)*] [AS *type*]]...

Comments

variable is the name of any simple variable or array declared at the main program level. If the variable is an array, its name must be followed by a pair of parentheses (for example, "ARRAY%(*)*").

type is one of the following:

- INTEGER
- LONG
- SINGLE
- DOUBLE
- STRING [**bytecount*]
- *typename*

typename must have been defined in a previous TYPE statement.

Usually, variables and arrays referred to inside a subprogram are considered local to that subprogram and initial values of 0 are assumed.

However, by using either the SHARED statement in a subprogram, or the SHARED attribute to the COMMON, DIM, or REDIM statements at the main program level of the module, you can access variables without passing them into a subprogram as parameters. The SHARED attribute is used for sharing variables among all subprograms in a module,

SHARED Statement

while the SHARED statement affects variables within a single subprogram.

For simple variables, if a variable named in a SHARED statement has not yet been referenced at the main-program level, the variable is created at main-program level and is shared with the subroutine or function.

For arrays, if an array named in a SHARED statement has not yet been referenced at the main-program level, an error occurs. The compiler cannot know how many dimensions to give the array and what bounds they should have.

The SHARED statement *must* appear only inside a named subprogram (see “SUB Statement” and “FUNCTION Statement”). If it occurs outside a subprogram, an error occurs.

Examples

The following program calculates the area of a chosen shape. By making use of shared variables, the area for a different shape from the one chosen can be calculated without re-entering the dimensions.

SHARED Statement

```
200 REM AREA CALCULATION PROGRAM
210 CLS
220 PRINT "AREA CALCULATION"
230 PRINT: PRINT "1. RECTANGLE"
240 PRINT "2. SQUARE"
250 PRINT "3. TRIANGLE"
260 PRINT "4. EXIT"
270 PRINT "MAKE SELECTION: ";
280 FIG$=INKEY$: IF FIG$ = "" THEN 280
290 IF ASC(FIG$) < 49 OR ASC(FIG$) > 52 THEN 280
300 IF FIG$ = "4" THEN 400
310 CLS
320 INPUT "ENTER BASE: " ; BSE
330 IF FIG$ = "2" THEN 350
340 INPUT "ENTER HEIGHT: " ; HGHT
350 CALL CALCAREA
360 PRINT "AREA = "; AREA
370 LOCATE 23,1: PRINT "PRESS ANY KEY TO CONTINUE"
380 $$ = INKEY$: IF $$ = "" THEN 380
390 GOTO 210
400 END
410 REM **** AREA SUBPROGRAM ****
420 SUB CALCAREA STATIC
430 SHARED AREA, BSE, HGHT, FIG$
440 ON VAL(FIG$) GOTO 450, 470, 490
450 AREA = BSE * HGHT
460 GOTO 500
470 AREA = BSE ^ 2
480 GOTO 500
490 AREA = (BSE * HGHT) / 2
500 END SUB
```

Purpose

The SHELL function performs OS/2-mode commands and runs other programs.

This function is not available in DOS mode.

Format

v = SHELL(*command string*)

Comments

command string is a string expression containing the name of a program or an OS/2 mode command to run, and, optionally, any parameters you are passing to the program or command.

Any program run under a BASIC program is referred to as a child process. SHELL runs child processes by loading and running a copy of the command processor with the **/C** switch. By using CMD.EXE in this way, SHELL correctly passes any parameters you can have to the child process. You can redirect standard input and output and run built-in commands such as DIR and PATH. You can also invoke batch files from the SHELL function.

When you run SHELL as a function, the parent BASIC program and the child process run at the same time.

The value returned by the SHELL function is the id assigned to the child process by the operating system.

When you run child processes from a BASIC application using the SHELL function, there are some procedures and rules that your application should follow. Going outside the boundaries of these guidelines could cause your application to fail or produce unpredictable results.

SHELL

Function

A child process that alters any file opened by the BASIC application can have unpredictable results. If you must update such files, be sure to close them from your BASIC application before SHELLing to your child process. Remember, files that were opened under the redirection of standard input and output constitute OPEN files. These files should not be modified using the SHELL process.

A program name in *command string* can have any extension you choose. If you do not supply an extension, OS/2 looks for an .EXE extension, and then a .CMD extension. If the *filename* is not located during this search, the CMD.EXE issues an error message.

Any text separated from a program name supplied in *command string* by at least one blank is processed as a program parameter.

BASIC applications inherit their environments from the operating system. Any changes your application makes to the environment are reflected in the environment for the child process.

Note: For more information on the environment, see *IBM Operating System/2 User's Reference* or *IBM Disk Operating System Version 3.30 Reference*.

Make sure that your system has enough available memory to load any programs that you want to run as child processes. An attempt to run a child process with insufficient free memory causes an **Out of Memory** error.

SHELL Function

Examples

The following example uses the SHELL function to make a backup copy of the file SAMPLE.BAS. The program continues to run while this copy is being made.

```
DUMMYVAR = SHELL("COPY SAMPLES.BAS SAMPLES.BAK")
```

In the next example, the program called MYPROG is run as a child process.

```
PROGNAME$ = "MYPROG"  
ARGV$ = " FILE1 FILE2 /XYZ"  
X = SHELL(PROGNAME$ + ARGV$)
```

SHELL

Statement

Purpose

The SHELL statement performs operating system commands and runs other programs.

Format

SHELL [*command string*]

Comments

command string is a string expression containing the name of a program to run, and, optionally, any parameters you are passing to the child process.

Any program run under a BASIC program is referred to as a child process. When you run SHELL as a statement, the parent BASIC program waits until the child process finishes. When the child process has finished running, control returns to the parent BASIC program at the statement following the SHELL statement.

SHELL runs child processes by loading and running a copy of the command processor with the **IC** switch. By using COMMAND.COM or CMD.EXE in this way, SHELL correctly passes any parameters you can have to the child process. You can redirect standard input and output and run built-in commands such as DIR and PATH.

The BASIC program waits in memory while the child process is running. When the child process finishes, the BASIC program continues.

If you enter SHELL with no *command string*, a copy of the command processor is loaded, the operating system prompt appears, and you can enter any commands that are valid operating system commands (DIR, COPY, and so on). You can return to the BASIC program by typing the word EXIT.

SHELL Statement

You can also invoke batch files from the SHELL statement. To return to the parent BASIC program, your batch file must run an EXIT statement.

When you run child processes from a BASIC application using the SHELL statement, there are some procedures and rules that your application should follow. Going outside the boundaries of these guidelines could cause your application to fail or produce unpredictable results.

To guarantee that you return to your BASIC program from your child process in the screen mode that you expect, you can do one of two things:

- If you are using DOS mode, use BIOS Interrupt 10H, function call 15, to save the current video mode. When your child process returns to DOS mode, use function call 0 to restore the video mode.

If you are using OS/2 mode, use the VIOGETMODE and VIOSETMODE OS/2 calls to get and set the video mode.

- From your BASIC program, run a SCREEN statement followed by a CLS statement immediately after the SHELL statement.

Under DOS mode, before a BASIC program runs a SHELL statement, it saves any interrupt vectors it uses. However, interrupt vectors your routines use (but are not used by the BASIC program) are not restored. So be sure to save any interrupt vectors your routine uses to preserve the proper interface to the operating system.

In DOS mode, any routine that you run from the SHELL statement should never end and stay resident. If this procedure is attempted, the following can occur: all files close, the error message prints, and control returns to the operating system.

A child process that alters any file opened by the BASIC application can have unpredictable results. If you must update such files, be sure to close them from your BASIC application before running a SHELL to your child process. Remember, files that were opened under the redirection of standard input and output constitute OPEN files. These files should not be modified using the SHELL process.

SHELL

Statement

A program name in *command string* can have any extension you choose. If you do not supply an extension, the operating system looks for a .COM extension, then a .EXE extension, and finally a .BAT or .CMD extension. If the *filename* is not located during this search, the command processor issues an error message.

Any text separated from a program name supplied in *command string* by at least one blank is processed as a program parameter.

BASIC applications inherit their environments from the operating system. Any changes your application makes to the environment are reflected in the environment for the child process.

Note: For more information on the environment, see *IBM Operating System/2 User's Reference* or *IBM Disk Operating System Version 3.30 Reference*.

Make sure that your system has enough available memory to load any programs that you want to run as child processes. An attempt to run a child process with insufficient free memory causes an **Out of Memory** error.

SHELL Statement

Examples

The following example displays a disk directory from your compiled program:

```
100 SHELL
A>DIR (type DIR command at DOS prompt)
A>EXIT (type EXIT to return to program)
```

The same result can be achieved with:

```
100 SHELL "DIR"
```

The next example creates a file, exits to the operating system SORT utility, and returns to the BASIC application:

```
10 OPEN "SORTIN.DAT" FOR OUTPUT AS #1
20 'writes data to be sorted
.
.
.
100 CLOSE 1
110 SHELL "SORT <SORTIN.DAT >SORTOUT.DAT"
120 OPEN "SORTOUT.DAT" FOR INPUT AS #1
130 'processes the sorted data
```

SIGNAL

Function

Purpose

The SIGNAL function enables and disables trapping of OS/2 mode signals.

This function is not available in DOS mode.

Format

SIGNAL(*n*) ON
SIGNAL(*n*) OFF
SIGNAL(*n*) STOP

Comments

n is the number of a signal sent through OS/2. For a list of the valid signal numbers, see the *IBM Operating System/2 Programmer's Guide*.

If you specify a signal number other than those which the OS/2 mode supports, BASIC returns an **Illegal function call** error message.

You must run a SIGNAL(*n*) ON statement to enable trapping by the ON SIGNAL(*n*) statement. (See "ON SIGNAL Statement.") After a SIGNAL(*n*) ON statement, every time the program starts a new statement, the compiler checks to see if signal *n* has been received.

If you run a SIGNAL(*n*) OFF statement, BASIC does not trap the OS/2 mode *n* signal. Even if a signal *n* occurs, the program does not remember the event.

If you run a SIGNAL(*n*) STOP statement, BASIC does not trap signal *n*, but BASIC remembers the event and traps signal *n* as soon as you run a SIGNAL(*n*) ON statement.

SIGNAL Function

Examples

This is an example of using `SIGNAL(n)` to allow processing to continue in a program while it is waiting for an inter-process signal:

```
ON SIGNAL(4) GOSUB ProcessSignal
SIGNAL(4) ON

DO

    ' program could go here ...

LOOP UNTIL INKEY$ <> ""

END

ProcessSignal:

PRINT "Signal received ..."
RETURN
```

SIN Function

Purpose

The SIN function calculates the trigonometric sine function.

Format

$v = \text{SIN}(x)$

Comments

The x is an angle in radians.

To convert degrees to radians, multiply by $\text{PI}/180$, where $\text{PI}=3.141593$.

Examples

This example calculates the sine of 90 degrees after first converting the degrees to radians:

```
10 PI=3.141593
20 DEGREES = 90
30 RADIANS=DEGREES * PI/180 ' PI/2
40 PRINT SIN(RADIANS)
```

Results:

1

SOUND Statement

Purpose

The SOUND statement generates sound through the speaker.

This statement is not available under the OS/2 mode.

Format

SOUND *freq*, *duration*

Comments

freq is the desired frequency in Hertz (cycles per second). It must be a numeric expression in the range 37 through 32767.

duration is the desired duration in clock ticks. The clock ticks occur 18.2 times per second. The *duration* must be a numeric expression. The range of values for *duration* is 0.0015 through 65535.

When the SOUND statement produces a sound, the program continues to run until another SOUND statement is reached. If *duration* of the new SOUND statement is 0, the currently running SOUND statement is turned off. Otherwise, the program waits until the first sound ends before it runs the new SOUND statement.

You can cause the sounds to be buffered so that running does not stop when a new SOUND statement is encountered.

If a SOUND statement is followed immediately by an END statement, the sound being produced stops upon running the END statement and the SOUND buffer is flushed.

See the explanation of the Music Background (**MB**) command under "PLAY Statement."

SOUND

Statement

If no SOUND statement is running, SOUND x, 0 has no effect.

The tuning note, A, has a frequency of 440. The following figure correlates notes with their frequencies.

Note	Frequency	Note	Frequency
C	130.810	C	523.250
D	146.830	D	587.330
E	164.810	E	659.260
F	174.610	F	698.460
G	196.000	G	783.990
A	220.000	A	880.000
B	246.940	B	987.770
C*	261.630	C	1046.500
D	293.660	D	1174.700
E	329.630	E	1318.500
F	349.230	F	1396.900
G	392.000	G	1568.000
A	440.000	A	1760.000
B	493.880	B	1975.500

*Middle C.

Higher (or lower) notes can be approximated by doubling (or halving) the frequency of the corresponding note in the previous (next) octave.

To create periods of silence, use SOUND 32767,*duration*.

SOUND Statement

The duration for one beat can be calculated from beats per minute by dividing the beats per minute into 1092 (the number of clock ticks in a minute).

This figure shows typical tempos in terms of clock ticks:

Tempo	Beat/ Minute	Ticks/ Beat
very slow	Larghissimo	27.3-18.2
	Largo	40-60
	Larghetto	60-66
	Grave	
	Lento	16.55-14.37
	Adagio	66-76
slow	Adagietto	14.37-10.11
	Andante	76-108
medium	Andantino	10.11-9.1
	Moderato	108-120
fast	Allegretto	9.1-6.5
	Allegro	120-168
	Vivace	
	Veloce	6.5-5.25
	Presto	168-208
very fast	Prestissimo	

SOUND

Statement

Examples

The following program creates a glissando (sliding up and down the scale):

```
FOR I=440 TO 1000 STEP 5  
SOUND I, 0.5  
NEXT  
FOR I=1000 TO 440 STEP -5  
SOUND I, 0.5  
NEXT
```

Purpose

The SPACE\$ function returns a string consisting of n spaces.

Format

$v\$ = \text{SPACE}\(n)

Comments

n must be in the range 0 through 32767.

See also "SPC Function."

Examples

This example uses the SPACE\$ function to print each number N on a line preceded by N spaces. An additional space is inserted because BASIC puts a space in front of positive numbers.

```
10 FOR N = 1 TO 5
20 X$ = SPACE$(N)
30 PRINT X$;N
40 NEXT N
```

Results:

```
 1
 2
 3
 4
 5
```

SPC Function

Purpose

The SPC function skips n spaces in a PRINT statement.

Format

```
PRINT SPC( $n$ )
```

Comments

The n must be in the range 0 through 255.

If n is greater than the defined width of the device, the value used is $n \text{ MOD } width$. SPC can be used only with PRINT, LPRINT, and PRINT # statements.

See also "SPACE\$ Function."

Examples

This example prints OVER and THERE separated by 15 spaces:

```
PRINT "OVER" SPC(15) "THERE"
```

Results:

```
OVER           THERE
```

Purpose

The SQR function returns the square root of x.

Format

$v = \text{SQR}(x)$

Comments

The x must be greater than or equal to 0.

Examples

This example calculates the square roots of the numbers 10, 15, 20, and 25:

```
10 FOR X = 10 TO 25 STEP 5
20 PRINT X, SQR(X)
30 NEXT
```

Results:

10	3.162278
15	3.872984
20	4.472136
25	5

STATIC Statement

Purpose

The `STATIC` statement designates simple variables or arrays as local to the subprogram in which they are declared and preserves their values when the subprogram is exited and reentered.

Format

`STATIC variable[(n)][AS type] [,variable[(n)][AS type]]...`

Comments

variable is a simple variable or array. An array declaration consists of a variable symbol followed by an integer constant in parentheses.

n is an integer constant. It represents the number of dimensions in the array, not the actual value of the dimensions.

type is one of the following:

- INTEGER
- LONG
- SINGLE
- DOUBLE
- STRING [**bytecount*]
- *typename*

typename must have been defined in a previous `TYPE` statement.

The `STATIC` statement may appear only inside a named subprogram (see “`SUB` and `END SUB` and `EXIT SUB` Statement”). If `STATIC` appears outside a named subprogram, an error occurs.

Simple variables and arrays referred to or declared in subprograms are local to the subprograms in which they are declared. Initial values of 0 or null string are assumed; however, if the subprogram is

STATIC Statement

exited and reentered, the values retained for the variables may have been changed. Declaring the variables in a `STATIC` statement guarantees that the previous values are retained.

If the first reference to an array is made with the `STATIC` statement, that array is allocated dynamically. Such arrays are not actually allocated until they are dimensioned with the `DIM` statement or are re-dimensioned with the `REDIM` statement.

Simple variables or arrays declared within a `STATIC` statement override any shared variables or arrays with the same name.

By default, variables used in functions are global. The `STATIC` statement can also be used inside multiline function definitions to declare a variable as local to that function only.

For related information, see "SHARED Statement."

Examples

The following program computes the factorial value of a number:

```
200 REM FACTORIAL PROGRAM
210 DIM SHARED NFACT, N
220 INPUT "ENTER NUMERAL"; M
230 N=M
240 WHILE N > 1
250   CALL FACTORIAL
260   N = N - 1
270 WEND
280 PRINT M;" FACTORIAL = ";NFACT
290 END
300 REM **** FACTORIAL SUBPROGRAM ****
310 SUB FACTORIAL STATIC
320 STATIC N1
330 REM N1 WILL RETAIN ITS VALUE
340 REM AFTER SUBPROGRAM IS EXITED
350 IF N1 <> 0 THEN N1 = N1 * N ELSE N1 = N
360 IF N = 2 THEN NFACT = N1
370 END SUB
```

STICK

Function

Purpose

The STICK function returns the x and y coordinates of two joysticks.

This function is not available under OS/2 mode.

Format

$v = \text{STICK}(n)$

Comments

n is a numeric expression in the range 0 through 3 that affects the result as follows:

- 0** returns the x-coordinate for joystick A.
- 1** returns the y-coordinate of joystick A.
- 2** returns the x-coordinate of joystick B.
- 3** returns the y-coordinate of joystick B.

Note: STICK(0) reads all four values for the coordinates. To use any of the other stick functions (STICK(1), STICK(2) or STICK(3)), you must first call STICK(0).

The range of values for x and y depends on your particular joysticks.

STICK Function

Examples

This program prints 100 samples of the coordinates of joystick B:

```
10 PRINT "Joystick B"  
20 PRINT "x coordinate","y coordinate"  
30 FOR J=1 TO 100  
40 TEMP=STICK(0)  
50 X=STICK(2): Y=STICK(3)  
60 PRINT X,Y  
70 NEXT
```

STOP Statement

Purpose

The STOP statement stops a program and returns to command level.

Format

STOP

Comments

STOP statements can be used anywhere in a program to stop running. When BASIC encounters a STOP statement, it displays the following message:

```
STOP in line xxx of module modulename at address ----:---
```

Hit any key to return to system

When the compiler encounters a STOP statement, it displays a message with a hexadecimal address indicating where the program stopped. If you compile using the **/D**, **/X**, or **/E** switch, the compiler displays the line number of the stopping point also.

STOP closes all open files.

Examples

This example calculates the value of TEMP, then stops. The program was named STOPEX.BAS and it was compiled with the **/D** option.

```
10 INPUT A, B
20 TEMP= A*B
30 STOP
40 FINAL = TEMP+200: PRINT FINAL
```

STOP Statement

Results:

) ? 26, 2.1

STOP in line 30 of module STOPEX at address 1E82:0092

Hit any key to return to system

STR\$ Function

Purpose

The STR\$ function returns a string representation of the value of x.

Format

v\$ = STR\$(x)

Comments

The x is any numeric expression.

If x is positive, the string returned by STR\$ contains a leading blank (the space reserved for the plus sign). For example:

```
PRINT STR$(321); LEN(STR$(321))
```

Results:

```
321 4
```

The VAL function is complementary to STR\$. See "VAL Function."

Examples

This example branches to different sections of the program according to the number of digits in a number that is entered. The number of digits is counted by using STR\$ to convert the number to a string; then the program branches, based on the length of the string.

```
10 INPUT "TYPE A NUMBER";N
20 ON LEN(STR$(N))-1 GOSUB 30,100,200,300
  .
  .
  .
```

STRIG Statement and Function

) Purpose

The STRIG statement and function returns the status of the joystick buttons (triggers).

The STRIG statement and function is not available OS/2 mode.

Format

As a statement:

STRIG ON

STRIG OFF

As a function:

$v = \text{STRIG}(n)$

Comments

n is a numeric expression in the range 0 through 7. It affects the value returned by the function as follows:

- 0** Returns -1 if button A1 was pressed since the last STRIG(0) function call; returns 0 if it was not pressed.
- 1** Returns -1 if button A1 is currently pressed; returns 0 if it is not pressed.
- 2** Returns -1 if button B1 was pressed since the last STRIG(2) function call; returns 0 if it was not pressed.
- 3** Returns -1 if button B1 is currently pressed; returns 0 if it is not pressed.

STRIG

Statement and Function

- 4 Returns -1 if button A2 was pressed since the last STRIG(4) function call; returns 0 if it was not pressed.
- 5 Returns -1 if button A2 is currently pressed; returns 0 if it is not pressed.
- 6 Returns -1 if button B2 was pressed since the last STRIG(6) function call; returns 0 if it was not pressed.
- 7 Returns -1 if button B2 is currently pressed; returns 0 if it is not pressed.

STRIG ON must run before any STRIG(*n*) function calls can be made. After STRIG ON, every time the program starts a new statement, the compiler checks to see if a button was pressed.

If STRIG is OFF, no testing takes place.

See "STRIG(*n*) Statement" for enhancements to the STRIG function.

Examples

```
STRIG ON
ATRIG = 0
'Wait for trigger A to be pressed:
WHILE NOT ATRIG : ATRIG = STRIG(0) : WEND
'As long as trigger A is down, beep:
WHILE ATRIG : ATRIG = STRIG(1) : BEEP : WEND
```

STRIG(*n*) Statement

Purpose

The STRIG(*N*) statement enables and disables trapping of the joystick buttons.

This statement is not available under the OS/2 mode.

Format

STRIG(*n*) ON
STRIG(*n*) OFF
STRIG(*n*) STOP

Comments

n can be 0, 2, 4, or 6 and indicates the button to be trapped, as follows:

0	button a
2	button b
4	button a
6	button b

STRIG(*n*) ON must run to enable trapping by the ON STRIG(*n*) statement. See the ON STRIG statement. After STRIG(*n*) ON, every time the program starts a new statement, the compiler checks to see if the specified button has been pressed.

If STRIG(*n*) OFF runs, no testing or trapping takes place. Even if the button is pressed, the event is not remembered.

If a STRIG(*n*) STOP statement runs, no trapping takes place. However, if the button is pressed it is remembered so that an immediate trap takes place when STRIG(*n*) ON runs.

Examples

STRIG(0) ON 'Enables trapping of button A1

STRING\$ Function

Purpose

The STRING\$ function returns a string length n whose characters all have ASCII code m or the first character of x .

Format

v \$ = STRING\$(n,m)

v \$ = STRING\$(n,x)

Comments

n, m are in the range 0 through 32767.

x is any string expression.

Examples

This example repeats an ASCII value of 45 to print a string of hyphens:

```
10 X$ = STRING$(10,45)
20 PRINT X$ "MONTHLY REPORT" X$
```

Results:

```
-----MONTHLY REPORT-----
```

This example repeats the first character of the string "ABCD":

```
10 X$="ABCD"
20 Y$=STRING$(10,X$)
30 PRINT Y$
```

Results:

```
AAAAAAAAAA
```

SUB and END SUB and EXIT SUB Statement

Purpose

The SUB AND END SUB AND EXIT SUB statement marks the beginning and end of an subprogram.

Format

```
SUB globalname [(parameter [AS type] [,parameter [AS type]]...)] STATIC
  statements
  [EXIT SUB]
  statements
END SUB
```

Comments

globalname is a name up to 40 characters long. This name cannot appear in any other SUB or FUNCTION statement.

parameter is the name of a simple variable or an array. If the parameter is an array, it must be specified in the form:

parameter (*integer*)

where *integer* is the number of dimensions the array has.

type is one of the following:

- INTEGER
- LONG
- SINGLE
- DOUBLE
- STRING
- *typename*

typename must have been defined in a previous TYPE statement.

SUB and END SUB and EXIT SUB Statement

STATIC is required to indicate that the subprogram is nonrecursive; that is, it does not call itself or a subprogram that in turn calls it.

statements are the **BASIC** statements to be performed when the subprogram is called.

A subprogram must begin with a **SUB** statement and end with an **END SUB** statement. The **EXIT SUB** statement is used to exit a subprogram before the **END SUB** statement is reached. Executing an **EXIT SUB** or **END SUB** statement transfers program control back to the calling routine.

To execute a subprogram, use the **CALL** statement. If you want to call a the subprogram before it is defined, you must use the **DECLARE** statement to describe the subprogram to **BASIC**.

A subprogram is similar to a function defined by a **FUNCTION** statement. However, unlike a function, a subprogram does not return a value associated with its name and, therefore, cannot appear as part of an expression.

Any simple variables or arrays referred to in the subprogram are considered to be local unless they have been explicitly declared to be **SHARED** variables.

When a subprogram is exited and reentered, the values of its local variables are reset to 0s and null strings. To guarantee that a local variable retains its assigned value upon reentry to the subprogram, it should be declared as **STATIC**. See “Scope of Variables” under “Modular Programming” in *IBM BASIC Compiler/2 Fundamentals* for more information.

See also “**CALL** Statement,” “**SHARED** Statement,” and “**STATIC** Statement” and “Modular Programming” in *IBM BASIC Compiler/2 Fundamentals*.

SUB and END SUB and EXIT SUB Statement

Examples

This program contains two subprograms. One subprogram converts all characters of a string to uppercase; the other subprogram converts all characters of a string to lowercase.

```
220 A$="abcde12345ABCDE!@#$$%"
230 CALL upper(A$,B$)
240 CALL lower(A$,C$)
250 IF B$="ABCDE12345ABCDE!@#$$%"_
    AND C$="abcde12345abcde!@#$$%" THEN 270
260 PRINT "FAILED"
270 END
280 SUB upper(N$,R$) STATIC
290 LCASELET$="abcdefghijklmnopqrstuvwxy"
300 UCASELET$="ABCDEFGHIJKLMNOPQRSTUVWXYZ"
310 L=LEN(N$)
320 IF L=0 THEN upper$="":EXIT SUB
330 FOR I = 1 TO L
340   P=INSTR(LCASELET$,MID$(N$,I,1))
350   IF P <> 0 THEN N$=MID$(N$,1,I-1)_
      +MID$(UCASELET$,P,1)+MID$(N$,I+1,L-I+1)
360 NEXT
370 R$=N$
380 END SUB
390 SUB lower(N$,R$) STATIC
400 LCASELET$="abcdefghijklmnopqrstuvwxy"
410 UCASELET$="ABCDEFGHIJKLMNOPQRSTUVWXYZ"
420 L=LEN(N$)
430 IF L=0 THEN lower$="":EXIT SUB
440 FOR I = 1 TO L
450   P=INSTR(UCASELET$,MID$(N$,I,1))
460   IF P <> 0 THEN N$=MID$(N$,1,I-1)_
      +MID$(LCASELET$,P,1)+MID$(N$,I+1,L-I+1)
470 NEXT
480 R$=N$
490 END SUB
```

SWAP Statement

Purpose

SWAP statement exchanges the values of two variables.

Format

SWAP *variable1*, *variable2*

Comments

variable1, *variable2*

are the names of two variables or array elements.

You can swap any type variable (integer, long integer single-precision, double-precision, string, or a type you defined with the TYPE statement), but the two variables must be of the same type or an error results. If you swap two strings, they do not have to be the same length. You can also swap records or record elements.

Examples

In this example, after line 30 is run, A\$ has the value " ALL " and B\$ has the value " ONE ":

```
10 A$=" ONE " : B$=" ALL " : C$="FOR"  
20 PRINT A$;C$;B$  
30 SWAP A$, B$  
40 PRINT A$;C$;B$
```

Results:

```
ONE FOR ALL  
ALL FOR ONE
```

SYSTEM Command

Purpose

The SYSTEM command exits your compiled program and returns to the operating system.

Format

SYSTEM

Comments

SYSTEM closes all files and returns to the operating system.

TAB Function

Purpose

The TAB function tabs to position n .

Format

PRINT TAB(n)

Comments

The n must be in the range 1 through 255.

If the current print position is already beyond space n , TAB goes to position n on the next line. Space 1 is the leftmost position and defined WIDTH is the rightmost position.

TAB can be used only in PRINT, LPRINT, and PRINT # statements.

If the TAB function is at the end of the list of data items, BASIC does not add a carriage return, as though the TAB function had an implied semicolon after it.

Examples

TAB is used in the following example to cause the information on the screen to line up in columns:

```
10 PRINT "ITEM" TAB(25) "AMOUNT" : PRINT
20 READ A$,B$
30 PRINT A$ TAB(25) B$
40 DATA "GROCERIES", "$25.00"
```

Results:

ITEM	AMOUNT
GROCERIES	\$25.00

Purpose

The TAN function returns the trigonometric tangent of x .

Format

$$v = \text{TAN}(x)$$

Comments

x is the angle in radians. To convert degrees to radians, multiply by $\text{PI}/180$, where $\text{PI}=3.141593$.

Examples

This example calculates the tangent of 45 degrees:

```
10 PI=3.141593
20 DEGREES=45
30 PRINT TAN(DEGREES*PI/180)
```

Results:

```
1
```

TIMES\$

Variable and Statement

Purpose

The `TIMES$` variable and statement sets or retrieves the current time.

Format

As a variable:

```
v$ = TIMES$
```

As a statement:

```
TIMES$ = x$
```

Comments

For the variable (`v$ = TIMES$`):

The current time is returned as an 8-character string. The string is in the form `hh:mm:ss`, where `hh` is the hour (00 to 23), `mm` is the minute (00 to 59); and `ss` is the second (00 to 59). (You may have set the time before you invoked your program).

For the statement (`TIMES$ = x$`):

The current time is set. `x$` is a string expression indicating the time to be set. `x$` can be given in one of the following forms:

`hh` Set the hour in the range 0 through 23. Minutes and seconds default to 00.

`hh:mm` Set the hour and the minute. Minutes must be in the range 0 through 59. Seconds default to 00.

`hh:mm:ss` Set the hour, the minute, and the second. Seconds must be in the range 0 through 59.

TIMES\$

Variable and Statement

A leading 0 can be omitted from any of the above values, but you must include at least one digit. For example, if you want to set the time as a half hour after midnight, you can enter:

```
TIMES$="0:30"
```

but not

```
TIMES$=":30"
```

If any of the values are out of range, an error is issued. The previous time is retained. If x\$ is not a valid string, a **Type mismatch** error results.

Examples

The following program continuously displays the time on the screen:

```
10 CLS
20 LOCATE 10,15
30 PRINT TIMES$
40 GOTO 20
```

TIMER

Function and Statement

Purpose

The **TIMER** function returns a single-precision number representing the number of seconds elapsed since midnight or a system reset.

The **TIMER** statement activates and deactivates the trap routine set up by the **ON TIMER(*n*)** statement.

Format

As a Function:

`V = TIMER`

As a Statement:

`TIMER ON`

`TIMER OFF`

`TIMER STOP`

Comments

As a Statement (TIMER...):

A **TIMER ON** statement must be used to start the **ON TIMER** statement. After **TIMER ON**, if a non-0 line number is specified in the **ON TIMER** statement, then every time the program starts a new statement or line number, (depending on whether you compiled using **/V** or **/W**), **BASIC** checks to see if the specified number of seconds have passed. When *n* seconds have elapsed, **BASIC** performs a **GOSUB** to the specified line. The event trap occurs, and **BASIC** starts counting again from 0.

Note: If your program contains any event- statements, such as **ON TIMER**, for example, you need to compile your program using the **/V** or **/W** switch.

TIMER Function and Statement

If `TIMER OFF` is used, no trapping takes place. Even if `TIMER` activity takes place, the event is not remembered.

If a `TIMER STOP` statement is used, no trapping takes place, but `TIMER` activity is remembered so that an immediate trap occurs when `TIMER ON` is used.

When the trap occurs, an automatic `TIMER STOP` is run so that recursive traps never take place. The `RETURN` from the trap routine automatically does a `TIMER ON` unless an explicit `TIMER OFF` was performed inside the trap routine.

You can use `RETURN line/label` to go back to the `BASIC` program at a fixed line number. Use this nonlocal return with care, because any other `GOSUBS`, `WHILES`, or `FORS` active at the time of the trap remain active.

As a Function (`v=TIMER`)

Fractional seconds are calculated to the nearest degree possible. `TIMER` is a read-only function.

Examples

This example illustrates how `TIMER` resets after midnight. Values may be slightly different for your system.

```
10 TIME$="23:59:59"  
20 FOR I=1 TO 20  
30 PRINT "TIME$= ";TIME$;" TIMER=";TIMER  
40 NEXT
```

TIMER

Function and Statement

Results:

```
TIME$= 23:59:59 TIMER= 86399.06  
TIME$= 23:59:59 TIMER= 86399.11  
TIME$= 23:59:59 TIMER= 86399.18
```

.

.

.

```
TIME$= 24:00:00 TIMER= 0  
TIME$= 00:00:00 TIMER= .05  
TIME$= 00:00:00 TIMER= .16  
TIME$= 00:00:00 TIMER= .21
```

ON TIMER is useful in programs that need an interval timer. This example displays the time of day on line 1 every minute:

```
10 CLS  
20 ON TIMER(60) GOSUB 10000  
30 TIMER ON  
  
.  
.  
.  
  
10000 OLDROW=CSRLIN 'save current row  
10010 OLDCOL=POS(0) 'save current column  
10020 LOCATE 1,1:PRINT TIME$;  
10030 LOCATE OLDROW,OLDCOL 'restore row & col  
10040 RETURN
```

TRON and TROFF Commands

Purpose

The TRON and TROFF commands trace the running of program statements.

Format

TRON

TROFF

Comments

As an aid in debugging, the TRON command enables a trace flag that prints each line number of the program as it is run. The numbers appear enclosed in square brackets.

If line numbers are only used occasionally in your program, the values printed by TRON are the last known line numbers. The trace is turned off by the TROFF command.

Note: When debugging your program with TRON and TROFF, you must compile using the **/D** switch. This switch causes the compiler to perform more extensive error checking, and detects errors such as **RETURN without GOSUB** that normally go undetected.

TRON and TROFF Commands

Examples

This example uses TRON and TROFF to trace running of a loop. The numbers in brackets are line numbers; the numbers not in brackets at the end of each line are the values of J, K, and L, which are printed by the program.

```
5 TRON
10 K=10
20 FOR I=1 TO 2
30 L=K + 10
40 PRINT J;K;L
50 K=K+10
60 NEXT
70 END
```

Results:

```
[10][20][30][40] 0 10 20
[50][60][30][40] 0 20 30
[50][60][70]
```

TYPE, ENDTYPE Statements

Purpose

The TYPE statement lets you define a variable type.

Format

```
TYPE typename  
    elementname AS type  
    [elementname AS type]  
.  
.  
.  
END TYPE
```

Comments

typename is a variable type that you want to define. It can be the same as any procedure name, label, variable name, or element name. The *typename* cannot have an explicit type character, such as %, &, !, #, or \$.

elementname is the name of a variable whose type you want to declare. It can be the same as any procedure name, label, variable name, or *typename*, but it cannot be an array. The *elementname* cannot have an explicit type character, such as %, &, !, #, or \$.

type is one of the following:

- INTEGER
- LONG
- SINGLE
- DOUBLE
- STRING * *bytecount*
- *typename*

typename must have been defined in a previous TYPE statement.

TYPE, ENDTYPE Statements

bytecount declares the length of a fixed-length string.

The TYPE definition can occupy several lines. Only AS clauses, remarks, and blank lines can be within the TYPE/END TYPE structure. Lines within the TYPE/END TYPE structure cannot have line numbers.

You can use the TYPE statement to declare the type of *elementname* before you use *elementname* in the program. A TYPE statement at the module level affects the entire module.

A TYPE statement may not be present in a subprogram or a function.

When the compiler checks to see if two types defined by TYPE statements are equivalent, it compares only the names of the types. One case where the compiler compares types is the LET statement. The type of variable on the right side must have the same name as the type of variable on the left side. Another case is when a subprogram or function is called. If a parameter is of a type defined in a TYPE statement, the name of that type must be the same as the name of the type of the corresponding parameter in the DECLARE, SUB, FUNCTION, or DEF FN statement, if there is one.

Type declarations cannot be recursive. That is, a type declaration cannot use itself as a *type*. For example, the following is incorrect:

```
TYPE MYTYPE
  XYZ AS MYTYPE
END TYPE
```

TYPE, ENDTYPE Statements

A type declaration can use a *type* that was defined by a previous TYPE statement.

Examples

```
TYPE ITEMREC
  NAME AS STRING * 10
  AMOUNT AS REAL
END TYPE
DIM ITEM AS ITEMREC, ITEMSAVE AS ITEMREC
READ ITEM.NAME, ITEM.AMOUNT
DATA "Groceries",25.00
ITEMSAVE = ITEM      ' Record assignment
PRINT "ITEM" TAB(25) "AMOUNT"
PRINT ITEMSAVE.NAME TAB(25) "$" ITEMSAVE.AMOUNT
```

Results:

ITEM	AMOUNT
Groceries	\$25

UBOUND

Statement

Purpose

The `UBOUND` statement returns the upper bound of the specified dimension of an array.

Format

`UBOUND(array[,dim])`

Comments

array is the name of the array being examined.

dim is an integer constant from 1 to the number of dimensions in the array. The default value is 1.

`UBOUND` returns the upper bound of the specified dimension of an array.

`LBOUND` and `UBOUND` are particularly useful for determining the size of an array passed to a subprogram.

UBOUND Statement

Examples

The following example uses LBOUND and UBOUND to determine the size of the array to be sorted:

```
200 OPTION BASE 1
210 DIM SHARED A(10)
220 CLS
230 PRINT "THE UNSORTED ARRAY"
240 FOR I = LBOUND(A) TO UBOUND(A)
250   READ A(I)
260   PRINT A(I)
270 NEXT I
280 CALL SORT
290 PRINT "THE SORTED ARRAY"
300 FOR I = LBOUND(A) TO UBOUND(A)
310   PRINT A(I)
320 NEXT I
330 DATA 40, 100, 19, 8, 66, 23
340 DATA 83, 6, 54, 120, 25, 98
350 END
360 REM **** EXCHANGE SORT SUBPROGRAM ****
370 SUB SORT STATIC
380 STATIC B
390 REM USE LBOUND TO DETERMINE LOWER
400 REM BOUNDARY OF ARRAY
410 FOR I = LBOUND(A) TO UBOUND(A) - 1
420   FOR J = I + 1 TO UBOUND(A)
430     IF A(I) <= A(J) THEN 470
440     B = A(J)
450     A(J) = A(I)
460     A(I) = B
470   NEXT J
480 NEXT I
490 END SUB
```

UBOUND

Statement

Results:

THE UNSORTED ARRAY

40
100
19
8
66
23
83
6
54
120

THE SORTED ARRAY

6
8
19
23
40
54
66
83
100
120

Purpose

The UCASE\$ function converts all the letters in a string to upper case.

Format

$v\$ = \text{UCASE\$}(m\$)$

Comments

$m\$$ is any string expression. The letters in this string are upper-case or lower-case.

The UCASE function returns a string containing the characters of an argument string converted to upper-case. You can use this function to increase the speed of programs that use comparisons that are not sensitive to case.

Examples

```
10 MIXED$ = "The UCASE$ Function."  
20 UPPER$ = UCASE$(MIXED$)  
30 PRINT "Mixed:",MIXED$  
40 PRINT "Upper case:",UPPER$
```

Results:

```
Mixed:      The UCASE$ Function.  
Upper case: THE UCASE$ FUNCTION.
```

UNLOCK

Statement

Purpose

The UNLOCK statement releases locks applied to an opened file.

This function is not available under OS/2.

Format

```
UNLOCK [#]n [, [recnum ] [TO recnum]]
```

Comments

n is the number of the opened file

recnum is the record number used to specify a range of records to be unlocked.

Under DOS, before you run an application that uses any LOCK or UNLOCK statements, you must first install the SHARE module. This module is on the DOS disk and is installed by entering the command "SHARE" at the DOS prompt or by installing network software.

If a record number or range of record numbers is specified and the file is opened in random mode, only those records in the range are unlocked. The record number range must exactly match the record number range given in the LOCK statement, or a **Permission denied** error occurs.

Failure to unlock all locks on a file before closing the file or exiting the program may cause undefined results.

The range of legal record numbers is 1 through 2147483647.

UNLOCK Statement

Examples

The following UNLOCK is legal:

```
LOCK #1,1 TO 4  
LOCK #1,5 TO 8  
UNLOCK #1,1 TO 4  
UNLOCK #1,5 TO 8
```

However, the following UNLOCK is illegal:

```
LOCK #1,1 TO 4  
LOCK #1,5 TO 8  
UNLOCK #1,1 TO 8
```

VAL Function

Purpose

The VAL function returns numeric value of string x\$.

Format

$v = \text{VAL}(x\$)$

Comments

x\$ is a string expression.

The VAL function strips blanks, tabs, and line feeds from the argument string to determine the result. For example:

```
VAL(" -3")
```

Results:

-3.

If the first characters of x\$ are not numeric, VAL(x\$) returns 0 (zero).

Note: When used with the VAL function, the letters D and E are used by BASIC to represent floating point constants. In this case, BASIC will convert any string containing a D or an E into exponential form. If the exponent of the number is greater than the maximum allowed by the precision, an **Overflow** error is returned.

See "STR\$ Function" for numeric-to-string conversion.

VAL Function

Examples

In this example, VAL is used to extract the house number from an address:

```
PRINT VAL("3408 SHERWOOD BLVD.")
```

Results:

```
3408
```

VARPTR

Function

Purpose

The VARPTR returns the offset into BASIC'S current data segment of a variable in memory.

Format

$v = \text{VARPTR}(\textit{variable})$

Comments

variable is the name of a numeric or string variable or array element in your program. *variable* can also be a record or a record element.

VARPTR returns an offset as an integer in the range 0 through 65535. This number is the offset into the compiler's data segment of the first byte of data identified with *variable*. The format of this data is described under "How Variables Are Stored" in *IBM BASIC Compiler/2 Fundamentals*.

In previous releases of the BASIC Compiler, VARPTR returned a 20-bit address when *variable* was a dynamic numeric array. With the IBM BASIC Compiler/2, dynamic numeric arrays require a 32-bit address (a segment and an offset). VARPTR now returns only the 16-bit offset to the variable. If you want to know the data segment, use the VARSEG function.

Note: If you created any programs with earlier versions of the BASIC Compiler that use VARPTR to return the address of a dynamic numeric array, you should change them to use VARPTR and VARSEG instead.

VARPTR Function

Warning: In the BASIC Compiler 2.00 you could use the VARPTR function to obtain the address of the FDB for an open file. This feature has been removed. If you have programs that you created with BASIC Compiler 2.00 that use this feature, they will fail with IBM BASIC Compiler/2. Use the FILEATTR function instead.

Examples

This example uses VARPTR to get the data from a variable. In line 30, P gets the address of the data. Integer data is stored in two bytes, with the least significant byte first. The actual value stored at location P is calculated in line 40. The bytes are read with the PEEK function, and the second byte is multiplied by 256 because it contains the high-order bits.

```
10 DEFINT A-Z
20 DATA1=500
30 P=VARPTR(DATA1)
40 V=PEEK(P) + 256*PEEK(P 1)
50 PRINT V
```

See also the example for “VARSEG Function.”

VARPTR\$

Function

Purpose

The VARPTR\$ function returns a character form of the offset of a variable in memory.

Format

$v\$ = \text{VARPTR\$}(\text{variable})$

Comments

variable is the name of a numeric or string variable or array element in your program. *variable* cannot be a record or a record element. You must assign a value to *variable* before you call VARPTR\$, or BASIC returns an **Illegal function call** error.

VARPTR\$ returns a three-byte string in the form:

Byte 0	Byte 1	Byte 2
<i>type</i>	low byte of variable address	high byte of variable address

The *type* indicates the variable type:

- 2 integer
- 20 long integer
- 3 string
- 4 single-precision
- 8 double-precision

The returned value is the same as:

$\text{VARPTR\$}(\text{variable}) = \text{CHR\$}(\text{type}) + \text{MKI\$}(\text{VARPTR}(\text{variable}))$

VARPTR\$ Function

You must use VARPTR\$ to indicate a variable name in the command string for PLAY or DRAW. For example:

```
PLAY "X"+VARPTR$(A$) PLAY "O="+VARPTR$(I)
```

Examples

The following example converts the value returned by VARPTR\$ into the type and offset of the variable.

```
A.PTR$ = varptr$ (a$)           'Get the information for A$

A.TYPE% = asc (left$ (A.ptr$,1)) 'Break off the type byte
SELECT CASE A.TYPE%
  CASE 2
    PRINT "A$ is short integer."
  CASE 20
    PRINT "A$ is a long integer."
  CASE 3
    PRINT "A$ is a string."
  CASE 4
    PRINT "A$ is a single-precision real number."
  CASE 8
    PRINT "A$ is a double-precision real number."
END SELECT

A.OFFSET% = CVI(RIGHT$(A.PTR$, 2)) 'Break off the offset bytes
PRINT "Its offset in the data segment is "A.OFFSET%
```

VARSEG

Function

Purpose

The VARSEG function returns the segment of a variable in memory.

Format

$v = \text{VARSEG}(\textit{variable})$

Comments

variable is the name of a numeric or string variable or array element in your program. The *variable* can also be a record or a record element.

VARSEG returns the segment of memory in which the compiler stores *variable*. VARSEG returns the segment number as an integer.

If you want to know the offset into the segment, use the VARPTR function.

VARSEG Function

Examples

This example uses VARSEG and VARPTR to get a pointer to a Dynamic Array:

```
' DIM Dynamic array:
'$DYNAMIC
DIM TestDynamic%(1 TO 5)

' Set the first elements of array:
TestDynamic%(1) = 6

' Get segment and offset to array:
DynamicPtr = VARPTR (TestDynamic%(1))
DynamicSeg = VARSEG (TestDynamic%(1))

' PEEK Value in array and compare it with original:
TestVal% = 256 * PEEK (DynamicPtr+1) + PEEK (DynamicPtr)
PRINT TestVal% "won't equal "TestDynamic%(1)

' Now, set the correct segment and try again:
DEF SEG=DynamicSeg
TestVal% = 256 * PEEK (DynamicPtr+1) + PEEK (DynamicPtr)
PRINT TestVal% "should equal "TestDynamic%(1)

END
```

VIEW

Statement

Purpose

The VIEW statement defines a rectangular subset of the screen onto which WINDOW and WINDOW contents are mapped.

This function is used in graphics mode only.

Format

VIEW [[SCREEN] (x1,y1)–(x2,y2) [, [attribute] [, boundary]]]

Comments

SCREEN If the SCREEN argument is included, all points plotted are absolute and can be inside or outside the screen limits. However, only those points within the viewport limits are visible. For example, if:

```
10 VIEW SCREEN (10,10)–(200,100)
```

is run, the point plotted by **PSET (0,0),3** does not appear on the screen because 0,0 is outside the viewport. **PSET (10,10),3** is within the viewport and places the point in the upper-left corner.

If the SCREEN argument is omitted, all points plotted are relative to the viewport. That is, *x1* and *y1* are added to the *x*- and *y*- coordinates before plotting the point on the screen. For example, if:

```
10 VIEW (10,10)–(200,100)
```

is run, the point plotted by **PSET (0,0),3** is at the screen location 10,10.

(x1,y1)–(x2,y2)

are the upper-left (x1,y1) and the lower-right (x2,y2) coordinates of the viewport defined. The *x*- and *y*- coordinates must be within the limits of the screen or an **Illegal func-**

VIEW Statement

tion call error occurs. For more information, see Graphics Modes” in *IBM BASIC Compiler/2 Fundamentals*.

attribute lets you fill the defined viewport with color. If *attribute* is omitted, the viewport is not filled. The *attribute* is an integer expression that chooses an attribute from the attribute range for the current screen mode. In SCREEN 1 (medium resolution), *attribute* can range from 0 through 3. In SCREEN 2 (high resolution), *attribute* can be 0 or 1.

The default color attribute for the foreground is the maximum color attribute for that screen mode.

The default color attribute for the background is always zero.

boundary lets you draw a boundary line around the viewport (if space is available). If *boundary* is omitted, no boundary is drawn. *boundary* is an integer expression in the range described in *attribute*.

It is important to note that VIEW sorts the x- and y- argument pairs, placing the smaller values for x and y first. For example:

```
VIEW (100,100)-(5,5)
```

becomes:

```
VIEW (5,5)-(100,100)
```

Another example:

```
VIEW (310,100)-(200,150)
```

becomes:

```
VIEW (200,100)-(310,150)
```

All possible pairings of x and y are valid. The only restriction is that x_1 cannot equal x_2 and y_1 cannot equal y_2 . The viewport cannot be larger than the screen.

VIEW with no arguments defines the entire screen as the viewport.

VIEW

Statement

You can define multiple viewports, but only one viewport can be active at a time. `RUN` and changes in `SCREEN` attributes disable the viewports.

`VIEW` used with `WINDOW` allows you to scale images. See the second example. See also the `WINDOW` statement.

Note: When `VIEW` is used, the `CLS` statement clears only the current viewport. To clear the entire screen, you must use `VIEW` to disable the viewport, and then use `CLS` to clear the screen.

Examples

The following example defines four viewports:

```
10 SCREEN 1:VIEW:CLS:KEY OFF
20 VIEW (1,1)-(151,91),,1
30 VIEW (165,1)-(315,91),,2
40 VIEW (1,105)-(151,195),,2
50 VIEW (165,105)-(315,195),,1
60 LOCATE 2,4:PRINT "Viewport 1"
70 LOCATE 2,25:PRINT "Viewport 2"
80 LOCATE 15,4:PRINT "Viewport 3"
90 LOCATE 15,25:PRINT "Viewport 4"
100 VIEW (1,1)-(151,91):GOSUB 1000
200 VIEW (165,1)-(315,91):GOSUB 2000
300 VIEW (1,105)-(151,195):GOSUB 3000
400 VIEW (165,105)-(315,195):GOSUB 4000
900 END
1000 CIRCLE (65,50),30,2
1010 'Draw a circle in first viewport
1020 RETURN
2000 LINE (45,50)-(90,75),1,B
2010 'Draw a box in second viewport
2020 RETURN
3000 FOR D=0 TO 360: DRAW "ta="+VARPTR$(d)+"nu20":NEXT
3010 'Draw spokes in third viewport
3020 RETURN
4000 PSET(60,50),2:DRAW "e15;f15;130"
4010 'Draw a triangle in fourth viewport
4020 RETURN
```

VIEW Statement

This example demonstrates scaling with VIEW:

```
10 KEY OFF:CLS: SCREEN 1,0:COLOR 0,0
20 WINDOW SCREEN(320,0)-(0,200)
30 GOTO 80
40 C=1
50 CIRCLE (160,100),60,C,,5\18
60 CIRCLE (160,100),60,C,,1
70 RETURN
80 GOSUB 40:FOR I=1 TO 1500:NEXT I: CLS
90 VIEW (1,1)-(160,90),,2:GOSUB 40
```

VIEW PRINT

Statement

Purpose

The VIEW PRINT statement sets the boundaries of the screen text window.

Format

VIEW PRINT [*row* TO *row*]

Comments

row is the number of a row on your display. It is an integer from 1 through 24. Line 25 is not used.

When you use the VIEW PRINT statement, PRINT prints output to your display only between the *rows* you specify.

Using VIEW PRINT without specifying the row parameters initializes the whole screen area as the text window.

WAIT Statement

Purpose

The WAIT statement suspends the program while monitoring the status of a machine input port.

This statement is not available in OS/2 mode.

Format

WAIT *port*, *n*[,*m*]

Comments

port is the port number, in the range 0 through 65535.

n, *m* are integer expressions in the range 0 through 255.

See your technical documentation for your computer for a description of valid port numbers (I/O addresses).

The WAIT statement suspends the program until a specified machine input port develops a specified bit pattern.

The data read at the port is XORed with the integer expression *m* and then ANDed with *n*. If the result is 0, BASIC loops back and reads the data at the port again. If the result is non-0, running continues with the next statement. If *m* is omitted, it is assumed to be 0.

The WAIT statement lets you test one or more bit positions on an input port. You can test the bit position for either a 1 or a 0. The bit positions to be tested are specified by setting 1's in those positions in *n*. If you do not specify *m*, the input port bits are tested for 1's. If you specify *m*, a 1 in any bit position in *m* (for which there is a 1 bit in *n*) causes WAIT to test for a 0 for that input bit.

WAIT

Statement

When run, the WAIT statement loops, testing those input bits specified by 1's in *n*. If *any one* of those bits is 1 (or 0 if the corresponding bit in *m* is 1), the program continues with the next statement. Thus WAIT does not wait for an entire pattern of bits to appear, but only for one of them to occur.

Note: It is possible to enter an infinite loop with the WAIT statement. You can do a Ctrl + Break or a System Reset to exit the loop.

Examples

This example waits for any key to be pressed. The key can still be read using any form of input (for example, INKEY\$).

```
100 WAIT &H60,&0
```

WHILE and WEND Statements

Purpose

The WHILE AND WEND statements run a series of statements in a loop as long as a given condition is true.

Format

```
WHILE expression  
    statements  
WEND
```

Comments

expression is any numeric expression.

If *expression* is true (not-0), *loop statements* run until the WEND statement is encountered. BASIC then returns to the WHILE statement and checks *expression*. If *expression* is still true, the process is repeated. If it is not true, running resumes with the statement following the WEND statement.

WHILE-WEND loops can be nested to any level. Each WEND will match the most recent WHILE. An unmatched WHILE statement causes a **WHILE without WEND** error, and an unmatched WEND statement causes a **WEND without WHILE** error.

WHILE and WEND Statements

Examples

The following example sorts the elements of array A into alphabetical order. A was defined with J-elements.

```
5 J=UBOUND(A,1)
10 'bubble sort array A
20 FLIPS=1 'force first pass thru loop
30 WHILE FLIPS
40 FLIPS=0
50 FOR I=1 TO J-1
60 IF A(I)>A(I+1) THEN SWAP A(I),A(I+1): FLIPS=1
70 NEXT I
80 WEND
```

WIDTH Statement

Purpose

The WIDTH statement sets the output line width in number of characters. After outputting the indicated number of characters, BASIC adds a carriage return.

Format

WIDTH *size*

WIDTH *device,size*

WIDTH *#filenum,size*

Comments

size is a numeric expression in the range 0 through 255. This is the new width. WIDTH 0 is the same as WIDTH 1.

device is a string expression for the device identifier. Valid devices are SCRN:, LPT1:, LPT2:, LPT3:, COM1:, and COM2:.

Note: The colons must be included as part of the device names.

filenum is a numeric expression in the range 1 through 127. This is the number of a file opened to an output device.

Depending on the device specified, the following actions are possible:

WIDTH *size* or WIDTH "SCRN:" *size*

Sets the screen width. Only 40- or 80-column widths are allowed. WIDTH 40 is not valid for the IBM Monochrome Display.

If the screen is in medium-resolution graphics mode (as occurs with a SCREEN 1 statement), WIDTH 80 forces the

WIDTH

Statement

screen into high resolution (as with a SCREEN 2 statement).
The reverse is true when in high resolution.

Note: Changing the screen width clears the screen and sets the border screen color to black.

WIDTH *device,size*

A deferred width assignment for the device. This form of WIDTH stores the new width value without changing the current width setting. A subsequent OPEN to the device uses this value for width while the file is open. The width does not change immediately if the device is already open.

Note: LPRINT, LLIST, LIST, and "LPTn" do an implicit OPEN and are therefore affected by this statement.

WIDTH *#filename,size*

The width of the device associated with *filename* is immediately changed to the new size specified. This allows the width to be changed at will while the file is open. This form of WIDTH has meaning only for LPT1:. The number sign (#) is required.

Any value entered outside the ranges indicated results in an **Illegal function call** error. The previous value is retained.

The width for each printer defaults to 80 when your program is started. The maximum width for the IBM Graphics Printer is 132. However, no error is returned for values between 132 and 255.

It is up to you to set the appropriate physical width on your printer. Some printers are set by sending special codes; some have switches. For the IBM Graphics Printer you should use LPRINT **CHR\$(15)**; to change to a condensed type style when printing at widths greater than 80. Use LPRINT **CHR\$(18)**; to return to normal. The IBM Graphics Printer is set up to automatically add a carriage return if you exceed the maximum line length.

Specifying a width of 255 disables line folding. This has the effect of "infinite" width. WIDTH 255 is the default for communications files.

WIDTH Statement

Changing the width for a communications file does not change either the receive or the transmit buffer; it just causes a carriage return character to be sent after every *size* characters.

Changing screen mode affects screen width only when moving between SCREEN 2 and SCREEN 1 or SCREEN 0. See "SCREEN Statement."

Examples

In this example, line 10 stores a printer width of 75 characters per line. Line 20 opens file #1 to the printer and sets the width to 75 for subsequent PRINT #1,... statements. Line 6020 changes the current printer width to 40 characters per line. Notice that the WIDTH value must come before the OPEN statement.

```
10 WIDTH "LPT1:",75
20 OPEN "LPT1:" FOR OUTPUT AS #1
.
.
.
6020 WIDTH #1,40
```

These examples change screen mode and width:

```
SCREEN 1,0      'Set to med-res color graphics
WIDTH 80       'Go to hi-res graphics
WIDTH 40       'Go back to medium res
SCREEN 0,1     'Go to 40x25 text color mode
WIDTH 80       'Go to 80x25 text color mode
```

WINDOW Statement

Purpose

The WINDOW statement redefines the coordinates of the viewport.

This statement is used in Graphics mode only.

Format

WINDOW [[SCREEN] (x1,y1)– (x2,y2)]

Comments

SCREEN controls the orientation of the screen coordinate system.

When SCREEN is omitted, the screen coordinates conform to the Cartesian coordinate system (x increases to the right, y increases upward).

When SCREEN is included, x increases to the right, and y increases downward.

(x1,y1),(x2,y2)

are programmer–defined coordinates called *world coordinates*. These coordinates are single-precision, floating-point numbers. They define the world coordinate space that is mapped into the physical coordinate space, as defined by the VIEW statement. See “VIEW Statement.”

WINDOW allows you to draw objects in space (“world coordinate system”) and not be bounded by the limits of the screen (“physical coordinate system”). This is done by specifying the world coordinate pairs (x1,y1) and (x2,y2). BASIC then converts world coordinate pairs for subsequent display within the viewport. To make this transformation from world space to the physical space of the screen, BASIC must know what portion of the unbounded world coordinate space contains

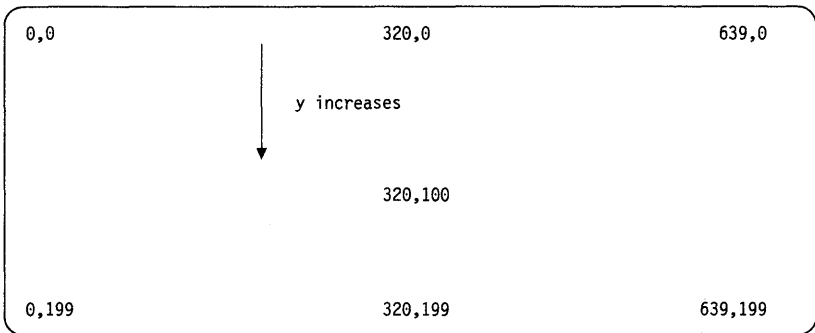
WINDOW Statement

the information you want to be displayed. This rectangular region in the world coordinate space is called a *window*.

In the physical coordinate system, if you run the following:

SCREEN 2

the screen appears with standard coordinates as:



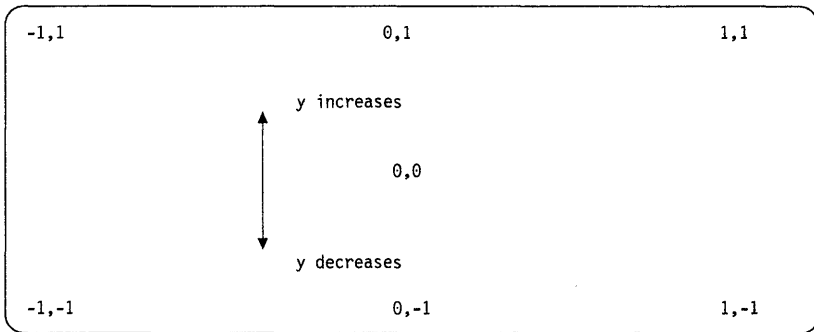
WINDOW

Statement

When WINDOW is used without the SCREEN attribute, the screen is viewed in true Cartesian coordinates. For example, given:

```
WINDOW (1,-1)-(-1,1)
```

the screen appears as:



You may be familiar with this method of specifying coordinates. Because the Cartesian coordinate system is widely known, many people consider the coordinate system used with the graphics statements to be “upside down.” The SCREEN attribute allows you to select the coordinate system you are most comfortable with.

It is important to note that WINDOW sorts the x - and y - argument pairs, placing the smaller values for x and y first. For example:

```
WINDOW (100,100)-(5,5)
```

becomes:

```
WINDOW (5,5)-(100,100)
```

Another example:

```
WINDOW (-4,4)-(4,-4)
```

becomes:

```
WINDOW (-4,-4)-(4,4)
```

WINDOW Statement

All possible pairings of x and y are valid. The only restrictions are that $x1$ cannot equal $x2$ and $y1$ cannot equal $y2$.

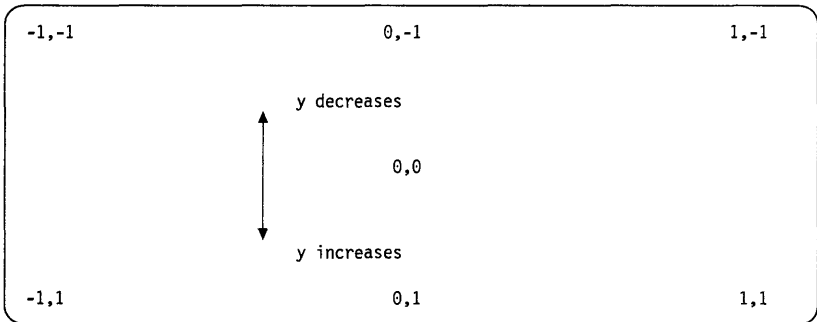
Note that the y coordinate is inverted so that $(x1,y1)$ is the lower-left coordinate and $(x2,y2)$ is the upper-right coordinate.

When the `SCREEN` attribute is included, with `WINDOW`, the coordinates are not inverted so that $(x1,y1)$ is the upper-left coordinate and $(x2,y2)$ is the lower-right coordinate.

For example:

```
WINDOW SCREEN (-1,-1)-(1,1)
```

defines the screen to look like this:



`WINDOW` also allows you to “zoom” and “pan.” Using a window with coordinates larger than an image displays the entire image, but the image is small and blank spaces appear on the sides of the screen. Choosing window coordinates smaller than an image forces clipping and allows only a portion of the image to be displayed and magnified. By specifying small and large window sizes, you can zoom in until an object occupies the entire screen, or you can zoom out until the image is just a spot on the screen.

`RUN`, `SCREEN`, and `WINDOW` with no attributes disable any `WINDOW` definitions and return the screen to physical coordinates.

WINDOW

Statement

Examples

The following example shows clipping using WINDOW:

```
10 SCREEN 2:CLS
20 WINDOW (-6,-6)-(6,6)
30 CIRCLE (4,4),5,1
40 'the circle is large and only part is visible
50 WINDOW (-100,-100)-(100,100)
60 CIRCLE (4,4),5,1 'the circle is very small
70 END
```

WINDOW Statement

The following example shows the effect of zooming using WINDOW:

```
10 KEY OFF:CLS: SCREEN 1,0
20 '
30 GOTO 160
40 '=====
50 'procedure display
60 '
70 LINE (X,0)-(-X,0),,,&HAA00 'create x axis
80 LINE (0,X)-(0,-X),,,&HAA00 'create y axis
90 '
100 CIRCLE (X/2,X/2),R 'circle has radius r
110 FOR P=1 TO 50:NEXT P 'delay loop
120 '
130 RETURN
140 '=====
150 '
160 X=1000:WINDOW (-X,-X)-(X,X):R=20
170 'create a graph with large coord range
180 GOSUB 50:FOR P=1 TO 1000:NEXT P:CLS
190 '
200 X=60:WINDOW (-X,-X)-(X,X):R=20
210 'smaller coord range increase circle size
220 GOSUB 50:FOR P=1 TO 1000:NEXT P:CLS
230 '
240 X=100:WINDOW (-X,-X)-(X,X):R=20
250 'modify window to show only portion of axes
260 GOSUB 50:FOR P=1 TO 1000:NEXT P:CLS
270 '
280 PRINT "... an +
example":PRINT" of +
zooming.."
290 FOR P=1 TO 1500:NEXT P
300 CLS:T=-50:U=100:X=U
310 FOR I=1 TO 45
320     T=T + 1:U=U - 1:X=X-1:R=20
330     WINDOW (T,T)-(U,U):CLS:GOSUB 50
340 NEXT I
350 END
```

WRITE

Statement

Purpose

The WRITE statement writes data to the screen.

Format

WRITE [*list of expressions*]

Comments

list of expressions

is a list of numeric and/or string expressions, separated by commas or semicolons.

If the list of expressions is omitted, a blank line is displayed. If the list of expressions is included, the values of the expressions are displayed on the screen.

When the values of the expressions are displayed, each item is separated from the one before it by a comma. Strings are delimited by quotation marks. After the last item in the list is printed, your program adds a carriage return/line feed.

WRITE is similar to PRINT. The difference between WRITE and PRINT is that WRITE inserts commas between the items as they are displayed and delimits strings with quotation marks. Also, positive numbers are not preceded by blanks.

WRITE Statement

Examples

The following example shows how WRITE displays numeric and string values:

```
10 A=80: B=90: C$="THAT'S ALL"  
20 WRITE A,B,C$
```

Results:

```
80,90,"THAT'S ALL"
```

WRITE # Statement

Purpose

The WRITE # statement writes data to a sequential file.

Format

WRITE #*filenum*, *list of expressions*

Comments

filenum is the number under which the file was opened for output.

list of expressions is a list of string and/or numeric expressions, separated by commas or semicolons.

The difference between WRITE # and PRINT # is that WRITE # inserts commas between the items as they are written and delimits strings with quotation marks. Therefore, it is not necessary for you to put explicit delimiters in the list. Also, WRITE # does not put a blank in front of a positive number. A carriage return/line feed sequence is inserted after the last item in the list is written.

Examples

Let A\$="CAMERA" and B\$="93604-1". The statement:

```
WRITE #1,A$,B$
```

writes the following image to the file:

```
"CAMERA","93604-1"
```

A subsequent INPUT # statement:

```
INPUT #1,A$,B$
```

inputs "CAMERA" to A\$ and "93604-1" to B\$.

Appendix A. BASIC Compiler Error Messages

During development of a BASIC program with the IBM BASIC Compiler/2, two kinds of errors may occur:

- Compile–time errors.
- Run–time errors.

The BASIC compile–time errors occur when you compile your program. The BASIC run–time errors only occur at the last step in the development process, when you actually run your compiled program. All these messages are listed in this appendix.

The first part of this appendix lists error codes and messages for the errors detected by the IBM BASIC Compiler/2. They are separated into two groups: prompt errors that issue a prompt describing the error and usually allow you to correct the error and continue with the compiling process, and listing errors that generally indicate an error in your program and appear in the compiler listing.

Errors While Compiling a Program

The errors in this section occur when you are compiling a source file to produce an object module.

Prompt Errors

The following errors from the compiler are severe errors, that is, they must be corrected before you can continue. When they occur, the sequence of prompts to start the compiler is restarted, giving you a chance to correct the error.

Message Meaning

Extra file name ignored

You entered too many file specifications.

This message is only a warning, but you should make sure the source, object, and source listing file names were used by the compiler as you expected.

Line invalid. Start again

An invalid filename character was used following the path characters “\” or “:”.

Enter the correct file specification.

The following long messages are compilation error messages. When they occur, you must correct the problem and start the compiler again from the beginning.

Message Meaning

BASIC fatal: Input file not found

The source file you named does not exist on the drive specified.

Check the file specification for the source file. If it is correct, insert the correct diskette and retry the operation.

Binary source file

The source file you specified to the compiler was not in ASCII format.

Make sure you specified the right file. If necessary, start the interpreter, load the file, and save it again with the A option.

/C: buffer size too large

The size that you specified for the communications buffer was too large.

The maximum size allowed is 32767 bytes.

Colon expected after /C

The colon after the /C parameter was missing.

Insert a colon after the /C parameter and retry the operation.

Buffer size expected after /C:

The desired size of the buffer for receiving communications data was missing.

Insert the number of bytes that you want to reserve for the communications buffer (an integer from 256 to 32767). If you omit the /C parameter, the compiler will reserve 256 bytes for the communications receive buffer.

Internal Error near xxxxx

An internal malfunction occurred in the IBM BASIC Compiler/2.

Recopy you compiler diskette. Check the hardware and retry the compile. If the error reoccurs, report the conditions under which the message appeared to your computer dealer.

Line nnnnn is undefined

A statement or command in the program refers to a line that does not exist.

Check the line references in your program so they all refer to actual program lines.

Memory Overflow

The compiler working memory is full. The program is too large to compile successfully.

Try compiling the program again with the */S* parameter to reduce compiler working memory requirements, or break the program up into smaller programs.

Missing NEXT for z

No NEXT statement was found for the variable z.

Correct the static nesting of your FOR and NEXT statements.

Out of memory

Your computer does not have enough work space to complete the requested task.

Read error on standard input

A system error occurred while reading the source file.

Unknown option /z

The only valid /Z options are */Zi* and */Zd*.

Option unknown

You have given an illegal option.

Unknown option /FP

The only valid /FP options are **/FPc**, **/FPc87**, and **/FPa**.

Unknown option /F

The only valid /F options are **/FPc**, **/FPc87**, and **/FPa**.

Unknown option /L

The only valid /L options are **/Lp** and **/Lc**.

Listing Errors

The compiler points to the errors it finds in your source listing file by displaying the line containing the error with an arrow beneath that line pointing to the place where the error occurred and a message describing the error. In some cases, the compiler reads ahead on a line to determine whether an error has actually occurred. In those cases, the arrow points a few characters beyond the error, or to the end of the line.

Some of the compile – time messages are only warnings; warnings do not need to be corrected before you go on to the linking step. If a message is a warning, it is noted in the explanation for the message. If the explanation does not say that the message is only a warning, the message indicates a severe error that must be corrected.

Message Meaning

\$INCLUDE file access error

The file specified in the \$INCLUDE metaccommand could not be found. Also, the \$INCLUDE metaccommand must be the last statement on the line.

\$Metaccommand error

The format of a metaccommand was invalid or included an invalid argument. The metaccommand is ignored. This message is only a warning.

Advanced feature error

You attempted to use a feature that is not available in the operating system or operating system mode for which you are compiling your program.

Array already dimensioned

You tried to define the size of the same array twice. This may happen in one of several ways:

- The same array is defined in two DIM statements.
- The program encounters a DIM statement for an array after the default dimension of 10 is established for that array.

- The program sees an `OPTION BASE` statement after an array has been dimensioned, either by a `DIM` statement or by default.

Array not dimensioned

Default dimensions were assigned to the array. This message is only a warning.

Array too big

There is not enough user data space to accommodate the array declaration.

Reduce the size of the array or use the `$DYNAMIC` metaccommand.

AS clause required on the first declaration

A variable that was not declared with an `AS` clause was later referenced with an `AS` clause.

AS clause required

A variable that was declared with an `AS` clause was later referenced without one.

If the first declaration of a variable has an `AS` clause, every subsequent `DIM`, `REDIM`, `SHARED`, and `COMMON` statement that references that variable must have an `AS` clause.

BYVAL only allowed with numeric arguments

You tried to pass a non – numeric argument to a subprogram with the `BYVAL` keyword.

Make sure that when you pass the actual value of a parameter (`BYVAL`), the argument is numeric.

CASE without SELECT

A `CASE` statement was encountered without a `SELECT`.

Make sure that the `CASE` statement block is preceded by the `SELECT CASE` expression.

Common array not dimensioned

A static array in a COMMON statement had not been dimensioned when the COMMON statement was encountered.

A static array passed in a COMMON statement must be defined in a DIM statement that precedes the COMMON statement.

COMMON out of order

The COMMON statement was found after executable statements in the program.

COMMON must precede any executable statements.

Data memory overflow

The program data is too big to fit in memory. This error is caused by too many constants, or too much array data.

Try turning off the debugging options. If memory is still exhausted, break your program into parts and use the CHAIN statement, or use the \$DYNAMIC metacommand.

Data type conflict

The variable is not of the required type (numeric or string). An array reference had an invalid dimension value (such as a string value).

DECLARE required

An implicit subprogram or function procedure call appeared before the procedure definition.

Implicit calls require the subprogram or function procedure to be declared before being called.

DEF without END DEF

A DEF FN statement does not have a corresponding END DEF statement. That is, a DEF FN function definition was active when the physical end of the program was reached.

Make sure that each DEF FN statement has a corresponding END DEF statement.

Divide by 0

You tried to divide by zero, or you had a divide overflow.

DO without LOOP

A DO statement does not have a corresponding LOOP. That is, a DO loop was active when the physical end of the program was reached.

Make sure that each DO statement has a corresponding LOOP.

Duplicate common variable

A variable appeared more than once in the COMMON statement(s) in the program.

Duplicate definition

A DECLARE, SUB, or FUNCTION statement contains information that conflicts with information that BASIC has about a subroutine or function.

Two common causes of this error are:

- Two DECLARE statements that do not match exactly were found for the same subroutine or function.
- A SUB or FUNCTION statement was found for a subroutine or function that was previously defined in a DECLARE statement that included "ALIAS" or "CDECL." ALIAS and CDECL can only be used with non-BASIC procedures.

Duplicate statement number

A duplicate line number was encountered.

Dynamic array element not allowed

Dynamic array elements are not allowed with VARPTR\$.

ELSE without IF

ELSEIF without IF

An ELSEIF was encountered without a corresponding IF.

END DEF without DEF

An END DEF statement without a corresponding DEF FN statement was encountered.

END IF without IF

An END IF was encountered without a corresponding IF.

END SELECT without SELECT

An END SELECT was encountered without a corresponding SELECT.

END SUB/FUNCTION without SUB/FUNCTION

An END SUB or END FUNCTION statement without a corresponding SUB or FUNCTION statement was encountered.

EXIT DO without DO

An EXIT DO was encountered without a corresponding DO.

EXIT FOR without FOR

An EXIT FOR was encountered without a corresponding FOR.

Expected "GOTO" or "GOSUB"

The compiler expected a GOTO or GOSUB statement.

Expecting simple or array variable

The compiler expected a variable argument.

Expression too complex

This error is caused when certain internal limitations are exceeded. For example during expression evaluation, strings that are not associated with variables are assigned temporary locations by the compiler. A large number of such strings can cause this error to occur.

Try simplifying expressions and assigning strings to variables.

Formal parameters not unique

A function or subprogram declaration contains duplicate parameters. For example, `SUB foo(a,b,c,a) STATIC`.

Fixed length strings not allowed

A fixed length string was encountered in a place where a variable length string is required.

Change the fixed length string to a variable length string.

FOR index variable already in use

The counter variable on a FOR statement is already in use.

Change the counter variable name.

FOR without NEXT

A FOR statement was encountered without a matching NEXT. That is, a FOR loop was active when the physical end of the program was reached.

Function already defined

You used `DEF FN` to define a function with the same name as a function previously defined in your program.

Function not defined

You called a function before defining it with the `DEF FN` statement.

Make sure the program executes the `DEF FN` statement before you use the function.

IF without END IF

The block format of the IF statement was used, and an `END IF` was not found.

Make sure that, if you use the block format of the IF statement, your block ends with `END IF`.

Illegal “.” in typed variable name

User defined type identifiers and element names cannot contain periods.

The period should only be used in a scalar variable name or as a record variable separator.

Illegal COMMON name

The block name of the COMMON statement was not a valid identifier.

The name can be an identifiers up to 40 characters long.

Illegal DEFxxx character specification

A DEFTYPE statement is entered incorrectly.

DEF can only be followed by LNG, DBL, INT, SNG, STR or (for user defined functions) a blank space.

Illegal FOR index variable

The FOR index variable was not valid.

The index variable must either be integer, long integer, single – precision, or double – precision.

Illegal formal parameter specification

There is an error in a function or subprogram parameter list.

Illegal function name

The function name was not a correct variable name.

The name of the user defined function must be a valid variable name, and must be preceded by FN.

Illegal outside of SUB, FUNCTION or DEF FN

An EXIT SUB, EXIT FUNCTION, or EXIT DEF statement was found that was not inside a SUB, FUNCTION, or DEF FN block, respectively.

Illegal separator

There is an illegal delimiting character in a PRINT USING or WRITE statement.

Use a semicolon or a comma as a delimiter.

Illegal subprogram name

The name of the subprogram was not a correct variable name.

The name can be up to 40 characters long, and this name cannot appear in any other SUB or FUNCTION statement.

Illegal subscript syntax

An array subscript contains a syntax error. For example, both string and integer data types were used as subscripts.

Illegal syntax

Caused by one of the following:

- Invalid argument name
- Invalid assignment target
- Invalid constant format
- Invalid format for statement number
- Invalid syntax
- Missing operand in expression
- Single variable only allowed

Illegal type character in numeric constant

A numeric constant contains an inappropriate type – declaration character.

Illegal TYPE element name

The element name in the TYPE statement was not valid.

Make sure that the element name is not an array and make sure that it does not have any explicit type characters, such as %, &, !, #, \$, or decimal points.

Illegal TYPE name

The type name in the TYPE statement was not valid.

Make sure that the type name does not have any explicit type characters, such as %, &, !, #, \$, or decimal points.

Incomplete control structure in IF..THEN..ELSE

An unmatched NEXT, WEND, END IF, END SELECT, or LOOP statement appears in a single line IF..THEN..ELSE statement.

Integer between 1 and 32767 required

The statement requires an integer argument.

Invalid character

The character was not in the BASIC character set.

The BASIC character set consists of alphabetic characters (A – Z), numeric characters (0 – 9), and special characters. See the “Character Set” section in the IBM BASIC Compiler/2 Fundamentals book for a complete list of the valid characters.

Label not defined:

A label that does not exist in the program was referred to in a command or statement.

Check the labels in your program, and use the correct label name.

Line too long

A line has too many characters.

The line must have 253 characters or fewer.

LOOP without DO

A LOOP was encountered without a corresponding DO.

Lower bound exceeds upper bound

The lower bound exceeds the upper bound defined in a DIM statement.

Math overflow

The result of a calculation is too large to be represented in BASIC number format.

Missing “”**

You used a variable length string in a TYPE statement.

Make sure that all string elements within the TYPE declaration are fixed length strings.

Missing “=”

Missing “/E” Switch

Your program included a RESUME line statement.

Recompile the program with the /E parameter. If the listing also contains a /X error, recompile using /X instead of /E.

Missing “/V” or “/W” Switch

The program contains event trapping statements.

Recompile the program using either of the event trapping parameters, /V or /W.

Missing “/X” Switch

Your program included a RESUME 0, RESUME, or RESUME NEXT statement.

Recompile the program with the /X parameter.

Missing “AS”

Missing “BASE”

Missing comma

Missing “GOSUB”

Missing “GOTO”

Missing “INPUT”

Missing left parenthesis

Missing line number or label

Missing minus sign

Missing right parenthesis

Missing semicolon

Missing slash

Missing “STATIC” on SUB/FUNCTION

Missing “SUB” or “FUNCTION”

Missing “THEN”

Missing “TO”

Missing “TYPE”

Must be first item on the line

Name too long

Identifiers cannot be longer than 40 characters.

Nested function definition

A function definition appears inside another function definition.

NEXT without FOR

A NEXT was encountered without a corresponding FOR.

Make sure that every NEXT statement has a corresponding FOR preceding it.

Only simple variables allowed

User – defined types and arrays are illegal in a READ or INPUT statement.

Overflow in numeric constant

A numeric constant was not within the range expected by the compiler, or an expression containing constants was calculated by the compiler and resulted in an overflow.

One way to correct this is to use single-precision constants instead of integer constants.

Parameter type mismatch

A subprogram parameter type does not match the DECLARE statement argument, or the calling argument.

Program memory overflow

You attempted to compile a program that has a code segment that is larger than 64K.

Split the program into subprograms and use the CHAIN statement.

SEG or BYVAL not allowed on CALLS

SEG or BYVAL keywords cannot be used with the CALLS statement. Use the CALL statement.

SELECT without END SELECT

A SELECT does not have a matching END SELECT. That is, a SELECT was still active when the physical end of the program was reached.

Correct the program so that each SELECT has a corresponding END SELECT.

Skipping forward to END TYPE statement

An error was found within the TYPE definition so the compiler skipped to the end of the TYPE declaration.

Statement Ignored

The statement was ignored by the compiler. It may be that the command is unimplemented. This message is only a warning.

String constant required for ALIAS

The DECLARE statement ALIAS keyword requires a string constant argument. String variables and expressions can not be used.

String assignment required

The string assignment is missing from an LSET or RSET statement.

String expression required

The statement requires a string expression argument.

String variable required

The statement requires a string variable argument.

SUB/FUNCTION without END SUB/FUNCTION

A SUB or FUNCTION statement does not have a corresponding END SUB or END FUNCTION statement. That is, a subroutine or function definition was active when the physical end of the program was reached.

Make sure that each SUB statement has a corresponding END SUB statement and that each FUNCTION statement has a corresponding END FUNCTION statement.

Subprogram error

Caused by one of the following:

- Subprogram definition error
- Subprogram already defined
- Incorrectly nested SUB/END SUB/EXIT SUB statements

Subprograms not allowed in control statements

Subprogram definitions are not allowed inside control constructs such as IF..THEN..ELSE and SELECT CASE.

Syntax error in numeric constant

A numeric constant is not properly formed.

Too many arguments in function call

The compiler has a limit of 60 arguments for a function call.

Too many dimensions

The compiler has a limit of 60 dimensions for an array.

Too many named COMMON blocks

The maximum number of named COMMON blocks permitted is 126.

Too many statement numbers

The maximum number of lines in the line list following an ON...GOTO/GOSUB statement is 255.

Too many TYPE definitions

The maximum number of user – defined types permitted is 240.

Too many variables for INPUT

The compiler has a limit of 60 variables in an INPUT statement.

Too many variables for LINE INPUT

Only one variable is allowed for LINE INPUT.

TYPE already defined

You used TYPE to define a variable type with the same name as a variable type that was previously defined in your program.

Make sure that the type name is not the same as another previously defined type name.

TYPE element already defined

You tried to define an element name twice in the same TYPE statement.

Make sure that the element name is not the same as another element name in that TYPE statement.

TYPE element not defined

You made reference to a TYPE element that has not been defined with the TYPE statement.

Make sure the element name is defined in a TYPE statement before you use the element.

TYPE more than 65535 bytes

You tried to create a variable type with the TYPE statement that was more than 65535 bytes.

TYPE not defined

You made reference to a variable type that has not been defined with the TYPE statement.

Make sure the type name is defined with a TYPE statement before you use the user – defined type.

TYPE statement improperly nested

User – defined type statements are not allowed in subprograms.

Typed variable not allowed in expression

Variables that are user – defined types are not allowed in expressions. For example, CALL foo(X), where X is a user – defined type.

Unexpected end of file in TYPE declaration

An end of file was encountered while processing a TYPE statement.

Unimplemented Command

The compiler did not implement the command. This message is only a warning.

Unrecognizable statement

The compiler cannot recognize the statement. It may be that you used a built-in function on the left side of an equal sign.

Variable already defined

You tried to define the same variable twice. This may happen in one of several ways:

- The same variable is defined in two DIM statements.
- The program encounters a DIM statement for a variable after the default type of single-precision has been established for that variable.
- The program encounters a DIM statement for a variable after that variable has been defined with the TYPE statement.
- The program sees a TYPE statement for the same variable after that variable has been defined, either by a DIM statement or by default.

Variable length string required

Only variable length strings are allowed in a FIELD statement.

Variable name is not unique

You attempted to define X as a user-defined type after X.Y had been used as a scalar.

Variable required here

The compiler expected a variable after an INPUT, LET, READ, or SHARED statement.

Variables must have the same type

Variables used in a SWAP statement must be of the same type.

WEND without WHILE

A WEND was encountered before a matching WHILE was executed.

Correct the program so that there is a WHILE for each WEND.

WHILE without WEND

A WHILE does not have a matching WEND. That is, a WHILE was still active when the physical end of the program was reached.

Correct the program so that each WHILE has a corresponding WEND.

Wrong number of arguments

You used an incorrect number of arguments with a BASIC subprogram or function.

Wrong number of dimensions

An array reference contained the wrong number of dimensions.

Wrong number of subscripts

An array was referenced with the wrong number of subscripts.

Make sure that the number of subscripts that the array was defined with, and the number of subscripts that you referenced the array with, are the same.

Errors while Running a Program

The following errors may occur when you run your compiled and linked program. The first group of errors can be trapped by using an ON ERROR statement. The error numbers match those issued by the BASIC interpreter. When an untrapped error occurs, the message is displayed followed by an address. If the /D, /E, or /X parameter was specified to the compiler, the number of the line in which the error occurred is displayed also.

Number Message

2 Syntax error

A string item was encountered in a DATA statement when the program wanted a numeric value.

Correct the DATA statement or the READ statement.

Or, you may have the wrong number of arguments in a COLOR, LOCATE, or SCREEN statement.

3 RETURN without GOSUB

A RETURN statement needs a previous unmatched GOSUB statement.

Correct the program. You probably need to put a STOP or END statement before the subroutine so the program does not “fall” into the subroutine code.

4 Out of DATA

A READ statement is trying to read more data than is in the DATA statements.

Correct the program so that there are enough constants in the DATA statements for all the READ statements in the program.

5 Illegal function call

A parameter that is out of range is passed to a system function. The error may also occur as the result of:

- A negative or unreasonably large subscript
- Trying to raise a negative number to a power that is not an integer
- A negative record number on GET or PUT (file)

- An improper argument to a function or statement (such as one that is out of the expected range for the parameter)
- Trying to concatenate strings where the result is more than 32767 characters long.

Correct the program. Refer to particular statement or function for more information.

6 **Overflow**

The magnitude of a number is too large to be represented in the required number format. Unlike the interpreter, the compiler always stops when this error occurs.

You may be able to change the order of operations in a calculation so the overflow does not occur; or you may have to restrict the range of numbers in the program to avoid the overflow. To correct integer overflow, you may try changing to single-precision or double-precision variables.

Note: As with the interpreter, if underflow occurs, the result is zero and execution continues without an error.

7 **Out of memory**

There is not enough free memory to allocate file buffers, communications buffers, and/or the music background buffer. Or you may be doing complex painting and have run out of work space.

9 **Subscript out of range**

You used an array element with a subscript that is outside the dimensions of the array, or you requested the LBOUND or UBOUND of a dimension that the array does not have.

Check the reference to the array variable.

10 **Duplicate Definition**

You tried to define the size of the same array twice. This may happen in one of several ways:

- The same array is defined in two DIM statements.
- The program encounters a DIM statement for an array after the default dimension of 10 is established for that array.

- The program sees an `OPTION BASE` statement after an array has been dimensioned, either by a `DIM` statement or by default.

11 Division by zero

In an expression, you tried to divide by zero, you tried to raise zero to a negative power, or you had an integer divide overflow.

13 Type mismatch

You gave a string value where a numeric value was expected, or you had a numeric value in place of a string value. This may occur in `DRAW` or `PLAY` with `VARPTR$`, or in a `PRINT USING` statement.

14 Out of string space

String variables exceed the amount of remaining free string space after housecleaning.

16 String formula too complex

A string expression is too long or too complex.

The expression should be broken into smaller expressions, or fewer variables should be requested in the input statements.

19 No RESUME

The physical end of the program was encountered while the program was in an error trapping routine.

Correct the error trapping routine so a `RESUME` statement runs. Or you may want to add an `ON ERROR GOTO 0` statement to the error trapping routine so `BASIC` will display the message for any uncaught error.

20 RESUME without error

The program has encountered a `RESUME` statement without having trapped an error. The error trapping routine should only be entered when an error occurs or an `ERROR` statement runs.

You probably need to include a `STOP` or `END` statement before the error trapping routine to prevent the program from “falling into” the error trapping code.

24 Device Timeout

`BASIC` did not receive information from an input/output device within a predetermined amount of time.

For a communications file, this indicates that one of the signals tested by OPEN "COM... is off.

25 Device Fault

A hardware error indication was returned by an interface adapter.

For communications files, this error may also occur when one of the signals tested by OPEN "COM... is lost.

27 Out of Paper

The printer is out of paper, or the printer is not switched on.

You should insert paper (if necessary), verify that the printer is properly connected, and make sure that the power is on. Then restart the program or continue the error trapping routine.

39 CASE ELSE expected

No block in a CASE qualifies and there is no CASE ELSE.

50 FIELD overflow

A FIELD statement is attempting to allocate more bytes than were specified for the record length of a random file in the OPEN statement. Or the end of the FIELD buffer is encountered while doing sequential I/O (PRINT #, WRITE #, INPUT #) to a random file.

Check the OPEN statement and the FIELD statement to make sure they correspond. If you are doing sequential I/O to a random file, make sure that the length of the data read or written does not exceed the record length of the random file.

51 Internal error

An internal malfunction occurred in the IBM BASIC Compiler/2 runtime routines.

Recopy your compiler diskette. Check the hardware and retry the compile. If the error occurs again, report the conditions under which the message appeared to your computer dealer.

- 52 **Bad file number****
A statement uses a file number of a file that is not open, or the file number is not in the range 1 to 127. Or, the device name in the file specification is too long or invalid, or the file name was too long or invalid.
- Make sure the file you wanted was opened and that the file number was entered correctly in the statement. Check that you have a valid file specification (refer to “Naming Files” in *IBM BASIC Compiler/2 Fundamentals*. for information on file specifications).
- 53 **File not found****
A KILL, NAME, FILES, or OPEN statement refers to a file that does not exist on the disk in the specified drive.
- Verify that the correct diskette is in the drive specified, and that the file specification was entered correctly. Then retry the operation.
- 54 **Bad file mode****
You tried to use PUT or GET with a sequential file or a closed file; or to run an OPEN with a file mode other than input, output, append, or random.
- Make sure the OPEN statement was entered and run properly. GET and PUT require a random file.
- 55 **File already open****
You tried to open a file for sequential output or append, and the file is already opened; or, you tried to use KILL on a file that is open.
- Make sure you only run one OPEN to a file if you are writing to it sequentially. Close a file before you use KILL.
- 56 **Field statement active****
You attempted a GET or PUT with a TYPED record, but a FIELD statement was active.
- 57 **Device I/O Error****
An error occurred on a device I/O operation. DOS cannot recover from the error.
- This error may occur with communications files from overrun, framing, break, or parity errors. If you are communicating with 7 or fewer data bits, the eighth is turned on in the byte in error.

- 58 File already exists**
The file name specified in a NAME statement matches a file name already in use on the diskette.
Retry the NAME command using a different name.
- 59 Bad record length**
The length of the record used in a GET or PUT operation is incorrect.
- 61 Disk full**
All storage space on the disk is in use. Files are closed when this error occurs.
If there are any files on the disk that you no longer need, erase them or use another disk. Then rerun the program.
- 62 Input past end**
This is an end of file error. An input statement is run for a null (empty) file, or after all the data in a sequential file was already input.
To avoid this error, use the EOF function to detect the end of file.
This error also occurs if you try to read from a file that was opened for output or append. If you want to read from a sequential output (or append) file, you must close it and open it again for input.
- 63 Bad record number**
In a PUT, GET, LOCK, or UNLOCK statement, the record number is equal to zero.
Correct the statement to use a valid record number.
- 64 Bad file name**
An invalid form is used for the file name with BLOAD, BSAVE, KILL, OPEN, NAME, or FILES.
Check "Naming Files" in *IBM BASIC Compiler/2 Fundamentals*. For information on valid file names, and correct the file name in error.
- 67 Too many files**
An attempt is made to create a new file (using OPEN) when all directory entries on the disk are full, or when the file specification is invalid.

If the file specification is okay, use a new formatted diskette and retry the operation.

68 Device Unavailable

You tried to open a file to a device that does not exist. Either you do not have the hardware to support the device (such as printer adapters for a second or third printer), or you have disabled the device.

69 Communication buffer overflow

A communication input statement was run, but the input buffer was already full. You should use an `ON ERROR` statement to retry the input when this condition occurs. Subsequent inputs attempt to clear this fault unless characters continue to be received faster than the program can process them. If this happens there are several possible solutions:

- Increase the size of the communications buffer using the **RB** option of the `OPEN "COM..."` statement or the **IC** parameter when you start the IBM BASIC Compiler/2.
- Implement a "hand-shaking" protocol with the other computer to tell it to stop sending long enough so you can catch up.
- Use a lower baud rate to transmit and receive.

70 Permission Denied

You tried to write to a diskette that is write-protected. Make sure you are using the right diskette. If so, remove the write protection, then retry the operation.

Or, you attempted to write or read a record that has been LOCKED by another process. Retry the process when the other process has unlocked the record.

Or, during an `OPEN`, you violated one of the sharing attributes of the file you are opening. Retry the `OPEN` with the correct sharing attribute.

71 Disk not Ready

The diskette drive door is open or a diskette is not in the drive.

- 72 Disk Media Error**
The controller attachment card detected a hardware or media fault. Usually, this means that the diskette has gone bad. Copy any existing files to a new diskette and re-format the bad diskette. If formatting fails, the diskette should be discarded.
- 73 Advanced Feature**
You tried to use a feature not available with this compiler.
- 74 Rename Across Disks**
You tried to rename a file but specified the wrong disk. The NAME operation is not performed.

When you use NAME, the drive you specify must be the same for the the old file name and the new file name. The exception to this is when the DOS ASSIGN command is active. The drive can be logically different, but must be the same physical drive.
- 75 Path/file access error**
During an OPEN, NAME, MKDIR, CHDIR, or RMDIR operation, an attempt was made to use a path or file name to an inaccessible file. For example, you tried to open a directory or volume identifier; you tried to open a read only file for writing; or you tried to remove the current directory. The operation is not completed.

You attempted to read or write to a file opened by another process which has denied read or write access to other processes.

No additional file handles are available.
- 76 Path not found**
During an OPEN, MKDIR, CHDIR, or RMDIR operation, the operating system is unable to find the path the way it is specified. The operation is not completed.

The following error messages are not numbered.

— **Incorrect DOS version**

Check *IBM BASIC Compiler/2 Fundamentals* and be sure you are using a correct version of DOS.

— **Unprintable error**

This message occurs whenever an error message is not available for the error condition that exists. This is usually caused by an ERROR statement with an undefined error code.

Check your program to make sure you handle all error codes that you create.

Errors that Cannot be Trapped

The following additional run-time error messages are unrecoverable and cannot be trapped:

Message Meaning

dos memory – arena error

While loading the run-time module or while CHAINING, the DOS memory management mechanism has been detected to be corrupt. This could be due, among other things, to POKES or BLOADS into improper locations, to errors in assembly or other language code, or to programs that improperly affect memory.

Error during runtime initialization

There is insufficient memory to initialize the program.

Error in CHAIN file format

The indicated file is in the wrong format. It should be an executable (.EXE) file. This error may also occur when a program that uses the run-time module tries to chain to a executable program which does not use the run-time module.

Error in EXE file

The indicated file is in the wrong format. It should be an executable (.EXE) file. This may happen with RUN, CHAIN, and when loading the run-time module.

This error also occurs when a program that uses the run-time module tries to chain to a executable program which does not use the run-time module.

Far heap corrupt

BASIC's far memory manager for Dynamic Arrays has detected that the memory it manages has become corrupt. This could be due, among other things, to POKES or BLOADS into improper locations, to errors in assembly or other language code, or to programs that improperly affect memory.

No line number

This error occurs when the error address cannot be found in the line number table during error trapping.

Out of memory

Your computer does not have enough work space to complete the requested task.

Out of stack space

Your computer does not have enough stack space to perform the call to the SUB, FUNCTION, DEF FN, or GOSUB procedure.

Out of memory during CHAIN

Your computer does not have enough work space to perform the CHAIN statement.

Requires DOS 2.10 or later

IBM BASIC Compiler/2 will not run on versions of DOS prior to DOS 2.10.

Restart your computer with DOS 2.10 or a later version and try the operation again.

String space corrupt

This error usually occurs because a string descriptor has been improperly modified.

Communication Errors

Errors occur on communication files in the following order:

1. When opening the file:
 - a. **Device timeout** – if one of the signals to be tested (CTS, DSR, or CD) is missing.
2. When reading data:
 - a. **Com buffer overrun** – if an overflow occurs.
 - b. **Device I/O error** – for overrun, break, parity, or framing errors.
 - c. **Device fault** – if you lose DSR or CD.
3. When writing data:
 - a. **Device fault** – if you lose CTS, DSR, or CD on a modem status interrupt while BASIC was doing something else.
 - b. **Device timeout** – if you lose CTS, DSR, or CD while waiting to put data in the output buffer.

















































Appendix B. ASCII Character Codes

The following table lists all the ASCII codes (in decimal) and their associated characters. These characters can be displayed using `PRINT CHR$(n)` where *n* is the ASCII code.

Some of the entries for ASCII codes 0 to 31 and 135 to 155 also indicate the control function associated with that character. For example, ASCII code 27 displays a left arrow, and is also recognized as an ESCAPE control character.

Each of these characters can be entered from the keyboard by pressing and holding the Alt key, then pressing the digits for the ASCII code on the numeric keypad. Note, however, that some of the codes have special meaning to the BASIC program editor supplied with the interpreter. The program editor uses its own interpretation for the codes and might not display the special character listed here.

000	001	002	003	004	005	006	007	008	009
NUL	☺ SOH	☹ STX	♥ ETX	♦	♣	♠	BEL	◼ BS	HT
010	011	012	013	014	015	016	017	018	019
LF	VT	FF	CR	🎵 SO	☀ SI	▶	◀	↕ DC2	!! DC3
020	021	022	023	024	025	026	027	028	029
⏏ DC4	§ NAK	▬ SYN	↕ ETB	↑ CAN	↓ EM	→ SUB	← ESC	FS	GS
030	031	032	033	034	035	036	037	038	039
RS	US	SP	!	"	#	\$	%	&	,
040	041	042	043	044	045	046	047	048	049
()	*	+	,	-	.	/	0	1
050	051	052	053	054	055	056	057	058	059
2	3	4	5	6	7	8	9	:	;
060	061	062	063	064	065	066	067	068	069
<	=	>	?	∩	A	B	C	D	E
070	071	072	073	074	075	076	077	078	079
F	G	H	I	J	K	L	M	N	O
080	081	082	083	084	085	086	087	088	089
P	Q	R	S	T	U	V	W	X	Y
090	091	092	093	094	095	096	097	098	099
Z	[\]	^	_	`	a	b	c
100	101	102	103	104	105	106	107	108	109
d	e	f	g	h	i	j	k	l	m
110	111	112	113	114	115	116	117	118	119
n	o	p	q	r	s	t	u	v	w
120	121	122	123	124	125	126	127	128	129
x	y	z	{		}	~	⏏	Ç	ü

130	131	132	133	134	135	136	137	138	139
é	â	ä	à	å	Ç BEL	ê	ë	è	ï
140	141	142	143	144	145	146	147	148	149
î FF	ì CR	Ä SO	Â SI	É	æ	Æ DC2	ô	ö DC4	ò
150	151	152	153	154	155	156	157	158	159
û	ù	ÿ CAN	ö	ü	Φ ESC	£	¥	P _t	f
160	161	162	163	164	165	166	167	168	169
á	í	ó	ú	ñ	Ñ	<u>a</u>	<u>o</u>	¿	¬
170	171	172	173	174	175	176	177	178	179
¬	½	¼	¡	<<	>>				
180	181	182	183	184	185	186	187	188	189
									
190	191	192	193	194	195	196	197	198	199
									
200	201	202	203	204	205	206	207	208	209
									
210	211	212	213	214	215	216	217	218	219
									
220	221	222	223	224	225	226	227	228	229
				α	β	Γ	Π	Σ	σ
230	231	232	233	234	235	236	237	238	239
μ	τ	Φ	Θ	Ω	δ	∞	∅	€	∩
240	241	242	243	244	245	246	247	248	249
≡	±	≥	≤	∩	J	÷	≈	◦	•
250	251	252	253	254	255				
•	√	∩	∩		SP				

Appendix C. Scan Codes

The following table lists the scan code, in decimal and in hexadecimal, for each key on the IBM Personal Computer keyboard and the IBM Enhanced keyboard.

Key	Hex Code	Dec. Code
ESC	01	01
! 1	02	02
@ 2	03	03
# 3	04	04
\$ 4	05	05
% 5	06	06
^ 6	07	07
& 7	08	08
* 8	09	09
(9	0A	10
) 0	0B	11
_ -	0C	12
+ =	0D	13
←	0E	14
← →	0F	15
Q	10	15
W	11	17
E	12	16
R	13	19
T	14	20
Y	15	21
U	16	22
I	17	23

Key	Hex Code	Dec. Code
O	18	24
P	19	25
{ [1A	26
}]	1B	27
← ↵	1C	28
Ctrl	1D	29
A	1E	30
S	1F	31
D	20	32
F	21	33
G	22	34
H	23	35
J	24	36
K	25	37
L	26	38
: ;	27	39
" '	28	40
~ `	29	41
shift left	2A	42
\	2B	43
Z	2C	44
X	2D	45
C	2E	46

Key	Hex Code	Dec. Code
V	2F	47
B	30	48
N	31	49
M	32	50
< ,	33	51
> .	34	52
? /	35	53
shift right	36	54
PrtSc*	37	55
Alt	38	56
Space Bar	39	57
Caps Lock	3A	58
F1	3B	59
F2	3C	60
F3	3D	61
F4	3E	62
F5	3F	63
F6	40	64
F7	41	65
F8	42	66
F9	43	67
F10	44	68

Key	Hex Code	Dec. Code
F11+	85	133
F12+	86	134
Num Lock	45	69
Scroll Lock	46	70
7 Home	47	71
8 ↑	48	72
9 Pg Up	49	73
—	4A	74
4 ←	4B	75
5	4C	76
6 →	4D	77
+	4E	78
1 End	4F	79
2 ↓	50	80
3 Pg Dn	51	81
0 Ins	52	82
. Del	53	83

+ - Only supported on keyboards with more than ten function keys.

Extended Codes

For certain keys or key combinations that cannot be represented in standard ASCII code, an extended code is returned by the INKEY\$ system variable. A null character (ASCII code 000) is returned as the first character of a two-character string. If a two-character string is received by INKEY\$, you should go back and examine the second character to determine the actual key pressed. Usually, but not always, this second code is the scan code of the primary key that was pressed. The ASCII codes (in decimal) for this second character, and the associated key(s) are listed below.

Key	Hex Code	Dec. Code
Nul	03	03
shift tab	0F	15
a + Q	10	16
a + W	11	17
a + E	12	18
a + R	13	19
a + T	14	20
a + Y	15	21
a + U	16	22
a + I	17	23
a + O	18	24
a + P	19	25
a + A	1E	30
a + S	1F	31
a + D	20	32

Key	Hex Code	Dec. Code
a + F	21	33
a + G	22	34
a + H	23	35
a + J	24	36
a + K	25	37
a + L	26	38
a + Z	2c	44
a + X	2D	45
a + C	2E	46
a + V	2F	47
a + B	3B	59
a + N	31	49
a + M	32	50
F1	3B	59
F2	3C	60
F3	3D	61
F4	3E	62
F5	3F	63
F6	40	64
F7	41	65
F8	42	66
F9	43	67
F10	44	68

Key	Hex Code	Dec. Code
Home	47	71
PgUp	49	73
End	4F	79
PgDn	51	81
Ins	52	82
Del	53	83
s+F1	54	84
s+F2	55	85
s+F3	56	86
s +F4	57	87
s+F5	58	88
s+F6	59	89
s+F7	5A	90
s+F8	5B	91
s+F9	5C	92
s+F10	5D	93
c+F1	5E	94
c+F2	5F	95
c+F3	56	86
c+F4	61	97
c+F5	62	98
c+F6	63	99
c+F7	64	100
c+F8	65	101

Key	Hex Code	Dec. Code
c+F9	66	102
c+F10	67	103
a+F1	68	104
a+F2	69	105
a+F3	6A	106
a+F4	6B	107
a+F5	6C	108
a+F6	6D	109
a+F7	6E	110
a+F8	6F	111
a+F9	70	112
a+F10	71	113
c+PrtSc	72	114
c+←	73	115
c+→	74	116
c+End	75	117
c+PgDn	76	118
c+Home	77	119
a+1	78	120
a+2	79	121
a+3	7A	122
a+4	7B	123
a+5	7C	124
a+6	7D	125

Key	Hex Code	Dec. Code
a+7	7E	126
a+8	7F	127
a+9	80	128
a+0	81	129
a+-	82	130
a+=	83	131
c+PgUp	84	132
F11	85	133
F12	86	134
s+F11	87	135
s+F12	88	136
c+F11	89	137
c+F12	8a	138
a+F11	8B	139
a+F12	8C	140

Appendix D. CodeView Error Messages

CodeView displays an error message whenever it detects a command it cannot run. You might see any of the following error messages. Except for start-up errors, most errors stop the CodeView command in which the error occurred, but do not stop CodeView.

Bad address

You specified the address in an non-valid form. For example, you might have entered an address containing hexadecimal characters when the radix is decimal.

Bad breakpoint command

You typed an non-valid breakpoint number with the BREAKPOINT CLEAR, BREAKPOINT DISABLE, or BREAKPOINT ENABLE command. The number must be in the range of 0 through 19.

Bad flag

You specified an non-valid flag mnemonic with the REGISTER dialog command (**R**). Use one of the mnemonics that appears when you enter the command **RF**.

Bad format string

You specified a non-valid type specifier following an expression. Expressions used with the DISPLAY EXPRESSION, WATCH, WATCHPOINT, and TRACEPOINT commands can have **printf** type specifiers set off from the expression by a comma. The valid type specifiers are **d**, **i**, **u**, **o**, **x**, **X**, **f**, **e**, **E**, **g**, **G**, **c**, and **s**. Some type specifiers can be preceded by the prefix **h** or **l**.

Bad radix (use 8, 10, or 16)

CodeView only uses octal, decimal, and hexadecimal radices.

Bad register

You typed the REGISTER command (**R**) with an non-valid register name. Use **AX**, **BX**, **CX**, **DX**, **SP**, **BP**, **SI**, **DI**, **DS**, **ES**, **SS**, **CS**, **IP**, or **F**.

Bad type cast

The valid types for type-casting are the BASIC types integer, long integer, single—precision, double—precision, and string. These types are listed and explained in *IBM BASIC Compiler/2 Fundamentals*.

Bad type (use one of 'ABDILSTUW')

The valid dump types are ASCII (**A**), byte (**B**), integer (**I**), unsigned (**U**), word (**W**), doubleword (**E**), short real (**S**), long real (**L**), and ten-byte real (**T**).

Badly formed type

The type information in the symbol table of the file you are debugging is incorrect. If this message occurs, please note the circumstances of the error and report it.

Breakpoint "# or *" expected

You entered the BREAKPOINT CLEAR (**BC**), BREAKPOINT DISABLE (**BD**), or BREAKPOINT ENABLE (**BE**) commands with no argument. These commands require that you specify the number of the breakpoint at which CodeView is to act or that you specify an asterisk (*), indicating that CodeView is to act on all breakpoints.

Cannot use struct or union as scalar

You cannot use a structure or union variable as a scalar value in a BASIC expression. The address-of operator must precede structure or union variables, and a field specifier must follow them.

Can't find filename

CodeView cannot find the executable file you specified when you started. You probably misspelled the file name, or the file is in a different directory.

Constant too big

CodeView cannot accept a constant number larger than 4294967295 (0xFFFFFFFF).

Divide by zero

An expression in an argument of a dialog command attempts to divide by zero.

Expression too complex

An expression given as a dialog command argument is too complex. Simplify the expression.

Extra input ignored

You specified too many arguments to a command. CodeView evaluates the valid arguments and ignores the rest. Often in this situation, CodeView does not evaluate the arguments in the order that you intended.

Floating point error

This message should not occur, but, if it does, please note the circumstances of the error and report it.

Internal debugger error

If this message occurs, please note the circumstances of the error and report it.

Invalid argument

One of the arguments you specified is not a valid CodeView expression.

Missing "

You specified a string as an argument to a dialog command, but you did not supply a closing double quote mark.

Missing ')'

You specified an argument to a dialog command as an expression containing a left parenthesis but no right parenthesis.

Missing '{'

You specified an argument to a dialog command as an expression containing a right parenthesis but no left parenthesis.

Missing '['

You specified an argument to a dialog command as an expression containing a right bracket but no left bracket. This error can also occur if you specify a regular expression with a right bracket but no left bracket.

No closing single quote

You specified a character in an expression used as a dialog command argument, but the closing single quote is missing.

No code at this line number

You tried to set a breakpoint on a source line that does not correspond to code. The line might be a data declaration or a comment.

No match of regular expression

CodeView can find no match for the regular expression you specified with the **SEARCH** command or with the **Find** selection from the **Search** menu.

No previous regular expression

You selected **Previous** from the **Search** menu, but there was no previous match for the last regular expression specified.

No program to debug

You have run to the end of the program you are debugging. You must restart the program (using the `RESTART` command) before using any command that runs code.

No source lines at this address

The address you specified as an argument for the `VIEW` command (**V**) does not have any source lines. It might be an address in a library routine or an assembly-language module.

No such file/directory

A file you specified in a command argument or in response to a prompt does not exist. For example, this message appears when you select **Load** from the **File** menu and then enter the name of a nonexistent file.

No symbolic information

The program file you specified is not in the CodeView format. You cannot debug in source mode, but you can use assembly mode.

Not a text file

You attempted to load a file using the **Load** selection from the **File** menu or using the `VIEW` command, but the file is not a text file. CodeView determines if a file is a text file by checking the first 128 bytes for characters that are not in the ASCII range of 9 through 13 and 20 through 126.

Not an executable file

The file you specified for debugging when you started CodeView is not an executable file having the extension `.EXE` or `.COM`.

Not enough space

You typed the `SHELL ESCAPE` command (**!**) or selected **Shell** from the **File** menu, but there is not enough free storage to run `COMMAND.COM`. Because storage is released by code in the BASIC start-up routines, this error always occurs if you try to use the `SHELL ESCAPE` command before you have run any code. Use any of the code run commands (`TRACE`, `PROGRAM STEP`, or `GO`) to run the BASIC start-up code, then try the `SHELL ESCAPE` command again. The message also occurs with assembly-language programs that do not specifically release storage.

Object too big

You entered a TRACEPOINT command with a data object, such as an array, that is larger than 128 bytes. You can watch data objects larger than 128 bytes using the storage version of the TRACEPOINT command.

Operand types incorrect for this operation

An operand in a BASIC expression had a type that is incompatible with the operation applied to it. For example, if you declare **p** as **char ***, then **? p*p** produces this error because a pointer cannot be multiplied by a pointer.

Operator must have a struct/union type

You used the one of the member selection operators (**->** or **.**) in an expression that does not refer to an element of a structure or a union.

Operator needs lvalue

You specified an expression that does not evaluate to an lvalue in an operation that requires an lvalue. For example, **? 3 = 100** is non-valid. See the *IBM BASIC Compiler Fundamentals* book for more information on lvalues.

Program terminated normally (*number*)

You ran your program to the end. The number displayed in parentheses is the exit code that your program returns to DOS. You must use the RESTART command (or the **Start** menu selection) to start the program before running more code.

Register variable out of scope

You tried to specify a register variable using the period (**.**) operator and a function name. For example, if you are in a third-level function, you can display the value of a local variable called **local** in a second-level function called **parent** with the following command:

```
? parent.local
```

However, this command does not work if you declare **local** as a register variable.

Regular expression too complex

The regular expression you specified is too complex for CodeView to evaluate.

Regular expression too long

The regular expression you specified is too long for CodeView to evaluate.

Syntax error

You specified an non-valid command line for a dialog command. Check for an non-valid command letter. This message also appears if you enter an non-valid assembly-language instruction using the ASSEMBLE command. The error follows a caret that points to the first character that CodeView cannot interpret.

Too many breakpoints

You tried to specify a 21st breakpoint. CodeView permits only 20 breakpoints.

Too many open files

You do not have enough file handles for CodeView to operate correctly. You must specify more files in your CONFIG.SYS file. See your *IBM Personal Computer Disk Operating System Version 3.30 Reference* book for information about using the CONFIG.SYS file.

Type conversion too complex

You tried to type cast an element of an expression in a type other than the simple types or with more than one level of indirection. An example of a complex type is type casting to a structure or union type. An example of two levels of indirection is **char ****.

Unable to open file

CodeView cannot open a file that you specified in a command argument or in response to a prompt. For example, this message appears when you select **Load** from the **File** menu and then enter the name of a file that is corrupted or has its file attributes set so that it cannot be opened.

Unknown symbol

You specified an identifier that is not in CodeView's symbol table. Check for a misspelling. CodeView cannot recognize a symbol name spelled with letters of the wrong case unless you turn off the **Case Sense** selection on the **Options** menu. Another potential cause for this message is if you try to use a local variable in an argument when you are not in the function in which you define the variable.

Unrecognized option *option* - The valid options are /B, /Ccommand, /F, /M, /S, /T, /W, or /43

You entered an non-valid option when starting CodeView. Retype the command line.

Usage: cv [options] file [arguments]

You failed to specify an executable file when you started CodeView. Try again with the syntax shown in the message.

Video mode changed without /S option

The program changed video modes from or to one of the graphics modes when screen swapping was not specified. You must use the /s option to specify screen swapping when you are debugging graphics programs. You can continue debugging when you get this message, but the output screen of the debugged program might be damaged.

Warning: packed file

You started CodeView with a packed file as the executable file. You can attempt to debug the program in assembly mode, but the packing routines at the start of the program might make this difficult. You cannot debug in source mode because EXEPACK strips all symbolic information from a file when it packs the file. This occurs with the /EXEPACK linker option.

Appendix E. Linker Error Messages and Limits

This section lists error messages produced by the IBM Linker.

Fatal errors cause the linker to stop running. Fatal error messages have the following format:

location: **error L1xxx**: *message text*

Non-fatal errors indicate problems in the executable file. LINK produces the executable file (and sets the error bit in the header if for OS/2). Non-fatal error messages have the following format:

location: **error L2 xxx**: *message text*

Warnings indicate possible problems in the executable file. LINK produces the executable file (it does not set the error bit in the header if for OS/2). Warnings have the following format:

location: **error L4xxx**: *message text*

In these messages, *location* is the input file associated with the error, or LINK if there is not input file. If the input file is a module definitions file, the line number will be included, as shown below:

foo.def(3): fatal error L1030: missing internal name

If the input file is an .OBJ or .LIB file and has a module name, the module name is enclosed in parentheses, as shown in the following examples:

SLIBC.LIB(_file)
MAIN.OBJ(main.c)
TEXT.OBJ

The following error messages may appear when you link object files with LINK.

L1001 option : option name ambiguous

A unique option name does not appear after the option indicator (/). For example, the command

```
LINK /N main;
```

produces this error, since LINK cannot tell which of the three options beginning with the letter **N** is intended.

L1002 option : unrecognized option name

An unrecognized character followed the option indicator (/), as in the following example:

```
LINK /ABCDEF main;
```

L1003 option : MAP symbol limit too high

The specified symbol limit value following the MAP option is greater than 32767, or there is not enough memory to increase the limit to the requested value.

L1004 option : invalid numeric value

An incorrect value appeared for one of the linker options. For example, a character string is entered for an option that requires a numeric value.

L1005 option : packing limit exceeds 65536 bytes

The number following the /PACKCODE option is greater than 65536.

L1006 option : stack size exceeds 65534 bytes

The size you specified for the stack in the /STACK option of the LINK command is more than 65534 bytes.

L1007 option : interrupt number exceeds 255

You gave a number greater than 255 as a value for the /OVERLAYINTERRUPT option.

L1008 option : segment limit set too high

The specified limit on the /SEGMENTS option is greater than 3072 using the /SEGMENTS

L1009 *option* : **CPARMAXALLOC : illegal value**

The number you specified in the /CPARMAXALLOC option is not in the range 1 to 65535.

L1020 **no object modules specified**

You did not specify any object-file names to the linker.

L1021 **cannot nest response files**

A response file occurs within a response file.

L1022 **response line too long**

A line in a response file is longer than 127 characters.

L1023 **terminated by user**

You entered **Ctrl + C**.

L1024 **nested right parentheses**

You typed the contents of an overlay incorrectly on the command line.

L1025 **nested left parentheses**

You typed the contents of an overlay incorrectly on the command line.

L1026 **unmatched right parenthesis**

A right parenthesis is missing from the contents specification of an overlay on the command line.

L1027 **unmatched left parenthesis**

A left parenthesis is missing from the contents specification of an overlay on the command line.

L1030 **missing internal name**

In the module definitions file, when you specify an import by entry number, you must give an internal name, so the linker can identify references to the import.

L1031 **module description redefined**

In the module definitions file, a module description specified with the DESCRIPTION keyword is given more than once.

L1032 **module name redefined**

In the module definitions file, the module name is defined more than once with the NAME or LIBRARY keyword.

L1040 **too many exported entries**

An attempt is made to export more than 3072 names.

L1041 resident-name table overflow

The total length of all resident names, plus three bytes per name, is greater than 65534.

L1042 nonresident-name table overflow

The total length of all nonresident names, plus three bytes per name, is greater than 65534.

L1043 relocation table overflow

There are more than 65536 load-time relocations for a single segment.

L1044 imported-name table overflow

The total length of all the imported names, plus one byte per name, is greater than 65534 bytes.

L1045 too many TYPDEF records

An object module contains more than 255 TYPDEF records.

These records describe communal variables. This error can only appear with programs produced by compilers that support communal variables.

L1046 too many external symbols in one module

An object module specifies more than the limit of 1023 external symbols. Break the module into smaller parts.

L1047 too many group, segment, and class names in one module

The program contains too many group, segment, and class names. Reduce the number of groups, segments, or classes, and recreate the object files.

L1048 too many segments in one module

An object module has more than 255 segments. Split the module or combine segments.

L1049 too many segments

The program has more than the maximum number of segments.

The SEGMENTS option specifies the maximum allowed number; the default is 128. Relink using the /SEGMENTS option with an appropriate number of segments.

L1050 too many groups in one module

The linker found more than 21 group definitions (GRPDEF) in a single module.

Reduce the number of group definitions or split the module.

L1051 too many groups

The program defines more than 20 groups, not counting DGROUP. Reduce the number of groups.

L1052 too many libraries

An attempt is made to link with more than 32 libraries. Combine libraries, or use modules that require fewer libraries.

L1053 symbol table overflow

The program has more than 256K bytes of symbolic information, such as public, external, segment, group, class, and file names). Combine modules or segments and recreate the object files. Eliminate a many public symbols as possible.

L1054 requested segment limit too high

The linker does not have enough memory to allocate tables describing the number of segments requested (the default is 128 or the value specified with the /SEGMENTS option). Try linking again using the /SEGMENTS option to select a smaller number of segments (for example, use 64 if the default was used previously), or free some memory by eliminating resident programs or shells.

L1056 too many overlays

The program defines more than 63 overlays.

L1057 data record too large

A LEDATA record (in an object module) contained more than 1024 bytes of data. This is a translator (compiler or assembler) error. Note which translator (compiler or assembler) produced the incorrect object module and the circumstances, and contact your authorized IBM Personal Computer dealer.

L1070 segment size exceeds 64K

A single segment contains more than 64K bytes of code or data. Try compiling, or assembling, and linking using the large model.

L1071 segment _TEXT larger than 65520 bytes

This error is likely to occur only in small-model C programs, but it can occur when any program with a segment named _TEXT is linked using the /DOSSEG option of the LINK command. Small-model C programs must reserve code addresses 0 and 1; this is increased to 16 for alignment purposes.

L1072 common area longer than 65536 bytes

The program has more than 64K bytes of communal variables. This error cannot appear with object files produced by the IBM Macro Assembler. It occurs only with programs produced by IBM C/2 or other compilers that support communal variables.

L1073 file-segment limit exceeded

There are more than 255 physical or file segments.

L1074 *name* : group larger than 64K bytes

A group contained segments which total more than 65536 bytes.

L1075 entry table larger than 65535 bytes

Because of an excessive number of entry names, you have exceeded a linker table size limit. Reduce the number of names in the modules you are linking.

L1080 cannot open list file

The disk or the root directory is full. Delete or move files to make space.

L1081 out of space for run file

The disk on which .EXE file is being written is full. Free more space on the disk and restart the linker.

L1082 stub .EXE file not found

The stub file specified in the module definitions file is not found.

L1083 cannot open run file

The disk or the root directory is full. Delete or move files to make space.

L1084 cannot create temporary file

The disk or root directory is full. Free more space in the directory and restart the linker.

L1085 cannot open temporary file

The disk or the root directory is full. Delete or move files to make space.

L1086 scratch file missing

Internal error. You should note the conditions when the error occurs and contact your authorized IBM Personal Computer dealer.

L1087 unexpected end-of-file on scratch file

The disk with the temporary linker-output file is removed.

L1088 out of space for list file

The disk on which the listing file is being written is full. Free more space on the disk and restart the linker.

L1089 *filename* : cannot open response file

The linker could not find the specified response file. This usually indicates a typing error.

L1090 cannot reopen list file

The original disk is not replaced at the prompt. Restart the linker.

L1091 unexpected end-of-file on library

The disk containing the library probably was removed. Replace the disk containing the library and run the linker again.

L1092 cannot open module definitions file

The specified module definitions file cannot be opened.

L1100 stub .EXE file invalid

The stub file specified in the definitions file is not a valid .EXE file.

L1101 invalid object module

One of the object modules is non-valid.

If the error persists after recompiling, contact your authorized IBM Personal Computer dealer.

L1102 unexpected end-of-file

A non-valid format for a library was found.

L1103 attempt to access data outside segment bounds

A data record in an object module specified data extending beyond the end of a segment. This is a translator error. Note which translator (compiler or assembler) produced the incorrect object module and the circumstances, and contact your authorized IBM Personal Computer dealer.

L1104 *filename* : not valid library

The specified file is not a valid library file. This error causes the linker to stop running.

L1110 DOSALLOCCHUGE failed

Internal error. You should note the conditions when the error occurs and contact your authorized IBM Personal Computer dealer.

L1111 DOSREALLOCHUGE failed

Internal error. You should note the conditions when the error occurs and contact your authorized IBM Personal Computer dealer.

L1112 DOSGETHUGESHIFT failed

Internal error. You should note the conditions when the error occurs and contact your authorized IBM Personal Computer dealer.

L1113 unresolved COMDEF; internal error

You should note the conditions when the error occurs and contact your authorized IBM Personal Computer dealer.

L1114 file not suitable for /EXEPACK; relink without

For the linked program, the size of the packed load image plus the packing overhead is larger than that of the unpacked load image. Relink without the EXEPACK option.

L2000 imported entry point

A MODEND, or starting address record, referred to an imported name. Imported program-starting addresses are not supported.

L2001 fixup(s) without data

A FIXUP record occurred without a data record immediately preceding it.

This is probably a compiler error. See the *IBM Disk Operating System Reference* for more information on FIXUP.

L2002 fixup overflow near *number in frame seg segname target seg segname target offset number*

The following conditions can cause this error:

- A group is larger than 64K bytes
- The program contains an intersegment short jump or intersegment short call
- The name of a data item in the program conflicts with that of a subroutine in a library included in the link
- An EXTRN declaration in an assembler-language source file appeared inside the body of a segment.

For example:

```
code    SEGMENT public 'CODE'
        EXTRN   main:far
start  PROC    far
        call   main
        ret
start  ENDP
code   ENDS
```

The following construction is preferred:

```

        EXTRN    main:far
code    SEGMENT public 'CODE'
start  PROC     far
        call    main
        ret
start  ENDP
code   ENDS

```

Revise the source file and recreate the object file.

L2003 intersegment self-relative fixup

An intersegment self-relative fixup is not allowed.

L2004 LOBYTE-type fixup overflow

A LOBYTE fixup produced an address overflow.

L2005 fixup type unsupported

A fixup type occurred that is not supported by the linker. This is probably a compiler error. You should note the conditions when the error occurs and contact your authorized IBM Personal Computer dealer.

L2010 too many fixups in LIDATA record

There are more fixups applying to a LIDATA record than will fit in the linker's 1024-byte buffer.

The buffer is divided between the data in the LIDATA record itself and run-time relocation items, which are 8 bytes apiece, so the maximum varies from 0 to 128. This is probably a compiler error.

L2011 'name' : NEAR/HUGE conflict

Conflicting NEAR and HUGE attributes are given for a communal variable.

This error can occur only with programs produced by compilers that support communal variables.

L2012 'name' : array-element size mismatch

A far communal array is declared with two or more different array-element sizes (for example, an array declared once as an array of characters and once as an array of real numbers). This error cannot occur with object files produced by the IBM Macro Assembler/2. It occurs only with IBM C/2 and any other compiler that supports far communal arrays.

L2013 LIDATA record too large

A LIDATA record in an object module contains more than 512 bytes of data. Most likely, an assembly module contains a very complex structure definition or a series of deeply-nested DUP

operators. For example, the following structure definition causes this error:

```
alpha DB 10DUP(11 DUP(12 DUP(13 DUP(...))))
```

Simplify the structure definition and reassemble. (LIDATA is a DOS term).

L2020 no automatic data segment

No group named DGROUP is declared.

L2021 library instance data not supported in real mode

The library module is directed to have instance data. This works in OS/2 mode only.

L2022 name alias internalname: export undefined

A name is directed to be exported but is not defined anywhere.

L2023 name alias internalname: export imported

An imported name is directed to be exported.

L2024 name : symbol already defined

One of the special overlay symbols required for overlay support is defined by an object.

L2025 'name' : symbol defined more than once

Remove the extra symbol definition from the object file.

L2026 multiple definitions for entry ordinal number

More than one entry point name is assigned to the same ordinal.

L2027 name : ordinal too large for export

You tried to export more than 3072 names.

L2028 automatic data segment plus heap exceeds 64K

The size of DGROUP near data plus requested heap size is greater than 64K.

L2029 unresolved externals

One or more symbols are declared to be external in one or more modules, but they are not publicly defined in any of the modules or libraries.

A list of the unresolved external references appears after the message, as shown in the following example:

```
_exit in file(s)
main.obj (main.c)
_fopen in files(s)
fileio.obj(fileio.c) main.obj(main.c)
```

The name that comes before **in file(s)** is the unresolved external symbol. On the next line is a list of object modules which have made references to this symbol. This message and the list are also written to the map file, if one exists.

L2030 starting address not code (using class 'CODE')

You specified a starting address to the linker which is a segment that is not a CODE segment. Reclassify the segment to CODE, or correct the starting point.

L4001 frame-relative fixup, frame ignored

A fixup occurred with a frame segment different from the target segment where either the frame or the target segment is not absolute. Such a fixup is meaningless in OS/2 mode, so the target segment is assumed for the frame segment.

L4002 frame-relative absolute fixup

A fixup occurred with a frame segment different from the target segment where both frame and target segments were absolute. This fixup is processed using base-offset arithmetic, but the warning is issued because the fixup may not be valid in OS/2 mode.

L4010 invalid alignment specification

The number following the **/ALIGNMENT** option is not a power of 2, or is not in numerical form.

L4011 PACKCODE value exceeding 65500 unreliable

Code segments of length 65501-65536 may be unreliable on the 80286 processor.

L4012 load-high disables EXEPACK

The options **/HIGH** and **/EXEPACK** are mutually exclusive.

L4013 invalid option for new-format executable file ignored

If an OS/2 mode program is being produced, then the options **/CPARMAXALLOC**, **/DSALLOCATE**, **/EXEPACK**, **/NOGROUPASSOCIATION**, and **/OVERLAYINTERRUPT** are meaningless, and the linker ignores them.

L4014 invalid option for old-format executable file ignored

If a DOS format program is produced, the options **/ALIGNMENT**, **/NOFARCALLTRANSLATION**, and **/PACKCODE** are meaningless, and the linker ignores them.

L4020 name : code-segment size exceeds 65500

Code segments of length 65501-65536 may be unreliable on the 80286 processor.

L4021 no stack segment

The program does not contain a stack segment defined with STACK combine type. This message should not appear for modules compiled with the IBM C/2, but it could appear for an assembler-language module. Normally, every program should have a stack segment with the combine type specified as STACK. You can ignore this message if you have a specific reason for not defining a stack or for defining one without the STACK combine type.

L4022 *name1, name2* : groups overlap

Two groups are defined such that one starts in the middle of another. This may occur if you defined segments in a module definitions file or assembly file and did not correctly order the segments by class.

L4023 *exportname* : export internal-name conflict

An exported name, or its associated internal name, conflict with an already-defined public symbol.

L4024 *name* : multiple definitions for export name

The name *name* is exported more than once with different internal names. All internal names except the first are ignored.

L4025 *name* : import internal-name conflict

An imported name, or its associated internal name, is also defined as an exported name. The import name is ignored.

The conflict may come from a definition in either the module definition file or an object file.

L4026 *modulename* : self-imported

The module definitions file directed that a name be imported from the module being produced.

L4027 *name* : multiple definitions for import internal-name

An imported name, or its associated internal name, is imported more than once. The imported name is ignored after the first mention.

L4028 *name* : segment already defined

A segment is defined more than once with the same name in the module definitions file. Segments must have unique names for the linker. All definitions with the same name after the first are ignored.

L4029 *name* : **DGROUP segment converted to type data**

A segment which is a member of DGROUP is defined as type CODE in a module definition file or object file.

This probably happened because a CLASS keyword in a SEGMENTS statement is not given.

L4030 *name* : **segment attributes changed to conform with automatic data segment**

The segment named *name* is defined in DGROUP, but the *shared* attribute is in conflict with the *instance* attribute. For example, the *shared* attribute is NONSHARED and the *instance* is SINGLE, or the *shared* attribute is SHARED and the *instance* attribute is MULTIPLE. The bad segment is forced to have the right *shared* attribute and the link continues. The image is not marked as having errors.

L4031 *name* : **segment declared in more than one group**

A segment is declared to be a member of two different groups. Correct the source file and recreate the object files.

L4032 *name* : **code-group size exceeds 65500 bytes**

Code segments of length 65501-65536 may be unreliable on the 80286 processor.

L4034 **more than 239 overlay segments; extra put in root**

You specified an overlay structure containing more than 239 segments. The extra segments have been assigned to the root overlay.

L4036 **no automatic data segment**

L4040 **NON-CONFORMING : obsolete**

In the module definitions file, NON-CONFORMING is a valid keyword for earlier versions of LINK and is now obsolete.

L4041 **HUGE segments not yet supported**

This feature is not yet implemented in the linker.

L4042 **cannot open old version**

An old version of the EXE file, specified with the OLD keyword in the module definitions file, could not be opened.

L4043 **old version not segmented-executable format**

The old version of the .EXE file, specified with the OLD keyword in the module definitions file, does not conform to segmented-executable format.

L4050 too many public symbols

The **/MAP** option is used to request a sorted listing of public symbols in the map file, but there were too many symbols to sort (the default is 2048 symbols). The linker produces an unsorted listing of the public symbols. Relink using **/MAP:number**.

L4051 filename : cannot find library

The linker could not find the specified file. Enter a new file name, a new path specification, or both.

L4053 VM.TMP : illegal file name; ignored

VM.TMP appears as an object-file name. Rename the file and rerun the linker.

L4054 filename : cannot find file

The linker could not find the specified file. Enter a new file name, a new path specification, or both.

Linker Limits

The table below summarizes the limits imposed by the linker. If you find one of these limits, you may adjust your program so that the linker can accommodate it.

Item	Limit
Symbol table	256K
Load-time relocations (for DOS programs)	Default is 32K. If /EXEPACK is used, the maximum is 512K.

Item	Limit
Public symbols	The range 7700-8700 can be used as a guideline for the maximum number of public symbols allowed; the actual maximum depends on the program.
External symbols per module	1023
Groups	Maximum number is 21, but the linker always defined DGROUP so the effective maximum is 20.
Overlays	63
Segments	128 by default; however, this maximum can be set as high as 3072 by using the /SEGMENTS option of the LINK command.
Libraries	32
Group definitions per module	21
Segments per module	255
Stack	64K

Appendix F. Library Manager Error Messages

Error messages produced by the IBM Library Manager, LIB, have one of the following formats:

- *filename*|LIB : fatal error U1xxx : *messagetext*
- *filename*|LIB : warning U4xxx : *messagetext*

The message begins with the input file name (*filename*), if one exists, or with the name of the utility. LIB may display the following error messages:

U1150 page size too small

The page size of an input library is too small, which indicates a non-valid input .LIB file.

U1151 syntax error : illegal file specification

You gave a command operator, such as a minus sign (-), without a module name following it.

U1152 syntax error : option name missing

You gave a forward slash (/) with a value following it.

U1153 syntax error : option value missing

You gave the /PAGESIZE option without a value following it.

U1154 option unknown

An unknown option is given. Currently, LIB recognizes the /PAGESIZE option only.

U1155 syntax error : illegal input

The given command did not follow correct LIB syntax.

U1156 syntax error

The given command did not follow correct LIB syntax.

U1157 comma or new line missing

A comma or carriage return is expected in the command line, but did not appear. This may indicate an inappropriately placed comma, as in the following line:

LIB math.lib,-mod1 + mod2;

The line should have been entered as follows:

LIB math.lib -mod1 + mod2;

U1158 terminator missing

Either the response to the **Output library:** prompt or the last line of the response file used to start LIB did not end with a carriage return.

U1161 cannot rename old library

LIB could not rename the old library to have a .BAK extension because the .BAK version already existed with read-only protection. Change the protection of the old .BAK version.

U1162 cannot reopen library

The old library could not be reopened after it was renamed to have a .BAK extension.

U1163 error writing to cross-reference file

The disk or root directory is full. Delete or move files to make space.

U1170 too many symbols

More than 4609 symbols appeared in the library file.

U1171 insufficient memory

LIB did not have enough memory to run. Remove any shells or resident programs and try again, or add more memory.

U1172 no more virtual memory

You should note the conditions when the error occurs and contact your authorized IBM Personal Computer dealer.

U1173 internal failure

You should note the conditions when the error occurs and contact your authorized IBM Personal Computer dealer.

U1174 mark : not allocated

You should note the conditions when the error occurs and contact your authorized IBM Personal Computer dealer.

U1175 free : not allocated

You should note the conditions when the error occurs and contact your authorized IBM Personal Computer dealer.

U1180 write to extract file failed

The disk or root directory is full. Delete or move files to make space.

U1181 write to library file failed

The disk or root directory is full. Delete or move files to make space.

U1182 *filename* : cannot create extract file

The disk or root directory is full, or the specified extract file already exists with read-only protection.

Make space on the disk or change the protection of the extract file.

U1183 cannot open response file

The response file was not found.

U1184 unexpected end-of-file on command input

An end-of-file character is received prematurely in response to a prompt.

U1185 cannot create new library

The disk or root directory is full, to the library file already exists with read-only protection.

Make space on the disk or change the protection of the library file.

U1186 error writing to new library

The disk or root directory is full. Delete or move files to make space.

U1187 cannot open VM.TMP

The disk or root directory is full. Delete or move files to make space.

U1188 cannot write to VM

You should note the conditions when the error occurs and contact your authorized IBM Personal Computer dealer.

U1189 cannot read from VM

You should note the conditions when the error occurs and contact your authorized IBM Personal Computer dealer.

U1190 DOSALLOCHUGE failed

You should note the conditions when the error occurs and contact your authorized IBM Personal Computer dealer.

U1191 DOSREALLOCHUGE failed

You should note the conditions when the error occurs and contact your authorized IBM Personal Computer dealer.

U1192 DOSGETHUGESHIFT failed

You should note the conditions when the error occurs and contact your authorized IBM Personal Computer dealer.

U1200 *name* : invalid library header

The input library file has a non-valid format. It is either not a library file, or it has been corrupted.

U1203 *name* : invalid object module near *location*

The module specified by *name* is not a valid object module.

U4150 *modulename* : module redefinition ignored

A module is specified to be added to a library, but a module with the same name is already in the library. Or, a module with the same name is found more than once in the library.

U4151 *symbol(modulename)* : symbol redefinition ignored

The specified symbol is defined in more than one module.

U4152 *filename* : cannot create listing

The directory or disk is full, or the cross-reference listing file already exists with read-only protection. Make space on the disk or change the protection of the cross-reference listing file.

U4153 *number* : page size too small; ignored

The value specified in the **/PAGESIZE** option is less than 16.

U4155 *modulename* : module not in library; ignored

The specified module is not found in the input library.

U4156 *libraryname* : output-library specification ignored

An output library is specified in addition to a new library name. For example, specifying

LIB new.lib + one.obj,new.lst,new.lib

where **new.lib** does not already exist causes this error.

U4157 *filename* : cannot access file

LIB is unable to open the specified file.

U4158 *libraryname* : invalid library header; file ignored

The input library has an incorrect format.

U4159 *filename* : **invalid format***hexnumber*; **file ignored**

The signature byte or word, *hexnumber*, of an input file is not one of the recognized types.

Index

Special Characters

.COM extension D-4
.EXE extension D-4
\$DYNAMIC 7
\$INCLUDE 9
 restrictions on use 9
\$LINESIZE 12
\$LIST 13
\$MODULE 14
\$OCODE 15
\$PAGE 16
\$PAGEIF 17
\$PAGESIZE 18
\$SKIP 19
\$STATIC 20
\$SUBTITLE 22
\$TITLE 23
/E 148
/S CodeView option D-7
/X 148
?Redo from start 194
334

A

A parameter 15
ABS 29
absolute value 29
active page 385
addresses D-1
 as arguments D-1
addresses as arguments D-4
aliasing of variables 39
alter system memory 387

ampersand symbol 334
animation 348
append 286
arctangent 31
arguments D-2, D-6
 dialog commands D-3, D-6
arrays 120, 142, 300
ASC 30
ASCII character codes B-1
ASCII code 382
ASCII codes 30, 66
 converting to 30
aspect ratio 71, 133
assemble command D-6
assembler language
 subroutines 37
assembly mode D-4
assignment statement 223
ATN 31

B

background 80, 303
BASIC program editor 10, 330
 question mark for PRINT 330
BEEP 32
blinking characters 81
BLOAD 33
border screen 80
boundary 303, 453
branching 180, 267
breakpoint clear command D-1,
 D-2
breakpoint disable
 command D-1, D-2

breakpoint enable
 command D-1, D-2
breakpoint set D-3
breakpoint set command D-3,
 D-6
BSAVE 35
burst, screen 384
BYVAL 106

C

C calling convention 48
C expressions D-2, D-3
CALL 37
CALL ABSOLUTE 52
CALL INT86 54
CALL INT86X 57
calling BASIC subprograms 38
calling C subprograms 48
calling Macro Assembler subpro-
grams 43
calling Pascal subprograms 46
CALLS 50
CASE 59
case sensitivity D-6
CDBL 61
CHAIN 62, 90
change current directory 64
CHDIR 64
child process 140, 298, 393, 396
CHR\$ 66
CINT 68
CIRCLE 69
CLEAR 73
clear screen 78
clear system buffer 364
CLNG 75
clock 403
CLOSE 76

close disk files 364
CLOSE Statement 364
CLS 78
CodeView error messages D-1
color 343, 453
COLOR statement 80, 304, 382
COLOR statement in graphics
 mode 84
COM 86
comma in formatting string 335
COMMAND\$ 88
commands D-3
 assemble D-6
 breakpoint clear D-1, D-2
 breakpoint disable D-1, D-2
 breakpoint enable D-1, D-2
 breakpoint set D-6
 radix D-1
 register D-1
 restart D-4, D-5
 search D-3, D-5
 shell escape D-4
 tracepoint D-1, D-5
 view D-4
 watch D-1
 watchpoint D-1
comments 362
COMMON 62, 90
communication errors A-34
communications 292
 communications buffer 293
communications trapping 86,
 262
compile—time errors A-1
compiler commands functions
 and statements 24
compiler's data segment 116
computed GOSUB/GOTO 267

- CONFIG.SYS file D-6
- constant numbers as
 - arguments D-2
- converting degrees to
 - radians 95
- converting from numbers for
 - random files 255
- converting from numeric to
 - octal 261
- converting IEEE numbers to
 - Microsoft Binary Format
 - format 257
- converting Microsoft Binary
 - Format numbers to IEEE
 - format 100
- converting numbers 61, 68, 75,
96
- converting numbers from
 - random files 98
- converting numeric to
 - string 416
- converting radians to
 - degrees 31
- converting string to
 - numeric 444
- converting strings to lower
 - case 219
- converting strings to upper
 - case 441
- converting to integer 68
- coordinates
 - physical 464
 - world 464
- coordinates, absolute or relative
 - form 303, 343
- COS 95
- cosine 95
- create a directory 253

- creating tree structure 253
- CSNG 96
- CSRLIN 97
- cursor position 97, 234, 329
- CVI, CVL, CVS, CVD 98
- CVSMBF, CVDMBF 100

D

- DATA 102, 356
- data segment 116
- DATE\$ 104
- decisions 59, 183
- DECLARE 106
- declaring arrays 120
- declaring variable types 118
- DEF FN 110
- DEF SEG 116
- defining variable types 435
- DEFtype statements 118
- deleting a file 215
- deleting arrays 142
- Device timeout 245
- DIM statement 120
- dimensioning arrays 120
- DIR 156
- direct mode 265
- directory 64
- Disk Operating System Refer-
 - ence 271
- display pages 385
- divide by zero D-2
- division by zero A-25
- DO 126
- documentation, internal
 - program 362
- DOS command 298, 393, 396
- DOS national diskettes 271
- DOS signals 400

double asterisk 337
double asterisk, dollar sign 335
double dollar sign 335
double-precision 61
DRAW statement 130
DS (compiler's DATA
segment) 116
duration, time 403
DYNAMIC 7

E

elapsed time 430
ELSE 183
ELSEIF 183
END 135
END DEF 110
END IF 183
end of file 141
END SUB 421
ending BASIC 425
ENDTYPE 435
ENVIRON 136
ENVIRON\$ 138
environment 136, 138
EOF 141
ERASE 142
ERASE (DOS) 215
erasing a file 215
erasing arrays 142
erasing variables 73
ERDEV 144
ERDEV\$ 144
ERL 146
ERR 146
ERROR 148
error codes 146, 148, Appendix
A
error line 146
error messages
CodeView D-1
Library Manager F-1
Linker Error Messages and
Limits E-1
error trapping 146, 148, 265,
366, A-24
errors messages A-1
Errors while compiling a
program A-2
errors, compile-time A-1
errors, I/O A-35
errors, run-time A-1, A-2
event trapping 212
KEY(n) 212
exchanging variables 424
exclamation point (!)
shell escape command D-4
exclamation point symbol 333
executable file D-4
command line D-7
EXEPACK link option D-7
EXIT 126
exit BASIC 425
EXIT DEF 110
EXIT SUB 421
EXP 150
exponential function 150
expression evaluation D-2, D-3
expressions, regular D-3, D-5
Extended Codes C-6

F

false or true 382
FIELD 151
File access control 237, 442
file handles D-6
file menu D-4
 load D-4, D-6
 shell D-4
file number 166
file size 239
file, position of 232
FILEATTR 154
FILES 156
finding D-3, D-5
 text strings D-3, D-5
FIX 159
fixed-length strings 246
flag bits D-1
flag mnemonics D-1
floor function 201
FOR 160
foreground 80
formatting 333
 numeric fields 334
 string fields 333
FRE 164
free space 73, 164
FREEFILE 166
frequency 403
frequency table 404
FUNCTION 168
function keys 206
function, declaring 106

G

GET (files) 173
GET (graphics) 175
glissando 406
GOSUB 178, 267
GOTO 180, 267
GRAFTABL Command 385
GRAPHICS
 COLOR 84
 VIEW 452
 WINDOW 464
graphics statements 130, 224
 DRAW 130
 LINE 224

H

HEX\$ 182
hexadecimal 182
high-intensity characters 81
how to use this book 1

I

I/O control 203
I/O errors A-34
IBM Enhanced Keyboard 207,
 208, 212, 269, C-1
identifiers in arguments D-6
IF 183
 block format 183
Illegal function call
 in KEY 207
imbedding files 9
INCLUDE 9
indent 426
index (position in string) 200
INKEY\$ 190
INP 192

INPUT 193
INPUT # 196
input editor 193, 228
input file mode 286
INPUT\$ 198
INSTR 200
INT 201
integer
internal debugger error D-2,
D-3
INT86 54
INT86X 57
IOCTL 203
IOCTL\$ 205

J

joystick 412
joystick button 281, 417, 419
jumping 180, 267

K

key trapping 206
KEY(n) 212
KILL 215

L

labels 178, 180
LBOUND 216
LCASE\$ 219
LEFT\$ 220
left-justify 246
LEN 221
length of file 239
length of string 221
LET 223
library manager error
messages F-1

light pen 273, 313
LINE 224
line drawing in graphics 224
line feed 289
LINE INPUT 228
LINE INPUT # 230
line styling 225
LINESIZE 12
linker error messages and
limits E-1
LIST 13
listing files 156
on disk 156
loading binary data 33
LOC 232
local variables D-6
LOCATE 234
LOCK 237
LOF 239
LOG 241
logarithm 241
LOOP 126
loops 160, 459
LPOS 243
LPRINT 244
LPRINT Statement 331
LPRINT USING 244
LPT1: 243, 244
LSET 246
LTRIM\$ 248

M

machine input port status 457
machine language
subprograms 52
machine language
subroutines 37, 50
member selection
operators D-5

- memory image 35
- menu
 - file D-4, D-6
 - load D-4, D-6
 - shell D-4
 - options D-6
 - case sensitivity D-6
 - run D-5
 - restart D-5
 - start D-5
- MERGE 62
- metacommands 6
- minus sign 335
- MKDIR 253
- MKI\$, MKL\$, MKS\$, MKD\$ 255
- MKSMBF\$, MKDMBF\$ 257
- mode, screen 384
- MODULE 14
- multi-line functions 114
- music 316, 404

N

- NAME 259
- national keyboard 271
- NEXT 160
- non-U.S. keyboard 271
- notes, sound 404
- number of notes in buffer 320
- numbers as arguments D-2
- numeric fields 334

O

- OCODE 15
- OCT\$ 261
- octal 261
- offset 116, 446, 448
- ON COM(n) 262

- ON ERROR 149, 265
- ON KEY(n) 269
- ON PEN 273
- ON PLAY(n) 275
- ON SIGNAL(n) 278
- ON STRIG(n) (joystick button) 281
- ON TIMER 284
- ON...GOSUB 267
- ON...GOTO 267
- OPEN 286
- OPEN "COM..." 292
- OPEN "PIPE..." 298
- opening files 286
- opening paths 286
- operand types D-5
 - incompatible operations D-5
- OPTION BASE 300
- options
 - CodeView D-7
 - /S D-7
 - linker D-7
 - EXEPACK D-7
- options menu
 - case sensitivity D-6
- OS/2 mode 34, 313, 328
- OUT 302
- output file mode 286
- overflow A-24
- overlay 62

P

- PAGE 16
- page, active 385
- page, visual 385
- PAGEIF 17
- PAGESIZE 18
- PAINT 303

- paint tiling 308
- palette 84
- panning 467
- Pascal calling convention 46
- passing variables 62
- paths 64
- paths, opening 286
- patterns 308
- PEEK 311
- PEN 313
- PEN OFF Statement 314
- PEN ON Statement 314
- period operator (.) D-5
- physical coordinates 464
- PLAY 316
- PLAY(n) 320
- plus sign 335
- PMAP 321
- POINT 324
- POKE 327
- POS 329
- position in string 200
- position of file 232
- positioning the cursor 234
- precision 118
- prefixes
 - printf type D-1
 - with type specifiers D-1
- PRESET 343
- PRINT 330
- PRINT # 340
- PRINT # USING 340
- print formatting 333
- PRINT USING 333
- print zones 330
- printf type prefixes D-1
- printf type specifiers D-1
- printing 244

- program editor 10
- program stop 414
- programming function keys 206
- protect mode 311, 316, 320
- protected mode 116, 273, 275,
281, 403, 412, 417, 419
- PSET 343
- punctuation, PRINT
Statement 330
- PUT (files) 345
- PUT (graphics) 347

R

- radix command D-1
- random files 151, 173, 286
- random numbers 353, 372
- RANDOMIZE 353
- READ 102, 356
- redefining function keys 206
- REDIM 358
- Redo 194
- register command D-1
- register variables D-5
- regular expressions D-3, D-5
- REM 6, 362
- remarks 362
- removing a directory 370
- removing spaces from
strings 248, 375
- RENAME 259
- renaming files 259
- repeating a string 420
- RESET 364
- restart command D-4, D-5
- RESTORE 356, 365
- RESUME 366
- RETURN 368
- right-justify 246

RMDIR 370
RND 372
rounding to an integer 68
RSET 246
RTRIM\$ 375
RUN 377
run menu D-5
 restart D-5
 start D-5
run-time errors A-1, A-23
running a program 377

S

SADD function 380
saving binary data 35
scan codes C-1
screen buffer address 36
SCREEN function 382
SCREEN statement 384
search command D-3, D-5
seeding random number generator 353
segment 450
segment of memory 116
SELECT CASE 59
sequential files 286
SETMEM 387
setting
 function keys 206
SGN 389
SHARED 390
SHARED attribute 91, 121, 359
shell escape command D-4
SHELL function 393
SHELL statement 396
sign of a number 389
SIGNAL 400
SIN 402
sine 402
single-precision 96
SKIP 19
slash (/), Search
 Command D-3, D-5
soft keys (see function keys)
SOUND 403
sounds 32, 316, 403
source mode D-4
space 464
SPACE\$ 407
spaces 331
SPC 408
SQR 409
square root 409
stack space 73
start-up D-2, D-4
 command line D-2, D-4
start-up code D-4
STATIC 20, 410
STEP 160
STICK 412
STOP 414
storage release D-4
STR\$ 416
STRIG 417
STRIG(n) 419
string fields 333
string space 73, 164
STRING\$ 420
strings as arguments D-3
SUB 421
subprogram, declaring 106
subroutines 178, 267
subscripts 120, 300
substring 220, 250, 369
SUBTITLE 22
superimpose image 348

SWAP 424
symbols in arguments D-6
Syntax error
 in KEY(n) 214
SYSTEM 425
system memory, alter 387

T

TAB 426
TAN 427
tangent 427
tempo table 405
terminating BASIC 425
text files, identifying D-4
THEN 183
tile painting 308
tiling 304
TIME\$ 428
time, duration 403
TIMER 430
TITLE 23
trace 433
 of keys 206, 212
trapping, communications 86
tree-structured directories
 changing 64
triggers, joystick 417
trigonometric functions
 arctangent 31
trigonometric sine 402
trigonometric tangent 427
TROFF 433
TRON 433
true or false 382
truncation 159, 201

TYPE 435
type casting D-1
type specifiers D-1

U

UBOUND 438
UCASE\$ 441
underflow A-24
underscore 336
UNLOCK 442
UNTIL 126
user workspace 73, 164
user-defined functions 110

V

VAL 444
VARPTR 446
VARPTR\$ 448
VARSEG 450
video modes D-7
VIEW 452
view command D-4
VIEW PRINT 456
visual page 385
vpage 385

W

WAIT 457
watch command D-1
watchpoint command D-1
watchpoint, defining D-1
WEND 459
WHILE 126, 459
WIDTH 461
WINDOW 452, 464
workspace 73, 164

world coordinates 464
WRITE 470
WRITE # 472

Z

zero, division by D-2
zones, print 330
zooming 467

© IBM Corp. 1987
All rights reserved.

International Business
Machines Corporation
P.O. Box 1328-W
Boca Raton,
Florida 33429-1328

Printed in the
United States of America

00F8662

