

GRiDBASIC REFERENCE MANUAL

March 9, 1983

Model Number 21020-40

COPYRIGHT © 1983 GRiD Systems Corporation
2535 Garcia Avenue
Mountain View, CA 94043
(415) 961-4800

Manual Name : GRiDBASIC Reference Manual
Model Number 21020-40
Issue date: March 9, 1983

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopy, recording, or otherwise, without the prior written permission of GRiD Systems Corporation.

The information in this document is subject to change without notice.

NEITHER GRiD SYSTEMS CORPORATION NOR THIS DOCUMENT MAKES ANY EXPRESSED OR IMPLIED WARRANTY, INCLUDING, BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY, QUALITY, OR FITNESS FOR A PARTICULAR PURPOSE. GRiD Systems Corporation makes no representation as to the accuracy or adequacy of this document. GRiD Systems Corporation has no obligation to update or keep current the information contained in this document.

GRiD System Corporation's software products are copyrighted by and shall remain the property of GRiD Systems Corporation.

UNDER NO CIRCUMSTANCES WILL GRID SYSTEMS CORPORATION BE LIABLE FOR ANY LOSS OR OTHER DAMAGES ARISING OUT OF THE USE OF THIS MANUAL.

The following are trademarks of GRiD Systems Corporation: GRiD, NAVIGATOR, COMPASS CENTRAL, COMPASS COMPUTER, and LEVERAGED LEARNING.

TABLE OF CONTENTS

GRiDBASIC COMMAND SUMMARY	x
-------------------------------------	---

ABOUT THIS BOOK	xi
---------------------------	----

CHAPTER ONE: GETTING STARTED

What Is GRiDBASIC	1-1
Invoking GRiDBASIC	1-1
The Programming (Indirect) Mode	1-2
Running, Continuing, and Stopping a Program.	1-3
The ESCape Key	1-3
Erasing Line(s).	1-3
Other Commands	1-3
The Direct Mode	1-4
About the GRiDBASIC Environment	1-4
Layout of the GRiDBASIC Screen	1-4
Lines and Line Numbering	1-6
Automatic Line Numbering	1-6
Manual Line Numbering	1-6
Renumbering	1-6
Multiple Statements	1-7
Reformatting your Listings	1-7

CHAPTER TWO: GENERAL INFORMATION ABOUT GRIDBASIC

Syntax Diagrams	2-1
Reserved Words	2-3
Constants	2-4
String Constants	2-4
Numeric Constants	2-5
Variables	2-6
Array Variables	2-7
Expressions and Operators	2-7
Order of Precedence and Numeric Operators.	2-7
Relational Operators	2-8
Logical Operators.	2-8
String Operators.	2-11
File Naming Conventions.	2-12
File Kinds.	2-13
Delimiters.	2-13

CHAPTER THREE: ASSIGNMENT AND DEFINITION STATEMENTS

DIM	3-2
LET	3-4
READ DATA [RESTORE]	3-5
REM	3-9

CHAPTER FOUR: STATEMENTS THAT CONTROL PROGRAM FLOW

END	4-2
FOR TO NEXT [STEP]	4-3
GOSUB RETURN	4-6
GOTO	4-8
IF THEN [ELSE]	4-9
ON GOTO and ON GOSUB	4-11
STOP	4-13
WHILE WEND	4-14

CHAPTER FIVE: ARITHMETIC AND LOGIC

INTEGER FUNCTIONS	5-2
ABS (Absolute)	5-3
ACOS (Arc cosine)	5-4

AND	5-5
ASIN (Arc sine)	5-6
ATN (Arctangent)	5-7
CDBL (Convert to double)	5-8
CINT (Convert to integer)	5-9
COS (Cosine)	5-10
CSNG (Convert to single)	5-11
EXP (Exponential)	5-12
FALSE	5-13
FIX	5-14
INT (Integer)	5-15
Integer Division (\)	5-16
LOG (Logarithm)	5-17
LOG10 (Log to base 10)	5-18
MOD	5-19
NOT	5-20
OR	5-21
PI	5-22
RANDOMIZE	5-23
RND (Random)	5-24
ROUND	5-28
SGN (Sign)	5-29
SIN (Sine)	5-30
SQR (Square root)	5-31
TAN (Tangent)	5-32
TRUE	5-33
TRUNC (Truncate)	5-34
XOR (Exclusive OR)	5-35

CHAPTER SIX: STRING FUNCTIONS

ASC (ASCII)	6-2
CHR\$ (Character string)	6-3
INSTR (In string)	6-4
LEFT\$ (Left string)	6-6
LEN (Length)	6-7
MID\$ (Mid string)	6-8
RIGHT\$ (Right string)	6-9
SPACE\$ (Space string)	6-10
STR\$ (S-T-R string)	6-11
STRING\$ (String)	6-12
VAL (Value)	6-13

CHAPTER SEVEN: INPUT/OUTPUT STATEMENTS

COMMA	7-2
DATE\$	7-4
INKEY\$	7-5
INPUT	7-6
LOCATE	7-9
PRINT	7-10
PRINT USING	7-12
SEMICOLON	7-19
TAB	7-20
TIME\$	7-21

CHAPTER EIGHT: SEQUENTIAL FILES STATEMENTS

CLOSE	8-3
EOF (End of file)	8-4
EOLN (End of line)	8-5
GETFILE\$	8-7
INPUT#	8-8
INPUT\$	8-10
KILL	8-11
LOC (Locating)	8-12
LOF (Length of file)	8-13
OPEN	8-14
PRINT#	8-17
PRINT# USING	8-19

CHAPTER NINE: RANDOM FILE STATEMENTS

CVI,CVS,CVD (Convert)	9-3
FIELD	9-5
GET	9-6
LOC (Locating)	9-7
LOF (Length of file)	9-8
LSET and RSET	9-9
MKI\$,MKS\$,MKD\$ (Make string)	9-10
OPEN	9-11
PUT	9-13

CHAPTER TEN: GRAPHICS STATEMENTS

CLEARMSG	10-4
DOMENU	10-5
DRAWBOX	10-6
DRAWCHARS	10-7
DRAWCIRCLE	10-9
DRAWDOT	10-10
DRAWLINE	10-11
ERASEBOX	10-12
ERASECIRCLE	10-14
ERASEDOT	10-15
ERASELINE	10-16
INVERTBOX	10-17
INVERTCIRCLE	10-18
INVERTDOT	10-19
INVERTLINE	10-20
MOVEBOX	10-21
STACKMSG	10-22

APPENDICES

APPENDIX A: ERROR MESSAGES

APPENDIX B: ASCII CHARACTERS

INDEX

LIST OF FIGURES

Figure 1-1.	The Initial Program Editing Screen	1-2
Figure 1-2.	The Program Editor Screen	1-5
Figure 1-3.	Line 1300 after Inserting SHIFT-RETURNS	1-8
Figure 2-1.	Example of a Type Mismatch Failure	2-9
Figure 2-2.	Example of Faulty Logic	2-10
Figure 2-3.	A Working Logic	2-10
Figure 2-4.	Another Working Logic	2-11
Figure 3-1.	Results of a Simple READ DATA Program	3-7
Figure 3-2.	Results of a READ with Two Variables	3-7
Figure 5-1.	Three Types of Random Numbers	5-24
Figure 5-2.	A Program and Series of Random Numbers	5-25
Figure 5-3.	Output of RND on Three Numeric Ranges	5-27
Figure 7-1.	Examples of Comma Formatting	7-3
Figure 7-2.	The Input Statement Illustrated	7-8
Figure 7-3.	How Format Characters Pad Digits	7-14
Figure 7-4.	Basic formatting PRINT USING	7-15
Figure 7-5.	The PRINT USING Format with Signs	7-16
Figure 7-6.	Asterisk and Dollar Formatting	7-17
Figure 8-1.	Worksheet Figures for Example Program	8-6
Figure 8-2.	PRINT# USING Formatting of Random Numbers	8-20
Figure 10-1.	The Three Circle Graphics	10-2
Figure 10-2.	The Three Dot Graphics	10-2
Figure 10-3.	The Three Line Graphics	10-3
Figure 10-4.	Before CLEARMSG	10-4
Figure 10-5.	After CLEARMSG	10-4
Figure 10-6.	A Menu Created with DOMENU	10-5
Figure 10-7.	An Example of DRAWBOX	10-6
Figure 10-8.	The DRAWCHARS Example	10-8
Figure 10-9.	An Example of ERASEBOX	10-13
Figure 10-10.	An Example of INVERTBOX	10-17
Figure 10-11.	A STACKMSG Prompt Line	10-23

LIST OF TABLES

Table 2-1.	GRiDBASIC Reserved Words	2-4
Table 5-1.	A Table of Integer Functions	5-2
Table 5-2.	The AND Truth Table	5-5
Table 5-3.	The NOT Truth Table	5-20
Table 5-4.	The OR Truth Table	5-21
Table 5-5.	A Table of Ranges and Functions	5-26
Table 5-6.	The XOR Truth Table	5-35
Table 9-1.	Choosing MK\$ and CV Functions	9-5

GridBASIC COMMAND REFERENCE

ABS	5-3	INVERTLINE	10-20	XOR	2-11
ACOS	5-4	KILL	8-11	!	7-13
AND	5-5	LEFT\$	6-6	"	7-10
AS	9-5	LEN	6-7	#	7-14
ASC	6-2	LET	3-4	\$\$	7-17
ASIN	5-6	LOC	8-12, 9-7	&	7-13
ATN	5-7	LOCATE	7-9	'	3-11
CDBL	5-8	LOF	8-13, 9-8	*	5-1
CHR\$	6-3	LOG	5-17	##	7-16
CINT	5-9	LOG10	5-18	###	7-17
CLEARMSG	10-4	LSET	9-9	+	5-1, 7-15
CLOSE	8-3	MID\$	6-8	,	7-2
COMMA	7-2	MKD\$, MKI\$, MKS\$	9-10	-	5-1, 7-15
COS	5-10	MOD	2-8	/	5-1
CSNG	5-11	MOVEBOX	10-21	:	1-7
CVD, CVI, CVS	9-3	NEXT	4-3	;	7-19
DATA	3-6	NOT	5-20	<	2-8
DATE\$	7-4	ON GOSUB	4-11	=	2-8
DIM	3-2	ON GOTO	4-11	<=	2-8
DOMENU	10-5	OPEN	8-14, 9-11	>	2-8
DRAWBOX	10-6	OR	5-21	>=	2-8
DRAWCHARS	10-7	PI	5-22	<>	2-8
DRAWCIRCLE	10-9	PRINT	7-10	\	7-13
DRAWDOT	10-10	PRINT USING	7-12	^	2-5
DRAWLINE	10-11	PRINT#	8-17	~	2-12
ELSE	4-9	PRINT# USING	8-19	'	2-12
END	4-2	PUT	9-13	!	10-5
EOF	8-4	RANDOMIZE	5-23		
EOLN	8-5	READ DATA [RESTORE]	3-5		
ERASEBOX	10-12	REM	3-9		
ERASECIRCLE	10-14	RESTORE	3-6		
ERASEDOT	10-15	RIGHT\$	6-9		
ERASELINE	10-16	RND	5-24		
EXP	5-11	ROUND	5-28		
FALSE	5-13	RSET	9-9		
FIELD	9-5	SEMICOLON	7-19		
FIX	5-14	SGN	5-23		
FOR TO [STEP] NEXT	4-3	SIN	5-30		
GET	9-8	SPACE\$	6-10		
GETFILE\$	8-7	SQR	5-31		
GOSUB RETURN	4-6	STACKMSG	10-22		
GOTO	4-8	STEP	4-3		
IF THEN [ELSE]	4-9	STOP	4-15		
INKEY\$	7-5	STR\$	6-11		
INPUT	7-6	STRING\$	6-12		
INPUT#	8-8	TAB	7-20		
INPUT\$	8-10	TAN	5-23		
INSTR	6-4	THEN	4-9		
INT	5-15	TIME\$	7-21		
INVERTBOX	10-17	TRUNC	5-34		
INVERTCIRCLE	10-18	VAL	6-13		
INVERTDOT	10-19	WHILE WEND	4-14		

ABOUT THIS BOOK

This reference manual introduces the GRiDBASIC programming environment, including how to enter, run, and edit programs. It also covers the elements of syntax. Beyond that, the manual classifies and discusses each command in GRiDBASIC. Each discussion includes a brief, example program. We invite you to enter any that interest you and modify them as you wish.

Besides the Table of Contents, this manual includes an alphabetic Command Summary (following the Table of Contents) and an index.

NOTE: This is a reference manual, not a tutorial. If you have never programmed in the BASIC language, you would do well to find a book that teaches BASIC programming and/or take a class in BASIC.



CHAPTER 1: INTRODUCTION

This chapter introduces GRiDBASIC and its programming environment. It also touches on the following subjects:

- How to invoke GRiDBASIC and use its commands for writing, running, and listing programs
- The GRiDBASIC direct mode
- The GRiDBASIC editor and editing screen
- Lines and line numbering

WHAT IS GRiDBASIC?

GRiDBASIC meets and exceeds the requirements of the American National Standard for Minimal BASIC as described in document ANSI X3.60-1978. However, GRiDBASIC is much more than a "minimal" version -- it is compatible with full-featured industry standard versions of BASIC.

INVOKING GRiDBASIC

You can invoke GRiDBASIC in the same way that you invoke GRiD applications. For example, you could select a file of the appropriate kind, i.e., Basic. GRiDBASIC supports two modes: the programming or indirect mode and the direct mode. We will treat the programming mode first.

THE PROGRAMMING (INDIRECT) MODE

Once you have invoked GRiDBASIC, the program editing screen appears. Figure 1.1 below shows the editor screen ready for input in its initial form -- with no program listing. When you have the program editor on the screen, type your program.

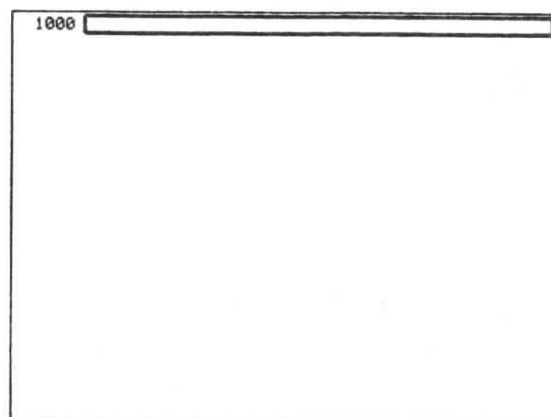


Figure 1-1. The Initial Program Editing Screen

The term "editor" refers to that portion of the GRiDBASIC software through which you type, modify, and list programs. Pressing RETURN at the end of each line generates a new line number and a new line.

When you write statements with line numbers, the computer waits for a CODE-R command before executing the program. We call this the "indirect mode," because statements don't execute directly when you complete a line. They wait in RAM memory, before, during, and after execution.

As with other GRiD applications, you press arrow keys to move within and between fields. Whenever you exceed the screen's depth (either by typing statements or by pressing a vertical arrow key), the displayed material "scrolls," letting you see previously undisplayed material.

After writing or editing a line, be sure to press RETURN, DownArrow, or CODE-RETURN before renumbering the listing or executing it. Any of these actions moves your newly written code from the keyboard buffer into RAM memory. Failure to take one of these three actions can cause the loss of the line in question.

Running, Continuing, and Stopping a Program

NOTE: GRiDBASIC does not ask you to type a RUN, LIST, or NEW command.

Press CODE-R to run (execute) the program you've typed. Whenever you run a program or execute a statement, GRiDBASIC displays the program's output instead of the current listing. If you have a STOP statement in your program, you can continue after the STOP line by pressing CODE-C.

When program execution encounters a STOP or END statement (or when you press ESC), you see the following message:

```
Program stopped at line nnnn
```

where nnnn is the statement's line number.

The ESCape Key

You can stop program execution at any point by pressing ESC. Press ESC once more and your listing comes back with the field outline where it was when you pressed CODE-R. If a syntax error stops your program, ESC returns you to your listing.

Erasing Line(s)

You can erase an entire line by placing the cursor on the line you want to erase, pressing CODE-E, and then confirming. To erase more than one line at a time, press the appropriate vertical arrow after pressing CODE-E. You can select as many lines for erasure as you want before confirming. If you start at the first line, press CODE-E, and follow it with CODE-SHIFT-DownArrow and CODE-RETURN, you erase the entire program.

Other Commands

Like GRiD applications, GRiDBASIC also has CODE-? to see and/or execute available commands (including Renumber), CODE-Q to Quit, CODE-T to Transfer and print files, CODE-U to see memory usage, and CODE-ESC to exit without saving the current file.

THE DIRECT MODE

The "direct mode" has no line numbers. When you confirm the line (CODE-RETURN), it executes and disappears from the computer's memory. People find the direct mode useful for quick computations and for debugging small segments of code. NOTE: All GRiDBASIC commands, except the looping commands (see Chapter Four), work in the direct mode.

To enter the direct mode, press DownArrow from the last statement field in the program. This creates a statement field with no number field. (Fields are discussed later in this chapter.) Enter the material you want to execute. For example,

```
PRINT 5+3
```

Press CODE-RETURN, RETURN, or DownArrow. NOTE: The material you entered disappears and the answer appears. To continue in direct mode, press ESC or DownArrow. The statement outline reappears. To return to the indirect mode and generate a line number, press RETURN. To position the cursor within the current listing, press UpArrow until the cursor reaches the desired line.

ABOUT THE GRiDBASIC ENVIRONMENT

This section gives details of the editing screen and discusses lines and line numbering.

Layout of the GRiDBASIC Screen

Figure 1-2 below shows the GRiDBASIC editor in the midst of working on a program. As you can see, the narrow column on the left displays program line numbers. We call it the "line number field." The wide column to the right displays program statements -- the actual "text" of the BASIC program. We refer to it as the "statement field." You can move within each field, and from one field to the other. A description of each element in the editor screen follows:

Statement Field	The field of the program editor screen where you enter and edit program statements. The statement field displays the "text" of your BASIC program.
Line Number Field	The field of the program editor screen where program line numbers are displayed. GRiDBASIC generates line numbers automatically. You

also can enter and edit line numbers manually. See the section below on "Lines and Line Numbering."

The Outline

A rectangular outline surrounds the current statement or line number field. When you first begin editing a program, the outline surrounds the statement field, indicating that you can enter or edit the text that composes your program.

Cursor

The blinking triangle within the outline. Its position indicates where your next keystroke will appear.

Highlighting

A form of display that causes text on the screen to show as dark-on-light when the other text is light-on-dark. (Also called "inverse video.")

Message line

The highlighted line displayed at the bottom of the screen. Your system prints command and error messages here.

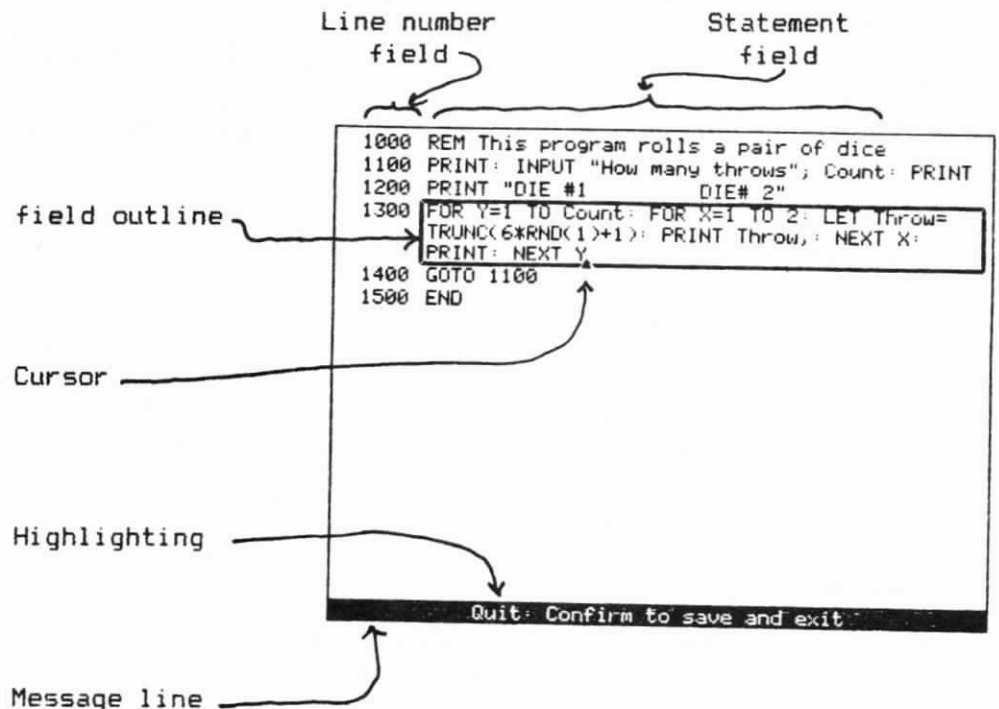


Figure 1-2. The Program Editor Screen

LINES AND LINE NUMBERING

Multiple statements and/or physical lines can follow any line number. Only the size of the screen limits the length of a line. When you type a line that exceeds the width of the statement field, the line breaks at the space character nearest to the end of the field and "text wraparound" automatically moves the next word onto the next line.

Any number from 1 to 64,000 constitutes a legal line number. GRIDBASIC supports automatic line numbering, manual line numbering, and renumbering. It also lets you reformat lines for enhanced readability.

Automatic Line Numbering

Automatic line numbering begins with line 1000. Each time you press RETURN, GRIDBASIC issues another line number. If you are not inserting a line between two existing lines, increments are by 100.

When you insert a line, the editor first tries to simply increment by 100. For example, a line inserted between 100 and 1000 receives the number 200. Increments become smaller as the difference between the two current line numbers shrinks. When the difference between the two line numbers is less than 100, GRIDBASIC next tries to increment by 10. The three remaining increments are by 5 (difference less than 10), 2 (difference less than 5), and 1 (difference of 2 or less).

Manual Line Numbering

To enter your own line numbers, position your cursor on the last display line of the program or on the last line number field. Press DownArrow. This causes a new statment field to emerge. Press LeftArrow to create a new line number field, then type the desired line number. Pressing RightArrow moves the cursor back into the statement field.

Renumbering

GRIDBASIC has a command that renumbers your line numbers so that they suit the current automatic numbering formula. To renumber, press CODE-? and select the Renumber option. In a moment, the command puts all statements into GRIDBASIC's default form -- with 1000 as the first line number and increments of 100 between all line numbers. This command also renumbers in the statement area so that numbers that point to lines (GOTO 2710) adjust correctly to the new line numbers.

Multiple Statements

You can put as many statements as the screen permits after a line number. However, you must place a colon (:) between each statement on the line. For example, you could put these three statements

```
1000 FOR X=1 TO 5
1100 PRINT X
1200 NEXT X
```

on one line:

```
1000 FOR X=1 TO 5: PRINT X: NEXT X
```

NOTE: Statements placed after a REM statement (See Chapter 3) do not execute.

Reformatting Your Listings

You can manually reformat your program listings in such a way as to add to the current number of display lines within a given program line. Press SHIFT-RETURN at that point in a program line where you want to begin a new display line.

The outline expands by one display line with the cursor positioned at the beginning of that new blank line. The program editor won't generate a new line number. Figure 1-3 breaks out the seven statements in line 1300 by inserting SHIFT-RETURNS after each the each statement. Compare it to Figure 1-2.

```

1000 REM This program rolls a pair of dice
1100 PRINT: INPUT "How many throws"; Count: PRINT
1200 PRINT "DIE #1      DIE# 2"
1300 FOR Y=1 TO Count:
      FOR X=1 TO 2:
        LET Throw=TRUNC(6*RND(1)+1):
        PRINT Throw,
      NEXT X:
      PRINT:
    NEXT Y
1400 GOTO 1100
1500 END

```

Figure 1-3. Line 1300 after Inserting SHIFT-RETURNS

To remove SHIFT-RETURN or other invisible characters, place the cursor to the right of the offending character and press BACKSPACE.

You terminate multi-line statements just as you do single line statements -- by pressing RETURN or either vertical arrow key. If you press RETURN, the statement field outline will move to the next line position and will return to its single line size.

If you choose an arrow key, you will generate a single statement field (unless the next program line also occupies multiple display lines). NOTE: If you press DownArrow from the last line of a program, you will get a statement field with no line number.

CHAPTER 2: GENERAL INFORMATION ABOUT GRiDBASIC

This chapter discusses concepts essential to programming:

- Syntax diagrams
- Reserved words
- Constants, variables, and arrays
- Expressions and operators
- File conventions
- Delimiters

SYNTAX DIAGRAMS

This book describes each GRiDBASIC and function according to the following conventions:

- We write BASIC statements and functions ("reserved words" -- see below) in all uppercase letters. For example, we render the statement that causes text to appear on the screen as PRINT. However, when you enter a statement or function, you can type any combination of upper- and lowercase. All the following iterations constitute legal forms of the PRINT statement:

PRINT print PrInt pRINT

- Variable names begin with a capital letter. For example:

```
LET A$=Name$
```

- You must supply any item shown in lowercase characters. For example, the GOTO command syntax

```
GOTO line#
```

means you must supply a line number. Failure to provide a required parameter results in a syntax error or unexpected program output. For example, GOTO 150, the legal form, tells the program to jump execution to line 150.

The following usages are illegal:

```
GOTO
GOTO bed
```

The first fails to supply a line number; the second supplies a character string.

- Items enclosed in square brackets ([]) are optional. The LET command syntax looks like this:

```
[LET] variableName=expression
```

This means that in assigning a constant to a variable, you may drop the "LET." Thus both

```
LET A=5
and
A=5
```

are correct and accomplish the same purpose within a program: they store the constant 5 in variable A.

- If you have a choice between two items, the choices are separated by a vertical slash (/) and surrounded by curly brackets ({}). For example, the syntax diagram

```
PRINT [expression][{,;}]
```

means that you have the option (note square brackets) of putting either a comma or semi-colon after an expression. Note that though you have a choice between the items in the curly brackets, you must supply one of them.

- A trailing ellipsis (three dots -- ...) indicates continuation. For example, in the DIM syntax statement,

```
DIM variableName(subscripts)[, variableName(subscripts)]...
```

the ellipsis at the end indicates that you can continue the variableName(subscript) pattern as many times as you want.

- A vertical ellipsis indicates that other statements may come between the first and last items. For example, you may place executable statements between FOR and NEXT.

```
FOR
.
.
.
NEXT
```

- When programming, you must include all punctuation -- commas, parentheses, semicolons, colons, or equal signs, as shown (except the syntax punctuation -- square brackets, curly brackets, and the vertical slash).

RESERVED WORDS

GRiDBASIC reserves the words that represent its statements, functions, operators, and constants for their individual tasks. Because they are reserved, you cannot use them as variables. However, you can place them within variables. The first two following examples are valid; the second two aren't.

```
INPUT "Type each extra item", REMainder$
LETter=ASC(Name$)
```

```
INPUT "Cost of unit"; VAL
LET Total=Data+(.065*Data)
```

In the first instance, the reserved word AND is hidden within the variable "Bands." In the second, a reserved word, DATA, is used as a variable.

This list also serves as a quick index to GRiDBASIC's statements, commands, operators, and functions. NOTE: The reserved words ON, and USING are not whole commands, but with other words make up such commands. These words are:

ABS	END	LEFT\$	PUT	VAL
ACOS	EOF	LEN	RANDOMIZE	WEND
AND	EOLN	LET	READ	WHILE
AS	ERASEBOX	LOC	REM	XOR
ASC	ERASECIRCLE	LOCATE	RESTORE	"
ASIN	ERASEDOT	LOF	RETURN	#
ATN	ERASELINE	LOG	RIGHT\$	\$
CDBL	EXP	LOG10	RND	%
CHR\$	FALSE	LSET	ROUND	'
CINT	FIELD	MID\$	RSET	)
CLOSE	FOR	MKD\$	SGN	(
CLRMSG	GET	MKI\$	SIN	*
COS	GETFILE\$	MKS\$	SPACE\$	+
CSNG	GOSUB	MOD	SQR	-
CVD	GOTO	MOVEBOX	STACKMSG	/
CVI	IF	NEXT	STEP	:
CVS	INKEY\$	NOT	STOP	;
DATA	INPUT	ON	STR\$	<
DATE\$	INPUT#	ON GOTO	STRING\$	=
DIM	INPUT\$	ON GOSUB	TAB	>
DOMENU	INSTR	OPEN	TAN	\
DRAWBOX	INT	OR	THEN	^
DRAWCHARS	INVERTBOX	PI	TIME\$	~
DRAWCIRCLE	INVERTCIRCLE	PRINT	TO	~
DRAWDOT	INVERTDOT	PRINT USING	TRUE	
DRAWLINE	INVERTLINE	PRINT#	TRUNC	
ELSE	KILL	PRINT# USING	USING	

Table 2-1. GRiDBASIC Reserved Words

CONSTANTS

Program execution operates on values that we call "constants." GRiDBASIC recognizes two kinds of constants: string constants and numeric constants.

String Constants

A string constant is a sequence of characters (ranging in length from 0 to 65,535). NOTE: You can only enter one screenful of characters at a time. A string constant can include any valid character. You must place double quotation marks before any string. If the string does not end the program line, you must also close it with double quotation marks. You can treat a number like a character string by placing it within quotation marks. See the examples of string constants (surrounded by their quotation marks) below:

```
"A"  
"$100,000,000.00"  
"Quarterly Profit Statement"  
"%Q*%?"  
"675"
```

Numeric Constants

Numeric constants are positive or negative numbers. GRiDBASIC operates on two different types of numeric constants: real numbers (also known as decimal or floating point numbers) and integers.

GRiDBASIC performs all numeric operations in double precision. This allows for 15 significant digits. GRiDBASIC handles numbers as small as 4.19E-307 and as large as 1.67E308.

Real numbers are positive or negative numbers that can include decimal points. GRiDBASIC works on double precision, real numbers and provides 15 digits of precision. Three examples of real number constants are:

```
0.12345678901234  
987654321.098765  
-1.1
```

NOTE: GRiDBASIC does not return numbers in scientific notation (also known as "E notation"). You can enter numbers with the carat (^) to indicate power (10^3 is the same as 10 cubed), but the GRiDBASIC will never print 10E3.

Integers are whole numbers between -32768 and +32767, inclusive. Integer constants do not have decimal points. GRiDBASIC stores all integer values in 15-digit format and converts them to real numbers before operating on them. No cost in speed results: Operations on real numbers are as fast as integer operations, because of special arithmetic hardware.

Here are examples of integer constants:

```
-10101  
0  
2001  
64
```

VARIABLES

A variable is a symbol. It stands for the memory address where the computer stores the expression you assign to the variable. Thus "A=15" tells the computer to store the value 15 at a position in memory that you have assigned the address "A." When BASIC executes the program it substitutes the constant found at the address for the variable.

Thus when the computer executes the command

```
PRINT A
```

it goes to the address labeled A, and prints the constant it finds there. If we use the constant assigned above, the number 15 would appear on the screen.

Variable names must begin with a letter, but the rest of the characters in the name can be any number, letter, or the decimal point. The length of a variable can range from one character to one full screen.

Variables, like constants, can be one of two types: string or numeric. The last character in a variable name identifies the variable's type.

String variables must end with dollar sign (\$). For example:

```
Name$  
DayOfWeek$
```

Integer variables must end with the percent sign (%). For example:

```
Age%  
Answer%(2,3)
```

Real variables can end with any character except a dollar sign (\$) or percent sign (%). GRI DBASIC assumes all variables are real variables, unless told otherwise. Thus the following are real variables:

```
Results  
Forecast_1983  
a123.0  
Radians*1.8
```

ARRAY VARIABLES

An array is a group of values referenced by a single variable name. Individual values in the array are called "elements." Because each element is itself a variable, you can place an element in an expression. You can also operate on it with any function or statement that takes variables as arguments.

Elements within an array are named with the array name combined with a number(s) enclosed in parentheses. For example, if an array name is `Month$` and consists of string variables that are the names of months, you might refer to December (an element of the array) as `Month$(12)` and January would be `Month$(1)`.

In the example above, the array named `Month$` is a one-dimensional array with 12 elements. An array can have up to 255 dimensions and a single dimension can have up to 65,535 elements. The maximum total number of elements you can place in an array is 65,535.

Thus you could have an array with 1 dimension and 65,535 elements and you could have an array of 255 dimensions, each dimension having 257 elements. When you access an element in a multi-dimensional array, you must specify the element's position within each dimension of the array.

The DIM statement specifies the number of dimensions that an array can have and the number of elements within each dimension. You do not have to dimension (DIM) variables, unless they have more than one dimension or more than ten elements. GRiDBASIC automatically expands the storage space required for string variables. All arrays start at 1 (one), not zero. Chapter Three describes the DIM statement in detail.

EXPRESSIONS AND OPERATORS

In its simplest form, an expression consists of a constant or a variable. You can also connect constants, variables, and functions with operators. GRiDBASIC has three types of operators: numeric, string, and logical.

Order of Precedence and Numeric Operators

GRiDBASIC supports the numeric operators listed below. We have listed them by their order of precedence, that is, by the order in which GRiDBASIC evaluates them when they appear in an expression.

()	Parentheses
	Functions (SIN, LOG, etc. -- see Chapter Six)
-	Unary minus (the negative sign)
^	Exponentiation
* / \ MOD	Multiplication, floating point division, integer division
+ -	Addition, Subtraction
	Relational Operators (See below)
	Logical operators (See below)

Relational Operators

The relational operators are a special sub-category of numeric operators and have the lowest precedence of all the numeric operators. These are the relational operators. NOTE: All of these operators have equal precedence.

<	Less Than
>	Greater Than
<=	Less Than or Equal To
>=	Greater Than or Equal To
=	Equal To
<>	Not Equal To

Logical Operators

The logical operators, listed in order of precedence, are:

- NOT
- AND
- OR
- XOR

In GRiDBASIC, logical operators perform logical (Boolean) operations by acting on every bit in the 16-bit integer value presented to it. To do this, GRiDBASIC follows three steps:

- ① Convert value to an integer
- ② Perform a bit-wise logical operation
- ③ Convert the integer back to a real number.

With the exception of NOT, a logical operator connects two or more operands and returns a true or false value. NOT is a unary operator (like the signs plus and minus) and simply changes the truth value of its operand. The results, "true" (not zero) or "false" (zero) value, form the basis for the computer to make a decision.

For example, in an IF statement, the computer takes one course of action when it finds a zero and another course in the case of a non-zero value. You must be careful in stating your logic. As an example, here are four programs, all of which try to trap any unwanted user responses. Their common theme is to act as an input filter. The program only accepts a response of "Y" or "N." If the user types any other character(s), the filter says "Sorry" and loops back to the INPUT statement.

Look at the first program in Figure 2-1.

```
1000 REM Logic 1 -- Type mismatch
1100 INPUT "Y or N"; Answer$
1200 IF Answer$ <> "Y" OR "N" THEN GOTO 1500
1300 PRINT "Thanks"
1400 END
1500 PRINT "Sorry, "; GOTO 1100
```

Figure 2-1. Example of a Type Mismatch Failure

It fails at line 1200 and issues a "Type mismatch" error message. Why? Because the program first evaluates the statement

```
Answer$ <> "Y"
```

Depending on the response, it returns either a Boolean 0 or 1. The OR then compares the Boolean to N, a string. And that creates a type mismatch, because GRiDBASIC won't compare data of differing types (in this case, Booleans and strings).

The program in Figure 2-2 won't find anything true. No matter what you type, it says "Sorry." Look closely at line 1200.

```
1000 REM Logic 2 -- evaluates nothing as true
1100 INPUT "Y or N"; Answer$
1200 IF Answer$ <> "Y" OR Answer$ <> "N" THEN GOTO 1500
1300 PRINT "Thanks"
1400 END
1500 PRINT "Sorry, "; GOTO 1100
```

Figure 2-2. Example of Faulty Logic

In this case the logic says, if the input is either not Y or not N then say "Sorry." But Y is not N and vice versa. Therefore, the logic fails Y and N, as well as everything else!

Now for a working example.

```
1000 REM Logic 3 -- evaluates correctly
1100 INPUT "Y or N"; Answer$
1200 IF NOT (Answer$="Y" OR Answer$="N") THEN GOTO 1500
1300 PRINT "Thanks"
1400 END
1500 PRINT "Sorry, "; GOTO 1100
```

Figure 2-3. A Working Logic

In Figure 2-3, the logic says if the response is neither "Y" or "N," then say, "Sorry." That's what we want, because if the response is one of those, we have an appropriate input.

This is not to say that line 1200 in Figure 2-3 is the only way of presenting this logic. Line 1200 in Figure 2-4 also works.

```

1000 REM Logic 4 -- AND also works
1100 INPUT "Y or N"; Answer$
1200 IF Answer$ <> "Y" AND Answer$ <> "N" THEN GOTO 1500
1300 PRINT "Thanks"
1400 END
1500 PRINT "Sorry, "; GOTO 1100

```

Figure 2-4. Another Working Logic

It says if the input is not "Y" AND not "N" say "Sorry." That's what we want, because that's what we see as an invalid response.

Chapter Five discusses each of the logical operators in details. It also covers the two Boolean constants, TRUE and FALSE.

String Operators

String operators perform relational comparisons and concatenation on string expressions.

The relational operators for string expressions are the same as those described for numeric operators, because you are actually performing numeric operations. For example, if you use the Less Than (<) operator to compare two strings, the ASCII values for each character in the two strings are compared to see which has the smaller numeric value. The result of the operation is thus a numeric result.

NOTE: GRIDBASIC makes comparisons of alphabetic characters without regard to capitalization. Thus the characters "A" and "a" are regarded as equal even though they have different ASCII values.) For a list of the relational operators, see the discussion in the preceding paragraphs.

The plus symbol (+) "concatenates" or joins strings together. The example below illustrates this.

```

A$="This is a "
B$="concatenated string"

```

Therefore, A\$ + B\$ becomes

```

"This is a concatenated string"

```

FILE NAMING CONVENTIONS

When your programs use the OPEN or KILL statements (discussed in Chapter Eight), they must specify the name of the file to be opened or deleted. You cannot perform these operations with the Transfer command (CODE-T). However, you can incorporate the file form used by the Transfer to get the file name information you need. To do this, use the GETFILE\$ command (see Chapter Eight). If you choose GETFILE\$, you can ignore the details of the Compass Computer operating system's file naming conventions given below.

You identify a file by specifying its "pathname". A pathname defines the route the computer takes to a file. A complete pathname includes the device and subject where the file is located plus its title, and kind. The complete pathname schema is as follows:

```
'device'subject'title~kind
```

Thus to specify a GRIDWRITE file with the title Forecast, when it resides under the subject Business on the bubble, you would enter,

```
'b'Business'Forecast~Text
```

The system defaults to the current device, subject, and kind. As a result, you don't have to respecify them. If you were staying within current defaults, you could open the example file by typing

```
1000 OPEN "I",2,"'Forecast"
```

where 1000 is the line number, "I" indicates sequential input, and 2 is the file tag number (See the OPEN statement in Chapter Eight for syntactical details.)

Note the two pathname delimiter characters: the left single quote (') or "tick" and the tilde (~). The tick must precede device, subject, and title names. Press CODE-' to print a tick. The tilde (~) must precede the kind. Generate this character by pressing the CODE-; combination.

If you specify a pathname that does not begin with the tick, the system assumes that the first name it encounters is the title and that you have left off the device and subject names. This limits the search for the title to the current directory, that is, to the current device and subject and makes file access quicker.

If you provide the complete pathname including device, subject, and title, the computer first searches all active devices for the subject. From the subject it then searches for the title. If the title is on-line, this process locates it.

The maximum length of subject and title names is 80 characters each.

Subject and title names can consist of any printing characters (including spaces) except the following:

- ' left single quotation mark ("tick")
- ~ tilde
- hyphen
- : colon

File Kinds

Placing the file kind (sometimes referred to as the file "type") after a title is optional. Kinds let you classify several related files under the same title while assigning them different "kind" characteristics. Interpretation of the kind is left up to the application. For a thorough discussion of file kinds refer to the "Compass Computer Operating System Reference Manual."

Delimiters

Delimiters are characters that set off certain programming elements from others, so that the language's "interpreter" can separate variables from operators from constants from reserved words. GRIDBASIC has only one delimiter, the space character.

Though you needn't place spaces around operators (with the exception of MOD), you should place them around variables and reserved words. Improper delimiting results in an "Improper syntax" message.

CHAPTER 3: ASSIGNMENT AND DEFINITION STATEMENTS

The statements in this chapter assign values to variables and define the size of arrays.

DIM

This statement establishes the dimensions of an array and allocates storage space for the specified number of elements.

FORMAT

```
DIM variableName(subscripts)[,variableName(subscripts)]...
```

NOTES

You must dimension (DIM) any array that consists of more than ten elements or more than one dimension. If an array variable name is used without the DIM statement, the maximum value of its subscript is 10.

A subscript is an expression that defines the maximum number of elements in an array. It follows the array variable name and is enclosed in parentheses. Thus,

```
Year(12,31)
```

is a two-dimensional array. The first subscript can contain as many as 12 elements, the second 31. The minimum value of a subscript is one, not zero.

The maximum number of dimensions that an array can have is 255. Dimensions can hold any number of elements as long as the total number of elements for the entire array does not exceed 65,535. Thus if you dimensioned an array to have one dimension containing 65,534 elements, you would have limited your program to having just one other dimension, and a dimension possessing only one element, at that!

If a program tries to reference more elements than the subscript allows, you will see the error message:

```
Array reference is out of range
```

You can establish the dimensions of more than one variable array at a time with a single DIM statement. Just separate the specification for each variable array with a comma. For example, the statement

```
1000 DIM ANSWERS(2,14,100),NAME$(20)
```

defines a numeric variable array with the name ANSWERS having three dimensions and 2800 elements (2 x 14 x 100). The same DIM statement defines a string variable array called NAME\$ with one dimension that can have 20 elements.

A note on string arrays: With GRiDBASIC, you don't have to dimension the actual strings, just the number of elements.

Subscripts can be numeric variables, instead of constants. Be sure that you have previously declared any such variable; otherwise, your subscript will amount to zero. For example,

```
1000 DIM GONZO(25, A, B)
```

dimensions a numeric array. The first dimension has an absolute maximum number of elements, 25. The second two depend on what value you (or the program) have previously assigned the numeric variables A and B. NOTE: If a variable should change its value later, the array will remain unaffected. It holds the original value until you redimension it.

To redimension an array, just restate its DIM statement with the desired values. Redimensioning automatically clears the old array.

EXAMPLE

```
1000 DIM Array(4)
1100 FOR X=1 TO 4
1200 Array (X)=X
1300 NEXT X
1400 FOR X=1 to 5
1500 PRINT Array(X)
1600 NEXT X
2000 END
```

In this example, line 1200 assigns values from the loop (the value of X) to each element in the array. We assign four values to four elements, so everything is fine.

However, line 1400 exceeds our dimension by one, so the program halts when it tries to handle this larger number, leaving the error message:

```
Array reference is out of range
Program stopped at line 1500
```

We can fix this problem by changing the 5 to a 4 in line 1400.

LET

The LET statement assigns the value of an expression to a variable.

FORMAT

[LET] variableName=expression

NOTES

LET is optional. You can write the variable followed by the equal sign and then the expression with the value to be assigned without the word LET.

The two following statements perform exactly the same; they both store the value 2 at the variable LoopCounter.

```
1000 LET LoopCounter=2
1000 LoopCounter=2
```

If you assign a numeric value to a string variable or a string value to a numeric value a Type Mismatch error occurs.

EXAMPLE

```
1000 LET X=20
1100 Y=X*3
1200 PRINT X
1300 PRINT Y
1400 END
```

The example assigns values to variables with and without the LET statement. Lines 1200 and 1300 print each value, showing that both ways work.

READ DATA [RESTORE]

READ, DATA, and RESTORE constitute a trio of statements that, as a team, assign values to variables. NOTE: RESTORE is optional.

FORMAT

```
READ variable[,variable][, ...]  
.  
.  
[RESTORE [line#]]  
.  
.  
DATA constant[,constant][, ... ]
```

NOTES

THE READ STATEMENT

The READ statement begins with the word "READ" and follows that with at least one variable. The following are legal READ statements:

```
1000 READ NewNumber  
  
1200 READ LastName$, Counter  
  
1300 READ A, B, C, D
```

The READ statement assigns its variable(s) the value(s) it finds in the program's DATA statement(s).

READ cannot operate alone; its program must contain at least one DATA statement. Each time the program executes a READ, it moves a pointer to the next item in the DATA statement list. When program execution begins, READ has its pointer set to the first item in the first DATA statement. When no more DATA items exist, a

Ran out of data

error occurs. You can reset this pointer with the RESTORE statement (see below).

When a program has more than one DATA statement, READ proceeds by line number, reading all the data in each program line before continuing to the next line. Within each line, it reads each item of data in order.

A READ statement can have both numeric and string variables. The values read from the DATA statement are assigned on a one-by-one basis to the variables. These values, however, must agree with the

variable types specified in the READ statement or a Type Mismatch error occurs.

One READ statement can take constants from one or more DATA statements, because GRIDBASIC strings the items in multiple DATA statements together in one long list. Similarly, more than one READ statement can operate on a single DATA statement. Each READ statement takes the next item in the DATA statement(s) list of items.

THE DATA STATEMENT

A DATA statement begins with the word "DATA" and follows it with a list of numeric and/or string constants. (A list can be as short as one item.) A comma must separate individual constants, but should not appear after the last item. For example,

```
2900 DATA 1492, "Nina, Pinta, Santa Maria", Columbus, 3.14
```

Numeric constants can be integer, real, or short real numbers. String constants can have up to 65,535 characters, the maximum length for any DATA statement. These strings require no quotation marks unless they contain commas, colons, or significant leading or trailing spaces. NOTE: Expressions are not permitted in DATA statements.

A program can include as many DATA statements as memory permits. You can place them anywhere within a program (even after an END); they are nonexecutable.

THE RESTORE STATEMENT

RESTORE resets the READ statement's pointer to the beginning of the specified line. If you don't specify a line, the pointer returns to the first DATA statement in the program and its first item.

EXAMPLE

```
1000 DATA agate, 365, boy, cow, 3.14, dog, elbow, foot, girl, 100
1100 READ A$
1200 PRINT A$,
1300 GOTO 1000
1400 END
```

This example mixes both string and numeric constants in its DATA statements. When line 1100 reads line 1000, it turns the numeric constants into strings. The example immediately below is exactly the same program, but with its DATA statement in a different position.

```

1000 READ A$
1100 PRINT A$,
1200 GOTO 1000
1300 END
1400 DATA agate, 365, boy, cow, 3.14, dog, elbow, foot, girl, 100

```

The results are exactly the same. See Figure 3-1

agate	365	boy	cow
3.14	dog	elbow	foot
girl	100		

Figure 3-1. Results of a Simple READ DATA Program

In the next example, the READ statement contains both a string and a numeric variable (see Figure 3-2). As the result of this program show, the two variables take turns drawing from the DATA statements.

```

1000 DATA agate, 365, ball, 123, cake, 3.14, doll, 100
1100 READ Noun$, Number
1200 PRINT Noun$;" is a noun"
1300 PRINT Number;" is a number"
1400 GOTO 1000
1500 END

```

```

agate is a noun
365 is a number
ball is a noun
123 is a number
cake is a noun
3.14 is a number
doll is a noun
100 is a number

```

Figure 3-2. Results of a READ with Two Variables

The example below contains three READ statements. In each case, a loop controls the number of times the READ statement acts. In this way, none of the statements runs out of data.

```
1000 FOR Counter=1 TO 3
1100 READ X
1200 PRINT "X = ";X,
1300 NEXT Counter
1400 PRINT: PRINT: RESTORE 2600: PRINT "A RESTORE statement
here":PRINT
1600 LET LoopCount=0
1700 WHILE LoopCount <> 9
1800 LET LoopCount=LoopCount+1
1900 READ Y
2000 PRINT "Y = ";Y,
2100 WEND
2200 PRINT: PRINT: RESTORE 2600: PRINT "A RESTORE statement here":
PRINT
2300 READ Z
2400 IF Z<> 11 THEN PRINT "Z = ";Z, ELSE END
2500 GOTO 2300
2600 DATA 1, 2, 3, 4
2700 DATA 6, 7, 8, 9, 10, 11
```

REM

This statement lets you insert explanatory remarks into a program.

FORMAT

{REM remarks ; 'remarks}

NOTES

GRiDBASIC does not execute REM statements. They only appear when you display or print the program listing.

GRiDBASIC also recognizes a single quotation mark or apostrophe (') as a REM statement. If you branch into a REM statement (from a GOSUB or GOTO statement), execution continues with the first executable statement after the REM statement.

You can put a REM statement on a multiple statement line by separating it from preceding statements in the normal way, with a colon. NOTE: Program execution ignores any statements that follow a REM statement within the same program line. Note further that REM's take up memory space and slow program execution.

EXAMPLE

```
1000 REM This text is a remark.  
1100 'This text is a remark.  
1200 REM The next statement won't print: PRINT "You're Right!"  
1300 END
```

The first two statements are equivalent. The final statement demonstrates what happens to commands placed after a REM. Nothing! If you run this program, all you will get is a blank screen.

CHAPTER 4: STATEMENTS THAT CONTROL PROGRAM FLOW

This chapter describes statements that alter or halt normal statement-by-statement execution and allow loops and conditional execution of statements.

END

This statement terminates program execution and closes all files that were opened.

FORMAT

END

NOTES

You can terminate program execution with either the END statement or the STOP statement (described later in this chapter). END differs from STOP in that END closes all files. Therefore, you cannot resume program execution with the Continue command (CODE-C). When an END is encountered, the following message is displayed:

Program stopped at line nnnn

where nnnn is the line number where the END was encountered. You can return to the program editor by pressing any key. You can begin program execution again by pressing CODE-R (the Run command).

An END statement at the end of a program is optional. If there is no END statement at the end of the program, files remain open until you exit the program with the Quit command (CODE-Q).

Regardless of when the END statement is encountered and executed, it always causes termination of program execution.

EXAMPLE

```
1000 LET A=5
1100 INPUT "A equals 5. How much should B equal ";B
1200 IF A<>B THEN END
1300 PRINT "Good-bye for now..."
1400 END
```

The END statement can live within a program as easily as it does at the end. This example contains two END statements -- one at the end and the other within another statement (line 1200). The logic says to end if the variables A and B are unequal. If they both equal 5, print a "good-bye" message before ending.

FOR TO [STEP] NEXT

The FOR ,TO, and NEXT trio of statements create a program loop. Instructions within this loop repeat each time the loop executes. These statements help define the range, increments, and number of loops. Programmers often refer to these as "For Next loops."

FORMAT

```
FOR variable1=expression1 TO expression2 [STEP expression3]
.
.
.
NEXT variable1
```

NOTES

The following list describes the four parameters taken by FOR NEXT.

variable1: A variable that acts as a counter.

expression1: The initial value or setting for the counter.

expression2: The final or limit value of the counter.

expression3: The increment value added to or subtracted from the counter after each pass through the loop. This expression is optional. If you don't specify a value, GRiDBASIC assigns a value of 1 (one).

When program execution encounters the FOR statement, it checks to determine if the initial value (expression1) of the counter (variable1) is greater than the final value (expression 2). If it is already greater, the body of the loop is skipped and the statement following the NEXT statement is executed.

If it is not greater, the program lines following the FOR statement are executed until the NEXT statement is encountered. At that point, the counter (variable1) is incremented (or in the case of a negative STEP, decremented) by the amount specified in expression3. Program execution then branches back to the FOR statement and the process is repeated.

If the STEP value (expression3) is a negative, the logic just described is reversed. The loop is skipped when the counter (expression2) is less than the final value and the counter is decremented after each pass through the loop.

If expression3 (the STEP increment/decrement) evaluates to zero, an endless loop occurs, unless you provide some method of setting the

counter greater than the final value.

You can "nest" FOR NEXT loops (place one FOR NEXT loop inside another) to whatever depth you want; you are limited only by the amount of available memory. When you nest loops, you must provide a unique variable name for each loop counter.

Make sure that the NEXT statement for an inside loop appears before that of an outside loop. A loop like

```
1000 FOR X=1 TO 5
1100 FOR Y=1 TO 10
1200 PRINT X, Y
1300 NEXT X
1400 NEXT Y
```

causes an "Improper loop nesting" error message. Reversing lines 1300 and 1400 would solve the problem.

NOTE: You can jump out of a FOR NEXT loop, but NEVER jump into the middle of such a loop. The reason is that such jumps usually fail to properly initialize the counter and loop limits.

EXAMPLE

```
1000 FOR Counter = 1 TO 5
1100 PRINT "Counter now equals ";Counter
1200 Next Counter
1300 PRINT "Counter equals ";Counter
1400 END
```

This first example shows a simple loop. The new value of the counter prints each of the five times the program executes the loop. When the value of the counter reaches 6, the counter fails the test and execution passes out of the loop to print the end value of the counter (line 1300).

As with other examples, feel free to modify this example, playing with your own loop sizes and controls. A more complex example follows.

```
1000 REM The outer Loop begins on the next line
1100 FOR OuterLoop=1 TO 5
1200 PRINT: PRINT "Outer Loop number"; OuterLoop; " and counting..."
1300 REM The inner or "nested" loop is next
1400 FOR InnerLoop=27 TO 0 STEP -3
1500 PRINT InnerLoop; " ";
1600 NEXT InnerLoop
1700 REM That's it for the inner loop
1800 PRINT
1900 NEXT OuterLoop
2000 REM And that's it for the outer loop
2100 END
```

This example demonstrates nested loops, use of the STEP instruction and negative STEPs. The outer loop begins at line 1100 and ends at line 1900. The inner loop begins at line 1400 and ends on line 1600. Steps in the inner loop decrement by units of -3.

GOSUB RETURN

The GOSUB statement transfers control to a subroutine at specified line number. The RETURN statement must appear at the end of that subroutine and returns control to the main program.

FORMAT

```
GOSUB line#  
.  
.  
subroutine  
RETURN
```

NOTES

The line# is the line number of the first line of the subroutine. (A subroutine is one or more statements that performs a distinct task). A GOSUB jumps program execution to a subroutine. (This is sometimes referred to as "making a subroutine call".)

When the RETURN statement is encountered at the end of that subroutine, it causes execution to return to the statement following the most recent GOSUB statement. You can have more than one RETURN statement within a subroutine for situations where you want to exit the subroutine at different points.

If the specified line number contains a non-executable statement (for example a DATA, REM, or DIM), execution will begin at the first subsequent executable statement after line#.

You can call a subroutine as many times as you want, and you can call one subroutine from within another. Your only limit on this nesting of subroutines is the amount of available memory.

EXAMPLE

```
1000 GOSUB 1900
1100 PRINT "Dive! Dive!"
1200 GOSUB 1900
1300 GOSUB 1700
1400 GOSUB 1900
1500 PRINT: PRINT "At last. The END is in sight."
1600 END
1700 PRINT:PRINT "So this is a 'GOSUB.' Time to surface..."
1800 RETURN
1900 REM The next line just loops for time
2000 FOR X=1 TO 400: NEXT X
2100 RETURN
```

In this example, we have two subroutines -- one beginning at line 1700 and the other at line 1900. We call the subroutine three times during program execution to put time between the execution of the print statements. Note that the subroutine at line 1900 begins with a REM, not with an executable statement.

GOTO

This statement causes an unconditional transfer of control to a specified line number.

FORMAT

GOTO line#

NOTES

The GOTO statement differs from the GOSUB statement in that it lacks a RETURN statement. Any return of program execution to the line following the GOTO line must be forced by another GOTO.

If line# specifies a line containing an executable statement, then that statement and those following it are executed. If the specified line does not contain an executable statement (for example, a DATA statement), then execution continues at the first executable statement after the line specified by line#. The intervening lines are simply ignored.

NOTE: GRiDBASIC does not support an "implied GOTO" as in the example

```
2200 IF A=B THEN 1700
```

EXAMPLE

```
1000 PRINT "This line (1000) contains a GOTO statement.": GOTO 1300
1100 Print "This is the line (1100) after the first GOTO."
1200 GOTO 1500
1300 PRINT "This is the first line (1300) you went to."
1400 GOTO 1100
1500 PRINT "The END. (See message below.)": END
```

In this example, program execution jumps from line 1000 down to line 1300. It then jumps back up to line 1100. Execution moves straight down from 1100 to 1200 where the final GOTO appears, sending execution to the last line.

To understand the term "infinite" loop, remove line 1200 from this program and watch what happens. Remember: Pressing ESC will stop any such loop.

IF THEN [ELSE]

These statements allow the conditional execution of one of two statements or a series of statements, based on the result of an expression evaluation.

FORMAT

```
IF expression THEN statement1[:statement1a:statement1b ... ][ELSE  
statement2[:statement2a:statement2b ... ]
```

NOTES

If the expression following IF is true (not zero), the statement following THEN executes. If an ELSE statement exists, execution skips it.

If the result of the expression evaluation is false (zero), program execution skips any statement(s) following THEN and executes any ELSE statement(s). If no ELSE statement exists, execution goes to the next line number.

ELSE statements are optional. An ELSE statement only executes when the IF statement evaluates as zero (false).

Look at the following example:

```
1000 IF A=B THEN PRINT "Equal": GOTO 1500 ELSE PRINT "Unequal"  
1100 GOSUB 2000
```

If A does equal B, then the computer will print the word "Equal" and it will jump to line 1500. However, if A does not equal B, neither the THEN statement nor the GOTO will execute. Instead, the program will execute the ELSE statement and print the word "Unequal" before continuing to line 1100.

You can follow the THEN and ELSE statements with as many statements as you want. The statements must be separated by colons (:) and can be either on the same line or on a new line. If on a new line, the statements cannot have a new line number; they are considered part of the same line as the THEN or ELSE statement. Thus both of the following sequences are valid:

```
100 IF A=B THEN C=D:  
    E=F  
    ELSE J=K  
200 ...
```

```
100 IF A=B THEN C=D:E=F ELSE J=K
```

Note the absence of a colon between E=F and ELSE in the second example. The ELSE statement (if present) is also considered to be part of the same line as the IF and THEN statements and should not be separated from the preceding statement by a colon nor should it have a new line number. Therefore, the following sequences are invalid:

```
100 IF A=B THEN C=D:E=F
200 ELSE J=K
```

and also

```
100 IF A=B THEN C=D:E=F:ELSE J=K
```

You can nest IF THEN ELSE statements to any depth: you are limited only by the amount of available memory. If the statement does not contain the same number of ELSE and IF THEN clauses, each ELSE is matched with the closest unmatched IF THEN.

NOTE: The word "THEN" must always follow an IF clause. The following statement, omitting THEN, is not valid.

```
2200 IF A=B GOTO 1700
```

EXAMPLE

```
1000 Even$="Even Steven!": Odd$="Odd Bodkins!"
1100 INPUT "Try some conditions (Y/N and confirm)";Answer$
1200 IF Answer$="N" THEN PRINT "Whatever you say.": GOTO 1600
1300 INPUT "Type any integer and confirm (ESC to stop):",Number
1400 If Number MOD 2=0 THEN PRINT Even$ ELSE PRINT Odd$
1500 GOTO 1300
1600 END
```

This example contains two IF THEN statements -- lines 1200 and 1400. The line 1200 statement lacks an ELSE, but attaches a statement after THEN. If the condition is true, both will execute. The message "Whatever you say," prints and the program jumps to the END statement. However, if the statement evaluates as false, then execution falls through to the next line (1300).

Line 1300 contains a straightforward IF THEN ELSE statement. If the modulo test yields a 0, print the string variable for "even." If false, ELSE prints the "odd" string.

NOTE: The ESC key function mentioned in line 1300 comes from the system, not from this program. Remember: You can press ESC to halt execution of any GRiDBASIC program.

ON GOTO and ON GOSUB

These statements cause an unconditional transfer of control to one of several specified line numbers. The particular lines depend on the result obtained by evaluating the expression following the ON statement.

FORMAT

```
ON expression GOTO line#[,line#]...  
and  
ON expression GOSUB line#[,line#]...
```

NOTES

ON GOTO does in one statement what IF THEN would take numerous statements to achieve: it takes an expression and uses its value to send program execution to a particular line number.

In the example below, if the variable ANSWER evaluates to 2, program execution jumps to the second line number in the list, 1500.

```
500 ON Answer GOTO 1000,1500,2000
```

NOTE: GRIDBASIC rounds the expression value to an integer, if necessary. If the expression value is zero, or if it is greater than the number of line numbers in the list, execution simply continues with the next executable statement in the program.

EXAMPLE

```
1000 INPUT "Enter a number from 1 to 5 and confirm",A  
1100 ON A GOTO 1200, 1300, 1400, 1500, 1600  
1110 PRINT "Your entry is out of range.": GOTO 1000  
1200 PRINT "ONE": GOTO 1000  
1300 PRINT "TWO": GOTO 1000  
1400 PRINT "THREE": GOTO 1000  
1500 PRINT "FOUR": GOTO 1000  
1600 PRINT "FIVE": GOTO 1000
```

This example prints the name of the number given the INPUT statement. This is a typical use for ON GOTO in that particular values must connect with particular items. A more complex example might connect a U.S. President's order in the Presidency with his name.

If you enter a number greater than five or less than one, execution will drop through the ON GOTO statement to the next line, an error

message and a GOTO sending execution back to the INPUT statement.

Note the GOTO statements following each of the line numbers in the list (1200-1600). Without such an ending statement (you could use END, too), execution continues and prints all subsequent numbers. Hardly our purpose.

When you run this program, give it some out of range numbers and some decimals to see what happens. An example of ON GOSUB follows.

```
1000 INPUT "Enter a number from 1 to 5 and confirm",A
1100 ON A GOSUB 1300, 1400, 1500, 1600, 1700
1200 IF A<1 OR A>5 THEN GOTO 1800 ELSE GOTO 1000
1300 PRINT "ONE": RETURN
1400 PRINT "TWO": RETURN
1500 PRINT "THREE": RETURN
1600 PRINT "FOUR": RETURN
1700 PRINT "FIVE": RETURN
1800 PRINT "Out of range": GOTO 1000
```

This is the same program except that an ON GOSUB statement guides execution to the proper line number. And because this is a GOSUB, a RETURN statement must end the one line subroutine.

RETURN sends execution to line 1200. To accomodate this, and still be able to issue an "Out of range" message, line 1200 contains a new logic. It checks to see if the input is within range. If it is, the program loops to the first line again. If not, execution goes to the error message on line 1800 before going to the first line.

STOP

The STOP statement suspends program execution.

FORMAT

STOP

NOTES

The STOP statement suspends program without closing any files. STOP serves as a good debugging tool; you can halt execution, check the status of variables, and then continue. You continue program execution by pressing CODE-C (the Continue command). Pressing any key return you to the program editor.

When a STOP is encountered, the following message appears:

Program stopped at line nnnn

where nnnn is the line number where the STOP was encountered.

EXAMPLE

```
1000 A$= "Hit the brakes!!!"  
1100 B$= " There's a STOP line just ahead."  
1200 C$=A$+B$  
1300 PRINT "Screeeeeeeeech"  
1400 STOP  
1500 PRINT C$
```

This example declares and concatenates two string variables. The STOP at line 1400 gives you the chance to preview the concatenation before executing it. By entering the direct mode and typing "PRINT C\$," you can see what C\$ looks like. Press CODE-C to continue.

WHILE WEND

These statements create a program loop that continues to execute as long as the WHILE statement evaluates as true.

FORMAT

```
WHILE expression
    .
    statement(s) and/or functions
    .
WEND
```

NOTES

If the result obtained by evaluating the expression is true (not zero), the statement or statements between the WHILE and WEND statements will be executed. WEND returns execution to the WHILE statement for another evaluation of the expression.

The intervening statements execute until the expression evaluates to zero (false). If the expression evaluates to zero the first time it is encountered, then the intervening statements will not execute at all. After the expression evaluates to zero, execution continues with the first executable statement following the WEND statement.

You can nest WHILE WEND statements to any depth; you are limited only by the amount of available memory. Program execution matches each WEND with the most recent WHILE. If you have unequal numbers of WHILE and WEND statements, an error will occur -- "Improper loop nesting error."

If you write FOR NEXT loops inside of WHILE WEND loops (or vice versa), be sure the inner loop lies entirely within the outer loop.

EXAMPLE

```
1000 LET GuessMe=TRUNC(5*RND(1)+1)
1100 WHILE UserGuess <> GuessMe
1200 INPUT "Guess a number between 1 and 5";UserGuess
1300 WEND
1400 PRINT "You got it! The number was ";GuessMe
1500 INPUT "Want to try again (Y or N)";YesNo$
1600 IF YesNo$ = "Y" THEN GOTO 1000 ELSE PRINT "Okay, bye!": END
```

This example is a guessing game that asks you to enter a number. The WHILE statement then tests to see if you guessed correctly. If the number qualifies, program execution falls through the WHILE WEND loop to the message. If the comparison fails, execution stays within the WHILE WEND loop, asking for another input.

CHAPTER FIVE: GRiDBASIC ARITHMETIC AND LOGIC

This chapter describes GRiDBASIC's arithmetic statements, functions, and constants. Chapter Five also discusses the GRiDBASIC's four logical operators -- AND, NOT, OR, and XOR -- plus its two Boolean constants, TRUE and FALSE. Additionally, it covers the two integer operators: integer division and MOD.

NOTE: Although not documented like other operators, GRiDBASIC has the four essential arithmetic:

- + (Addition)
- - (Subtraction)
- * (Multiplication)
- / (Long Division)

See Chapter Two for details on precedence among arithmetic, relational, and logical operators.

This chapter opens with a discussion of GRiDBASIC's six integer functions. It also discusses them individually within the chapter.

INTEGER FUNCTIONS

GRiDBASIC has six ways of converting floating point numbers to integers:

- CINT
- FIX
- INT
- ROUND
- TRUNC
- Assignment of a value into an integer variable (symbolized as VAR% below).

Table 5-1 below illustrates how GRiDBASIC applies its various functions to converting floating point numbers. NOTE: To ensure accuracy when converting decimals to integers, choose either ROUND or TRUNC. GRiDBASIC includes the CINT, FIX, and INT functions for compatibility with other BASIC's. The table below shows that ROUND performs the same as CINT and TRUNC acts like FIX.

		FUNCTION					
		CINT	FIX	INT	ROUND	TRUNC	VAR%
INPUT	-3.50	-4	-3	-4	-4	-3	-4
	-3.49	-3	-3	-4	-3	-3	-3
	3.49	3	3	3	3	3	3
	3.50	4	3	3	4	3	4
	33000.00	-32536	-32536	-32536	33000	33000	-32536

Table 5-1. A Table of Integer Functions

A discussion of each of GRiDBASIC's arithmetic functions begins on the next page. NOTE: GRiDBASIC cannot guarantee accurate integers whenever you give it a number that exceeds the boundaries of integer arithmetic: -32768 to +32767 inclusive.

ABS

This function returns the absolute value of its expression.

FORMAT

ABS(expression)

NOTES

The absolute value of the expression is the value unsigned. ABS strips away the minus sign of negative numbers. The absolute value of a number is always positive or zero.

EXAMPLE

```
1000 INPUT "Enter a number and confirm",A
1100 B=ABS(A)
1200 PRINT "Absolute value is ";B
1300 GOTO 1000
1400 END
```

ACOS

The arc cosine function.

FORMAT

ACOS(expression)

NOTES

This function takes an expression representing an angle in radians and returns its arc cosine (in the range of 0 to π). GRiDBASIC evaluates this expression in full precision. To convert from degrees to radians, multiply by $\pi/180$.

EXAMPLE

```
1000 INPUT "Enter a number between -1 and 1";Number
1100 PRINT
1200 Rads=ACOS(Number)
1300 Degrees=Rads*(180/PI)
1400 PRINT "The arc cosine of "; Number; " is ";Degrees; "
degrees"
1500 PRINT: PRINT
1600 GOTO 1000
1700 END
```

AND

The logical operator for conjunction

FORMAT

expression1 AND expression2

NOTES

The AND function unites elements, calculates their combined truth value, and issues a Boolean true or false. As the AND truth table (Table 5-2 below) shows, AND only issues a true (non-zero) when both elements are true.

A	B	A AND B
---	---	-----
-1	-1	-1
-1	0	0
0	-1	0
0	0	0

Table 5-2. The AND Truth Table

EXAMPLE

```
1000 PRINT "Separate the two numbers with a comma":PRINT
1100 INPUT "Type two numbers between 1 and 5"; A,B
1200 IF A=3 AND B=4 THEN PRINT "You win!" ELSE PRINT "Try again"
1300 PRINT:GOTO 1100
1400 END
```

As long as you enter both elements correctly (3,4), you win. Any other combination fails.

ASIN

The arc sine function.

FORMAT

ASIN(expression)

NOTES

This function takes an expression representing an angle in radians and returns the arc sine of that angle. GRiDBASIC evaluates this expression in full precision. Arc sines fall into the range of $-\pi/2$ to $\pi/2$. To convert from degrees to radians, multiply by $\pi/180$.

EXAMPLE

```
1000 INPUT "Enter a number between -1 and 1";Number
1100 PRINT
1200 Rads=ASIN(Number)
1300 Degrees=Rads*(180/PI)
1400 PRINT "The arcsine of "; Number; " is ";Degrees; " degrees"
1500 PRINT: PRINT
1600 GOTO 1000
1700 END
```

ATN

The arc tangent function.

FORMAT

ATN(expression)

NOTES

This function takes an expression representing an angle in radians and returns the arc tangent of that angle. GRiDBASIC always evaluates this expression in full precision. The result falls in the range of $-\pi/2$ to $+\pi/2$. To convert from degrees to radians, multiply by $\pi/180$.

EXAMPLE

```
1000 INPUT "Enter a number ";Number
1100 PRINT
1200 Rads=ATN(Number)
1300 Degrees=Rads*(180/PI)
1400 PRINT "The arc tangent of "; Number; " is ";Degrees; "
degrees"
1500 PRINT: PRINT
1600 GOTO 1000
1700 END
```

CDBL

The convert to double precision statement

FORMAT

CDBL(expression)

NOTES

Because GRiDBASIC performs all operations in double precision, this statement does nothing. It exists only for compatibility's sake. See CSNG below.

EXAMPLE

```
1000 LET Some=CDBL(4)
1100 PRINT Some
1200 END
```

Put any number you want in the parentheses. Line 1100 displays it just as you entered it.

CINT

The CINT (convert to integer) function converts an expression to an integer.

FORMAT

CINT(expression)

NOTES

CINT performs the conversion by rounding the fractional portion of the number.

NOTE: This function is identical to the GRiDBASIC's ROUND function described later in this chapter. The existence of both functions enhances the compatibility of GRiDBASIC with other BASIC's. See the discussion of integer functions at the beginning of this chapter.

EXAMPLE

```
1000 INPUT "Enter any number and confirm", Decimal
1100 Answer=CINT(Decimal)
1200 PRINT "The CINT integer is ";Answer;:PRINT
1300 GOTO 1000
```

COS

The cosine function.

FORMAT

`COS(expression)`

NOTES

This function takes an expression representing an angle in radians and returns the cosine of that angle. GRiDBASIC always evaluates this expression in full precision. To convert from degrees to radians, multiply by $\pi/180$.

EXAMPLE

```
1000 INPUT "Enter angle (in degrees) and confirm",Angle
1100 Rads=Angle*(PI/180)
1200 Calculation=COS(Rads)
1300 PRINT "The cosine of ";Angle;" degrees is "; Calculation:
PRINT
1400 GOTO 1000
```


CSNG

The convert to single precision statement

FORMAT

CSNG(expression)

Because GRiDBASIC performs all operations in double precision, this statement does nothing. It exists only for compatibility's sake. See CDBL above.

EXAMPLE

```
1000 LET Some=CSNG(4)
1100 PRINT Some
1200 END
```

Put any number you want in the parentheses. Line 1100 displays it just as you entered it.

EXP

The exponential function, referred to in mathematics as "e."

FORMAT

EXP(expression)

NOTES

In GRiDBASIC, a natural logarithm has a base of

2.718281882845905

The EXP function raises this base number to the power given as its expression. Thus

EXP(2)

equals 2.718281882845905 squared.

LOG is the inverse function of EXP, as demonstrated by the example program below. For this reason "the exponential of" and "the natural antilogarithm of" are synonymous phrases.

If the expression evaluates to greater than or equal to approximately 200, an overflow occurs.

EXAMPLE

```
1000 INPUT "An exponent please";Anex
1100 LET Answer1=EXP(Anex)
1200 PRINT "The natural log's value raised to the power ";Anex;"
is"; PRINT Answer1: PRINT
1300 Answer2=LOG(Answer1)
1400 PRINT "The natural log of this number is ";Answer2: PRINT
1500 Answer3=LOG10(Answer1)
1600 PRINT "Its log to the base 10 is ";Answer3:PRINT
1700 GOTO 1000
1800 END
```

FALSE

The Boolean constant for false.

FORMAT

FALSE

NOTES

The constant FALSE has a value of 0. Statements can interact with it in a number of ways. You can assign its value to variables, operate it on it logically, print it. The program below does all these things.

EXAMPLE

```
1000 PRINT "True=";TRUE; " and False=";FALSE
1100 INPUT "Type the number 3"; A
1200 IF A=3 THEN B=TRUE ELSE B=NOT TRUE
1300 PRINT B;
1400 IF B=FALSE THEN PRINT " means you didn't type 3" ELSE PRINT
" means you typed 3"
1500 PRINT:GOTO 1100
1600 END
```

Line 1000 prints the values of GRiDBASIC's two Boolean constants. Whenever you use TRUE or FALSE, you use the constant's value. For example, depending on the value of A, line 1200 does one of two things. It either assigns -1 to B (TRUE) or applies NOT to TRUE, changing the -1 to its opposite, a zero (0). Note that although the program never assigns "FALSE" to the variable B, it can evaluate B as "FALSE" (line 1400), if in line 1200 B proves to be "NOT TRUE."

FIX

The FIX function converts an expression to an integer.

FORMAT

FIX(expression)

NOTES

This function converts an expression to an integer by removing all numbers to the right of the decimal point. The difference between this function and the CINT and INT functions is that FIX does not round negative numbers down. Thus -2.3 and 2.9 both become -2.

Thus FIX (an "import" from other BASIC's) works like GRiDBASIC's own TRUNC function. See the section at the beginning of this chapter, comparing the different integer functions. Also see the TRUNC function later in this chapter for more details.

EXAMPLE

```
1000 INPUT "Enter any number and confirm", Decimal
1100 Answer=FIX(Decimal)
1200 PRINT "The FIX integer is ";Answer :PRINT
1300 GOTO 1000
```

INT

The INT function converts an expression to an integer.

FORMAT

INT(expression)

NOTES

GRiDBASIC performs the conversion by rounding down the fractional portion of the number. Thus a positive whole number remains the same regardless of the value of the number to the right of the decimal point.

In the case of negative numbers, however, INT rounds the number to the next smaller whole number. Thus with INT -2.3, -2.5, -2.9 all become -3. Because of this action, INT is sometimes referred to as a "floor function."

GRiDBASIC includes INT for compatibility with other BASIC's. See the article on integer functions at the beginning of this chapter for more information.

EXAMPLE

```
1000 INPUT "Enter any number and confirm", Decimal
1100 Answer=INT(Decimal)
1200 PRINT "The INT integer is "; Answer: PRINT
1300 GOTO 1000
```

INTEGER DIVISION (\)

The integer division operator.

FORMAT

dividend \ divisor

NOTES

Integer division acts like ordinary division (/) in that it delivers a quotient. Unlike, ordinary division, it does not issue a remainder. Thus the operation

```
PRINT 5\2
```

yields 2, not 2.5. NOTE: You make a back slash, the integer division sign, by pressing the CODE-SHIFT-' combination.

The MOD function is just the opposite of integer division; it prints the remainder, but not the quotient. (See MOD later in this chapter.)

EXAMPLE

```
1000 INPUT "Divide 51 by what number"; Divisor
1100 LET Quotient=51\Divisor
1200 LET Remainder=51 MOD Divisor
1300 PRINT "The quotient is ";Quotient; " with a remainder of
";Remainder
1400 PRINT: GOTO 1000
1500 END
```

This example shows the integer division quotient and the MOD remainder that result from dividing 51 by your input divisor. The second example asks you for both the dividend and the divisor; it then calculates the results from floating point division, integer division, and MOD.

```
1000 INPUT; "Dividend";N
1100 INPUT " Divisor";D
1200 PRINT "FPDiv=";N/D,
1300 PRINT "IntDiv=";N\D,
1400 PRINT "MOD=";N MOD D
1500 PRINT: GOTO 1000
1600 END
```

LOG

The (natural) logarithm function

FORMAT

LOG(expression)

NOTES

This function returns the natural logarithm of an expression. The value of the expression must be a positive number greater than zero.

EXAMPLE

```
1000 INPUT "An exponent please";Anex
1100 LET Answer1=EXP(Anex)
1200 PRINT "The natural log's value raised to the power ";Anex;"
is": PRINT Answer1: PRINT
1300 Answer2=LOG(Answer1)
1400 PRINT "The natural log of this number is ";Answer2: PRINT
1500 Answer3=LOG10(Answer1)
1600 PRINT "Its log to the base 10 is ";Answer3:PRINT
1700 GOTO 1000
1800 END
```

This example calculates the exponential of a number, its inverse (the LOG), and finally, the common logarithm (to the base 10).

LOG10

Logarithm to base 10.

FORMAT

LOG10(expression)

NOTES

This function returns the logarithm to the base 10 of an expression (NOTE: Natural logarithms have a base of 2.718). The log to the base 10 is the number to which you have to raise 10 to get a particular number. Thus log of 1000 is 3, because 10^3 yields 1000

The value of the expression must be a positive number greater than zero.

EXAMPLE

```
1000 INPUT "An exponent please";Anex
1100 LET Answer1=EXP(Anex)
1200 PRINT "The natural log's value raised to the power ";Anex;"
is": PRINT Answer1: PRINT
1300 Answer2=LOG(Answer1)
1400 PRINT "The natural log of this number is ";Answer2: PRINT
1500 Answer3=LOG10(Answer1)
1600 PRINT "Its log to the base 10 is ";Answer3:PRINT
1700 GOTO 1000
1800 END
```

This example calculates the exponential of a number, its inverse (the LOG), and finally, the common logarithm (to the base 10).

MOD

The modulo operator.

FORMAT

dividend MOD divisor

NOTES

The modulo function (MOD) prints the remainder of a division operation, but not the quotient. This makes it the opposite of the integer division operation, which prints the quotient, but not the remainder. (See Integer Division earlier in this chapter.)

MOD rounds its operands to integers. It then performs floating point division and throws away the resulting quotient.

EXAMPLE

```
1000 INPUT "Divide 51 by what number"; Divisor
1100 LET Quotient=51\Divisor
1200 LET Remainder=51 MOD Divisor
1300 PRINT "The quotient is ";Quotient; " with a remainder of
";Remainder
1400 PRINT: GOTO 1000
1500 END
```

This example shows the integer division quotient and the MOD remainder that result from dividing 51 by your input divisor. The second example asks you for both the dividend and the divisor; it then calculates the results from floating point division, integer division, and MOD.

```
1000 INPUT; "Dividend";N
1100 INPUT " Divisor";D
1200 PRINT "FPDiv=";N/D,
1300 PRINT "IntDiv=";N\D,
1400 PRINT "MOD=";N MOD D
1500 PRINT: GOTO 1000
1600 END
```

NOT

The logical operator for negation.

FORMAT

NOT expression

NOTES

NOT is a unary operator that reverses the truth value of the operand (expression) it addresses. The NOT truth table (Table 5-3) below illustrates this.

A	NOT A
---	----
-1	0
0	-1

Table 5-3. The NOT Truth Table

EXAMPLE

```
1000 PRINT "Separate the two numbers with a comma":PRINT
1100 INPUT "Type two numbers between 1 and 5"; A,B
1200 IF NOT(A=3 AND B=4) THEN PRINT "Try again" ELSE PRINT "You
win"
1300 PRINT:GOTO 1100
1400 END
```

Compare this example to the example for AND. To get the same evaluation, the results ("Try again" and "You win" are reversed. This suits NOT's action on truth values. Also see the logic examples under "Logical Operators" in Chapter Two.

OR

The logical operator for disjunction.

FORMAT

expression1 OR expression2

NOTES

OR links two expressions and issues a true when both expressions evaluate as true or when just one evaluates as true. Both expressions must be false for OR to issue a false. See Table 5-4 below. Compare this action with XOR (Described at the end of this chapter), which yields a true only when just one of the two expressions is true.

A	B	A OR B
---	---	-----
-1	-1	-1
-1	0	-1
0	-1	-1
0	0	0

Table 5-4. The OR Truth Table

EXAMPLE

```
1000 PRINT "Separate the two numbers with a comma":PRINT
1100 INPUT "Type two numbers between 1 and 5"; A,B
1200 IF A=3 OR B=4 THEN PRINT "You win!" ELSE PRINT "Try again"
1300 PRINT:GOTO 1100
1400 END
```

You win if you type 3 as the first number of the pair, or if you type 4 as the second number, or if you type both correctly (3,4). Also see the logic examples under "Logical Operators" in Chapter Two.

PI

The pi constant.

FORMAT

PI

NOTES

PI is not a function, but the mathematical constant representing the ratio of the circumference to the diameter of a circle. GRiDBASIC keeps PI equal to

3.14159265358979

EXAMPLE

```
1000 PRINT "Note: Pi equals "; PI: PRINT
1100 INPUT "Enter the radius of a circle and confirm"; Radius:
PRINT
1200 Circ=2*PI*Radius
1300 PRINT "The circumference of the circle is ";Circ: PRINT
1400 Area=PI*Radius^2
1500 PRINT "The area of the circle is "; Area: PRINT
1600 GOTO 1100
```

This example puts the PI function to work in two common formulae, those for the circumference and area of a circle. It also prints the value of pi (see line 1000).

RANDOMIZE

RANDOMIZE seeds the RND number generator.

FORMAT

RANDOMIZE [expression]

NOTES

This statement gives the random number generator a specific seed to work with. RND takes each seed and from it creates a known series of numbers. Therefore, placing RANDOMIZE before a RND statement yields a repeatable series of numbers.

RANDOMIZE without an expression, sends the RND function back to the realtime clock for its seed. See the RND statement, described next, for further details on random numbers.

EXAMPLE

```
1000 RANDOMIZE 101
1100 INPUT "Loop times";Number
1200 PRINT: PRINT
1300 FOR X = 1 TO Number
1400 PRINT ,10*RND(1)
1500 NEXT X
1600 PRINT: PRINT
1700 GOTO 1000
1800 END
```

In this example, the expression "101" causes the same series of random numbers to print, no matter when or where you use it. Try other expressions. You can treat these expressions as if they were labels for certain series.

RND

The RND function returns a random number between 0 and 1.

FORMAT

RND(expression)

NOTES

The RND function can generate three types of series of random number each time you RUN a program, depending on the type of expression you give it. The three expressions and their products are:

- A number less than zero (-1). This expression reseeds the random number generator every tenth of a second from the realtime clock. Thus it has the effect of producing groups of two or three random numbers. See Figure 5-1.
- Zero. This takes the most recent number generated in the current series. If produced by a loop, the same number occurs repeatedly.
- A number greater than zero (+1). A sequence of random numbers.

Figure 5-1 below shows a typical run of the three types. You can find the program that generated these numbers in the Example section below.

```
When the argument < 0 ...  
0.0831158922713  
0.99060044251164  
0.99060044251164  
0.99060044251164  
0.89806973373007  
  
When the argument = 0 ...  
0.89806973373007  
0.89806973373007  
0.89806973373007  
0.89806973373007  
0.89806973373007  
  
When the argument > 0 ...  
0.67539482719158  
0.13185320820935  
0.74769207293812  
0.41618982223239  
0.81486228732738
```

Figure 5-1. Three Types of Random Numbers

NOTE: To create a repeatable series of random numbers, place the

RANDOMIZE statement (See above, this chapter) with the RND function.

To create a random whole number, simply multiply the RND function by some integer. The integer gives the uppermost value the function can return. Remember: RND returns numbers between 0 and 1. Ten times one equals ten, the largest number that line 1300 below permits. The program in Figure 5-2 returns the column of figures at its right.

1000 INPUT "Loop times";Number	1.00343327992676
1100 PRINT: PRINT	4.02151522087434
1200 FOR X = 1 TO Number	9.8295567254139
1300 PRINT, 10*RND(1)	7.92690928511482
1400 NEXT X	6.04562447547112
1500 PRINT: PRINT	1.51720454718853
1600 GOTO 1000	1.85885404745556
1700 END	6.45288776989395
	8.69520103761349
	5.48760204470893
	8.44892042420081
	8.7191577019913
	4.73334859235523
	9.83886472877088
	4.13061722743572
	2.38178072785534
	7.44151979858091

Figure 5-2. A Program and Series of Random Numbers

To turn a whole number into an integer, we recommend submitting the RND function to either the ROUND or TRUNC function. In particular, when you want a range extending from 1 to n, try

`TRUNC(expression)+1`

If you want a number in the range of 0 to n or in a range of numbers (n1 to n2), choose

`ROUND(expression)`

These two functions act differently to create an integer. ROUND rounds all decimals of .5 or greater upward. TRUNC, on the other hand, just cuts the decimal portion off. Table 5-5 below gives several examples you can use as models.

Range of Integers	Example Function
0 to 10	ROUND(10*RND(1))
1 to 10	ROUND(9*RND(1)+1) or TRUNC(10*RND(1)+1)
1 to 11	ROUND(10*RND(1)+1) or TRUNC(11*RND(1)+1)
87 to 95	ROUND(8*RND(1)+87)

Table 5-5. A Table of Integer Ranges and Functions

The Example section contains a program illustrating ranges 0 to 10 and 87 to 95. In the last range (87 to 95), we didn't multiply RND by 87, because that would produce all the numbers from 0 to 87. Instead, we multiplied by the width of the range (8) and added the beginning number of the range.

EXAMPLE

```

1000 PRINT: PRINT "When the argument < 0 ..."
1100 FOR X = 1 TO 5
1200 PRINT RND(-1)
1300 NEXT X
1400 PRINT: PRINT "When the argument = 0 ..."
1500 FOR Y = 1 TO 5
1600 PRINT RND(0)
1700 NEXT Y
1800 PRINT: PRINT "When the argument > 0 ..."
1900 FOR Z = 1 TO 5
2000 PRINT RND(1)
2100 NEXT Z

```

Running the above example shows the difference that RND's expression makes. Figure 5-1 shows a typical printout produced by this program. You can change the lengths of any of the loops to create larger or smaller sample sizes. The second example (see below) shows how you can achieve different ranges of integers by manipulating RND with TRUNC, ROUND, and additional numerals.


```

1000 REM This pgm creates RND integer ranges
1100 PRINT: PRINT "For the range 1 to 10 ..."
1200 FOR X = 1 TO 12
1300 PRINT TRUNC(10*RND(1))+1),
1400 NEXT X
1500 PRINT "For the range 0 to 10..."
1600 FOR Y = 1 TO 12
1700 PRINT ROUND(10*RND(1)),
1800 NEXT Y
1900 PRINT "For the range 87 to 95..."
2000 FOR Z = 1 TO 12
2100 PRINT (ROUND(8*RND(1))+87),
2200 NEXT Z

```

This program produces the output like the one in Figure 5-3.

For the range 1 to 10 ...			
6	3	2	2
3	4	8	9
7	4	8	5
For the range 0 to 10...			
9	8	2	9
0	9	10	8
6	7	7	6
For the range 87 to 95...			
88	94	92	90
94	87	93	95
93	91	88	88

Figure 5-3. Output of RND on Three Numeric Ranges

ROUND

The ROUND function.

FORMAT

ROUND (expression)

NOTES

The ROUND function takes a decimal number and converts it to an integer. If the decimal portion is .5 or greater the integer increases by one. If it is less, it drops to the next lower integer. Negative numbers are rounded (-3.5 becomes -4).

NOTE: This function is identical to the GRiDBASIC's CINT function described earlier in this chapter. The existence of both functions enhances the compatibility of GRiDBASIC with other BASIC's. See the section at the first of this chapter on Integer Functions.

EXAMPLE

```
1000 INPUT "Enter any number and confirm", Decimal
1100 Answer=ROUND(Decimal)
1200 PRINT "The ROUND integer is ";Answer :PRINT
1300 GOTO 1000
```

SGN

The sign function.

FORMAT

SGN(expression)

NOTES

This function returns the algebraic sign of an expression. A positive expression returns 1, negative expressions return -1, and zero returns 0.

EXAMPLE

```
1000 PRINT
1100 INPUT; "The sign of"; Number
1200 ON SGN(Number)+2 GOTO 1300, 1400, 1500
1300 PRINT " is minus (-)": GOTO 1000
1400 PRINT " is zero (no sign)": GOTO 1000
1500 PRINT " is plus (+)": GOTO 1000
1600 END
```

This example tests for sign of number given it. The SGN function returns the appropriate number. The "+2" raises this number to a 1, 2, or 3 -- all numbers that the ON GOTO statement can use. The result points to the correct answer line.

SIN

The sine function.

FORMAT

SIN(expression)

NOTES

This function takes an expression representing an angle in radians and returns the sine of that angle. GRiDBASIC always evaluates this expression in full precision. To convert from degrees to radians, multiply by $\pi/180$.

EXAMPLE

```
1000 INPUT "Enter angle (in degrees) and confirm",Angle
1100 Rads=Angle*(PI/180)
1200 Calculation=SIN(Rads)
1300 PRINT "The sine of ";Angle;" degrees is "; Calculation:
PRINT
1400 GOTO 1000
```

SQR

The square root function.

FORMAT

SQR(expression)

NOTES

This function returns the square root of an expression. The value of the expression must be zero or greater.

EXAMPLE

```
1000 INPUT; "Square root of"; Number
1100 PRINT " is "; SQR(Number)
1200 PRINT: PRINT
1300 GOTO 1000
1400 END
```

TAN

The tangent function.

FORMAT

TAN(expression)

NOTES

This function takes an expression representing an angle in radians and returns the tangent of that angle. GRiDBASIC always evaluates this expression in full precision. To convert from degrees to radians, multiply by $\pi/180$.

EXAMPLE

```
1000 INPUT "Enter angle (in degrees) and confirm",Angle
1100 Rads=Angle*(PI/180)
1200 Calculation=TAN(Rads)
1300 PRINT "The tangent of ";Angle;" degrees is "; Calculation:
PRINT
1400 GOTO 1000
```

TRUE

The Boolean constant for true.

FORMAT

TRUE

NOTES

The constant TRUE has a value of -1. Statements can interact with it in a number of ways. You can assign its value to variables, operate it on it logically, and print it. The program below does all these things.

EXAMPLE

```
1000 PRINT "True=";TRUE; " and False=";FALSE
1100 INPUT "Type the number 3"; A
1200 IF A=3 THEN B=TRUE ELSE B=NOT TRUE
1300 PRINT B;
1400 IF B=FALSE THEN PRINT " means you didn't type 3" ELSE PRINT
" means you typed 3"
1500 PRINT:GOTO 1100
1600 END
```

Line 1000 prints the values of GRiDBASIC's two Boolean constants. Whenever you use TRUE or FALSE, you use the constant's value. For example, depending on the value of A, line 1200 does one of two things. It either assigns -1 to B (TRUE) or applies NOT to TRUE, changing the -1 to its opposite, a zero (0).

TRUNC

The truncate function.

FORMAT

TRUNC (expression)

NOTES

The TRUNC function converts a number (whether positive or negative) into an integer not by rounding it, but by chopping off anything to the right of the decimal point. TRUNC acts like another integer function, FIX. See the article on integer functions at the first of this chapter. Also compare TRUNC to the FIX and ROUND functions.

EXAMPLE

```
1000 INPUT "Enter any number and confirm", Decimal
1100 Answer=TRUNC(Decimal)
1200 PRINT "The TRUNC integer is ";Answer :PRINT
1300 GOTO 1000
```


XOR

The exclusive-OR logical operator.

FORMAT

expression1 XOR expression2

NOTES

XOR yields a true if just one just one of the expressions evaluates as true, but not if both or neither are true. Table 5-6 shows this.

A	B	A XOR B
---	---	-----
-1	-1	0
-1	0	-1
0	-1	-1
0	0	0

Table 5-6. The XOR Truth Table

EXAMPLE

```
1000 PRINT "Separate the two numbers with a comma":PRINT
1100 INPUT "Type two numbers between 1 and 5"; A,B
1200 IF A=3 XOR B=4 THEN PRINT "You win!" ELSE PRINT "Try again"
1300 PRINT:GOTO 1100
1400 END
```

With XOR you can only win by getting just one of the pair of numbers correct -- either the 3 in the first place or the 4 in the second. If you type "3,4" the program tells you to "Try again."



CHAPTER SIX: STRING FUNCTIONS

This chapter describes GRiDBASIC's string functions. String functions perform operations on sequences of characters specified in programs. A string is any sequence of characters. All of these functions require an input parameter or argument enclosed in parentheses.

A word on nomenclature. A number of the string function names end with the dollar sign (\$). Most programmers "pronounce" this symbol in either of two ways. Some say "dollar"; others say "string." Thus the statement LEFT\$ is called both "left dollar" and "left string." Take your pick.

ASC

The ASCII function.

FORMAT

ASC(string\$)

NOTES

ASC takes the first character of string\$ and returns that character's ASCII code (a decimal, numeric value). This is the inverse of the CHR\$ function, which converts an ASCII code to a character (see below). "ASCII" stands for "American Standard Code for Information Exchange."

If the string has a length of zero (no characters in the string), an error occurs.

EXAMPLE

```
1000 INPUT "Press a key and confirm",Text$
1100 LET Code=ASC(Text$)
1200 LET Letter$=CHR$(Code)
1300 PRINT "The ASCII code for ";Letter$;" is ";Code
1400 PRINT
1500 GOTO 1000
1600 END
```

This example converts text (including numbers, and punctuation, and other characters into their ASCII codes. Note that though the input variable (Text\$) is a string variable, ASC returns a numeric value, because each ASCII code is a number.

CHR\$

The character string function.

FORMAT

CHR\$(expression)

NOTES

This function converts an expression representing an ASCII code (in decimal) to its one character equivalent. The expression must be a value in range of 0 to 255. This function is the inverse of the ASC function, which performs ASCII-to-numeric conversion.

EXAMPLE

```
1000 INPUT "Enter an ASCII code and confirm", Ascode
1100 LET Letter$ = CHR$(Ascode)
1200 PRINT Ascode;" is the ASCII code for ";Letter$
1300 PRINT
1400 GOTO 1000
1500 END
```

This program takes any ASCII code (in decimal) from 0 to 255 and prints the character represented by the code. Note that line 1100 assigns the resulting character to a string variable, Letter\$ (whether or not it's a number).

INSTR

The in string function.

FORMAT

INSTR([expression],sourceString\$,findString\$)

NOTES

The INSTR (often called "in string") function locates a specified string (findString\$) within another string (sourceString\$) and returns the character position of the first occurrence of the string. INSTR differentiates between upper and lower case; specify characters accordingly.

The optional expression tells the function how many characters to skip (from the left) before beginning its search. Include this expression when you want to move past the string just located to find another occurrence of the same string.

INSTR returns a zero (0) when:

- The value of expression is greater than the length of sourceString\$
- SourceString\$ is null
- It cannot find findString\$.

If findString\$ is null, INSTR returns 1 or expression (if included).

EXAMPLE

```
1000 LET Sample$="The dollar the snowman the Cat"
1100 LET A$="he": Let B$="the": Let C$="man": LET D$="doll": LET
E$="cat": LET F=6
1200 LET Position1=INSTR(13,Sample$,B$)
1300 LET Position2=INSTR("weather",B$)
1400 LET POSITION3=INSTR(F,"Woebegone","e")
1500 LET POSITION4=INSTR(Sample$,"now")
1600 PRINT Position1
1700 PRINT Position2
1800 PRINT Position3
1900 PRINT Position4
2000 END
```

This program yields four numbers:

24
4
9
17

The example illustrates two facts. First, `expression`, `findstring$`, and `sourcestring$` can occur as variables and/or values (whether string or numeric) in the same specification. Second, `expression` views the number of characters in `sourcestring$` as absolute.

For example, the expression in line 1400 tells `INSTR` to position itself at the "g" in "Woebegone" and search for "e" (one character past the second "e"). In this case, it returns 9 -- the position of the last "e" -- not 3, which it would if it started counting at one from each position.

Note too, that if you searched for `E$ (cat)` within `Sample$`, `INSTR` would return a zero. The reason: The "Cat" within `Sample$` has an uppercase "C."

LEFT\$

The left string function.

FORMAT

LEFT\$(string\$,expression)

NOTES

This function returns the leftmost character(s) from a specified string. The function counts in from the left end of the string by the number of characters specified in the expression. For example,

LEFT\$(Compass Computer system,7)

yields the string "Compass."

If the value of expression is greater than the length of the string, the entire string is returned. If the value of expression is zero, a null string (no characters) is returned.

EXAMPLE

```
1000 LET Sample$="dollar toy pizza book tree home"
1100 PRINT "The string is ";Sample$;" "
1200 PRINT
1300 INPUT "Take how many letters from the left"; Number
1400 LET Someletter$=LEFT$(Sample$,Number)
1500 PRINT "LEFT$(Sample$,";Number;" ) is "; Someletter$;" "
1600 GOTO 1200
```


LEN

The length function.

LEN returns the number of characters in a specified string and thereby its length. All characters in the string, including signs, decimal points, blanks, and non-printable characters, are counted.

FORMAT

LEN(string\$)

EXAMPLE

```
1000 INPUT "Type some characters and confirm"; Stuff$
1100 PRINT "You entered "; LEN(Stuff$); " characters that time."
1200 PRINT
1300 GOTO 1000
1400 END
```

This example shows that the LEN function counts the number of characters in a string. (Also see the example for the STR\$ function.)

MID\$

The mid string function.

FORMAT

MID\$(string\$,I[,J])

NOTES

The MID\$ function returns a specified portion of a string. The parameter I specifies the first character (counting from the left end of the string) that MID\$ returns. The optional parameter J specifies the total number of characters the function should return. For example,

```
MID$(Compass Computer system,9,8)
```

yields the string, "Computer".

If J is omitted, or if there are fewer than J characters to the right of the Ith character, all characters from I to the right end of the string will be returned. If I is greater than the length of the string or if J is zero, MID\$ returns a null string, that is, a string with no characters in it.

EXAMPLE

```
1000 LET Sample$="dollar toy pizza book tree home"
1100 PRINT "The string is ";Sample$;" "
1200 PRINT
1300 INPUT "Go how far in from the left"; Number
1400 INPUT "And take how many letters";Letters
1500 LET Someletter$=MID$(Sample$,Number,Letters)
1600 PRINT "MID$(Sample$,";Number; ", "; Letters;") is ";
Someletter$;" "
1700 GOTO 1200
1800 END
```

RIGHT\$

The right string function.

FORMAT

RIGHT\$(string\$,expression)

NOTES

This function counts from the right end of a string of characters to return a number of characters. The expression returns the number of characters specified by expression. If the value of expression is greater than the length of the string, the entire string is returned. If the value of expression is zero, a null string (no characters) is returned.

EXAMPLE

```
1000 LET Sample$="dollar toy pizza book tree home"
1100 PRINT "The string is ";Sample$;" "
1200 PRINT
1300 INPUT "Take how may letters from the right"; Number
1400 LET Someletter$=RIGHT$(Sample$,Number)
1500 PRINT "RIGHT$(Sample$,";Number;" ) is "; Someletter$;" "
1600 GOTO 1200: REM if you're going to do this a lot, GOTO 1100
instead
```

SPACE\$

The space string function.

FORMAT

SPACE\$(expression)

NOTES

The SPACE\$ function returns a string consisting of spaces. The expression specifies the number of spaces.

EXAMPLE

```
1000 INPUT "How many spaces"; Number
1100 LET Blank$=SPACE$(Number)
1200 PRINT Number; " spaces lie between the asterisks": PRINT
    "*";Blank$;"*"
1300 PRINT
1400 GOTO 1000
1500 END
```

STR\$

The S-T-R string function.

FORMAT

STR\$(expression)

NOTES

The STR\$ function converts the value of a numeric expression into a string, so that you can perform strings (rather than numeric) operations on it.

EXAMPLE

```
1000 INPUT "Type a number"; Number1
1100 INPUT "And another to multiply it by"; Number2
1200 LET C$=STR$(Number1*Number2)
1300 PRINT "The answer is ";C$;". Length of this string is ";LEN(C$)
1400 PRINT
1500 GOTO 1000
1600 END
```

This example takes two numbers and converts their product to a string (line 1200). The fact that LEN, a string function operates on product, proves this is a string, not a numeric, constant (line 1300).

STRING\$

The string function.

FORMAT

```
STRING$(expression, ASCIIcode)  
STRING$(expression, string$)
```

NOTES

This function returns a string whose characters all have the same ASCII code. The value of expression defines the length of the string.

You specify the character returned by giving its ASCII code (in decimal) or by giving a string. STRING\$ returns only the first character of this string.

EXAMPLE

```
1000 CodeSample$=STRING$(10,42)  
1100 PRINT "The string using an ASCII code is ";CodeSample$  
1200 PRINT  
1300 LET A$="Hello"  
1400 FirstChar$=STRING$(10,A$)  
1500 PRINT "The sample taking the first character is "; FirstChar$  
1600 END
```

The example shows the STRING\$ function with both arguments. Line 1000 takes the ASCII argument and prints 10 asterisks in line 1100. Line 1400 takes the first character of A\$ (Hello) and prints it 10 times in line 1500. NOTE: the 10 in both print statements is the first argument in each STRING\$ definition.

VAL

The value function.

FORMAT

VAL(string\$)

NOTES

This function returns the numeric value of a specified string. The string should comprise nothing other than leading blank(s), a sign, and a number (the blank(s) and sign needn't be present).

VAL strips off any leading blanks from the string. If the first non-blank character is anything except a plus sign (+), minus sign (-), or a numeric digit, VAL returns a zero (0). If the string contains anything besides numeric digits, it also returns a zero (0).

EXAMPLE

```
1000 INPUT "Type a number"; Number1
1100 INPUT "And another to multiply it by"; Number2
1200 LET C$=STR$(Number1*Number2)
1300 PRINT "The answer is ";C$;". Length of this string is ";LEN(C$)
1400 LET Result=VAL(C$)/Number1
1500 PRINT "Dividing by the first yields the second: ";Result
1600 PRINT
1700 GOTO 1000
1800 END
```

This example turns a number into string (line 1200), and engages VAL in line 1400 to turn the string number back into a number that numeric operators can handle. NOTE: VAL's counterpart is STR\$ (see line 1200).



CHAPTER SEVEN: INPUT/OUTPUT STATEMENTS

The input/output statements discussed in this chapter transfer data to and from memory, the realtime clock, the keyboard, and the screen. For information on sequential file I/O, see Chapter Eight. For random access file I/O, see Chapter Nine.

COMMA

The comma character (,) formats output to the screen.

FORMAT

expression, expression[,]

NOTES

Whether in an INPUT statement or a PRINT statement, the comma simultaneously links elements in a series and keeps them separate. The comma differs from the semicolon in that it causes each element to print at predetermined tab position. The comma places each expression in one of four absolute fields -- at columns 0, 15, 30, and 45.

Within a PRINT statement, a comma following the last element in a list causes suppression of the carriage return and line feed characters that the PRINT statement normally issues after its expression(s). Instead, the expressions print at the appropriate tab field.

Placing the comma before the first expression in a PRINT statement causes the expression to print at the second field. Likewise, two commas preceding an expression cause printing at the third field, and so on. For example:

```
1800 PRINT ,,"Third tab"
```

Placing the comma between an INPUT string and its variable, suppresses the question mark (?) normally issued by the INPUT statement. For example:

```
1500 INPUT "Your name please", Name$
```

You can request multiple items with an INPUT statement, if you separate the statement's variables with commas. For example,

```
1600 INPUT "Please enter three numbers", A, B, C
```

NOTE: The response to this must also separate each item with a comma. For example,

```
54, 98.01, 1
```

When sending data to the Epson printer, you must supply tab position information for the comma to work correctly. Otherwise, you won't get the spaces between columns that you expect.

You must follow the PRINT# command with the file tag number, an ESC D (represented by CHR\$(27)+"D") and the column number of each tab preceded by the CHR\$ statement. Concatenate these tab positions with the plus sign (+). All such statements must end with the null character, CHR\$(0). Do NOT exceed an 80-character line. An example command assigning 15 character-wide tabs follows:

```
PRINT# 1, CHR$(27)+"D"+CHR$(15)+CHR$(30)+...+CHR$(0)
```

EXAMPLE

```
1000 INPUT "Your name please:", Name$
1100 INPUT "Three numbers", A, B, C
1200 PRINT
1300 PRINT "Hello there", Name$, "3", "Albert"
1400 PRINT "A very long string", "of",
1500 PRINT A, B, C
1600 PRINT ,,"Third tab"
1700 END
```

This example illustrates what commas can do in both INPUT and PRINT statements. The comma in Line 1000 suppresses INPUT's question mark. In line 1100, commas separate variables for INPUT.

Line 1300 shows the tab zones set up by the comma. Note in Figure 7-1 below that when a string exceeds the 15-character width set up by the comma that the next string appears in the next zone over. The first string does not collide with the second.

The comma at the end of line 1400 suppresses the carriage return-line feed at the end of that line, so that line 1400 and 1500's tab zones become continuous. The two commas before the expression in line 1600 push the expression one tab each so that the string "Third tab" prints at the third tab.

Your name please: John			
Three numbers 8,-912765,.0001243			
Hello there	John	3	Albert
A very long string		of	8
-912765	0.0001243		
		Third tab	

Figure 7-1. Examples of Comma Formatting

DATE\$

The date function.

FORMAT

DATE\$

NOTES

DATE\$ returns the current date from the Compass Computer system's real-time clock. The date is an eight character string in the form mm/dd/yy where mm is the month (00 to 12), dd is the day of the month (00 through 31) and yy is the year (00 through 99). NOTE: These characters are string, not numeric characters. For the program to use them numerically, you must convert them to numbers (see Chapter Six, the VAL statement and the example below).

EXAMPLE

```
1000 PRINT "The date is "; DATE$
1100 LET Month$ = LEFT$(DATE$,2)
1200 IF LEFT$(Month$,1)="0" THEN LET Month$=RIGHT$(Month$,1)
1300 PRINT "The number of the month is "; Month$
1400 LET MONTH=VAL(Month$): LET A$="The name of the month is "
1500 ON Month GOTO
1600,1700,1800,1900,2000,2100,2200,2300,2400,2500,2600,2700
1600 PRINT A$;"January":END
1700 PRINT A$;"February":END
1800 PRINT A$;"March":END
1900 PRINT A$;"April":END
2000 PRINT A$;"May":END
2100 PRINT A$;"June":END
2200 PRINT A$;"July":END
2300 PRINT A$;"August":END
2400 PRINT A$;"September":END
2500 PRINT A$;"October":END
2600 PRINT A$;"November":END
2700 PRINT A$;"December":END
```

This example prints the current date in line 1000. It then removes the "0" from the front of all single digit month numbers and prints the number of the month (lines 1100-1300). The rest of the example uses the ON GOTO statement so that the month's number can cause the month's name to print.

To do this, we convert the month numeral-as-string character to a numeral with the VAL statement (see Chapter 6). You can incorporate this program as a subroutine where you want a nicely formatted date.

INKEY\$

The inkey function.

FORMAT

INKEY\$

NOTES

INKEY\$ reads the keyboard and returns whatever value it finds there. If you press a key at the moment that INKEY\$ reads the keyboard, the function returns that key's value as a one-character string. It does not display that character on the screen. Instead, it passes the string to your program. When INKEY\$ reads the keyboard and finds nothing, it returns a null string (length zero).

NOTE: INKEY\$ does not of itself wait for a keypress to occur. If you want to monitor the keyboard continuously, you must put INKEY\$ in a loop (see example below).

EXAMPLE

```
1000 LET Loop=0
1100 LET Key$=INKEY$
1200 IF Key$="" THEN Loop = Loop+1:PRINT " "; Loop; " "; ELSE PRINT
" *** "; Key$; " *** ";
1300 GOTO 1100
1400 END
```

This example prints a sequential number on the screen each time INKEY\$ reads the keyboard. When you press a key, it prints the key surrounded on either side by " *** ".

INPUT

The INPUT statement asks the user to enter data. It then assigns the data to specified variables.

FORMAT

```
INPUT [;J["promptString"] {;!,} variablesList
```

NOTES

When an INPUT statement executes, it prints the contents of the promptString. If you follow the promptString with a semicolon, a question mark will follow this string. For example,

```
1000 INPUT "Your name"; Name$
```

prints on the screen as

```
Your name?
```

If you put a comma at the end of the string no question mark appears. If you do not include a promptString, the program only displays the question mark. You must enclose the promptString in quotation marks; it can contain any printable characters.

Program execution stops after displaying promptString and question mark (if specified). Execution waits for you to enter data and press CODE-RETURN. If you place a semicolon directly after the word INPUT, the cursor will remain on the same line as the user's response after confirming.

Multiple variables must appear at the end of the INPUT statement. You cannot place variables within the input string. The following example places a variable (Count) in the input string to describe a range of choices. This is an illegal statement.

```
1500 INPUT "Pick a number (from 1 to ";Count; ") "; Choice
```

The INPUT statement wants to put your data into Count, because Count comes at the end of a prompt string. To put such an informational variable in an INPUT statement, write two lines, one a PRINT statement, the other an INPUT statement. In the example below, we break the illegal line 1500 into two lines. The semicolon at the end of line 1500 causes the two to print like one statement.

```
1500 PRINT "Pick a number from 1 to ";Count;  
1600 INPUT Choice
```

Data entered via the keyboard is assigned to the variable(s)

specified in the variablesList. The number of data items entered must be equal to the number of variables specified in variablesList. You must separate multiple variables in variablesList with commas.

Each data item entered must be of the same type as that specified by the corresponding variable name. The variable names in variablesList can be any mix of numeric and string variable names including subscripted variables. However, each input must be of the same kind as its variable.

NOTE: INPUT does not accept a comma or a semicolon as valid input. You must start your string with the double quotation mark (") if you want to include either of these characters.

If you respond with the wrong kind of constant (giving letters to a numeric variable or including a comma or semicolon in an input string, for example), you will see the message

Invalid input: Re-enter data

EXAMPLE

```
1000 INPUT "Please enter your first name", First$
1100 PRINT "Okay, "; First$;
1200 INPUT ", what is your last name"; Last$
1300 INPUT; "Your area code"; Area$
1400 INPUT " And phone number"; Phone$
1500 PRINT "We can reach you at ("; Area$; ")"; " "; Phone$
1600 PRINT
1700 PRINT "Type three numbers...": INPUT "(Put a comma between each
one)", A, B, C
1800 PRINT: PRINT "Those numbers are: "; A, B, C
1900 END
```

This example illustrates the various possibilities inherent in the INPUT statement. In line 1000, the comma at the end of promptString suppresses a question mark, whereas the semicolon at the end of promptString in line 1200 prints a question mark. However, in line 1300 the semicolon following INPUT suppresses the carriage return-line feed character at the end of the line. As a result, lines 1300 and 1400 print on the same line. See Figure 7-2 below.

Lines 1100 and 1200 combine a variable that gives information and one that asks for input. Finally, lines 1700 and 1800 show how to gather multiple items of information with one INPUT statement.

```
Please enter your first name John
Okay, John, what is your last name? Smith
Your area code? 415 And phone number? 961-4800
We can reach you at (415) 961-4800

Type three numbers...
(Put a comma between each one) 12.06,-3.14,+.00001

Those numbers are: 12.06      -3.14      0.00001
```

Figure 7-2. The INPUT Statement Illustrated

LOCATE

This statement positions the cursor to a specified dot or pixel location on the screen.

FORMAT

LOCATE x,y

NOTES

The horizontal coordinate (x) must be in the range of 1 to 320 and the vertical coordinate must be in the range of 1 to 240. The coordinates describe the position of the top, left pixel of the first character in the string that follows the LOCATE statement.

When a program runs, it doesn't normally display the cursor. When you follow a LOCATE statement with an input/output statement such as INPUT, the cursor appears at the screen location specified by the last preceding LOCATE statement. Similarly, a subsequent PRINT statement will output its data beginning at the previously specified screen location.

Also see the DRAWCHARS statement in Chapter Ten. DRAWCHARS does not position the cursor, but rather specific character strings.

EXAMPLE

```
1000 LOCATE 140,110
1100 INPUT "Horizontal axis (0-320)", Horiz
1200 LOCATE 140,120
1300 INPUT "Vertical axis (0-240)", Vert
1400 LOCATE Horiz, Vert
1500 PRINT "."
1600 END
```

This example shows the power of the LOCATE statement by positioning its INPUT statements (lines 1000 and 1200) and then by letting you display a dot at your own coordinates (line 1400).

PRINT

The PRINT statement displays data on the screen.

FORMAT

PRINT [expression][[,!;]][expression] ... [[,!;]]

NOTES

The PRINT statement displays any expression that follows it and sends a carriage return-line feed combination at the end of that expression. When deprived of an expression, PRINT displays a blank line, the result of the carriage return-line feed characters. The following expressions are all legal. Line 1100 yields the product of 5×6 , 30.

```
1000 PRINT "Hello"  
1100 PRINT 5*6  
1200 PRINT
```

You must enclose string constants in quotation marks ("). You can omit the final quotation mark from any string appearing at the end of a program line. Only the size of the screen limits the number of expressions a single PRINT statement can handle.

To place multiple expressions after a single PRINT statement, you must separate the individual expressions with either a comma (,) or a semicolon (;).

If you place a semicolon between two expressions, the two expressions will print with no intervening characters. See "SEMICOLON" later in this chapter.

If you place a comma between two expressions, PRINT displays the value of the second expression at the beginning of the next "print zone". GRI DBASIC divides each line into print zones of 15 spaces each. Commas used as expression separators cause a "tabbing" effect so that the next expression value is displayed in the next print zone.

The zones begin at columns 0, 15, 30, and 45. If a string has more than 15 characters, PRINT will skip the zone that has been overwritten and begin the next display at the next zone. Thus the comma never causes concatenation. See "COMMA" earlier in this chapter.

Terminating a list of expressions with a comma or semicolon, cancels the carriage return-line feed pair so that a subsequent PRINT statement continues printing on the same line. If a printed line is

longer than the display's line width, printing continues on the next line. GRiDBASIC breaks strings at the right edge of the screen.

Printed numbers are always followed by one space and positive numbers are also preceded by one space. A minus sign precedes each negative number.

EXAMPLE

```
1000 LET A=5: B=3: C$="George": D$="Washington"
1100 PRINT A
1200 PRINT B
1300 PRINT "A+B=";A+B
1400 PRINT 5+3
1500 PRINT
1600 PRINT C$
1700 PRINT D$
1800 PRINT C$+D$
1900 PRINT
2000 PRINT TAB(0) "0"; TAB(10) "10"; TAB(20) "20"; TAB(30) "30";
    TAB(40) "40"; TAB(50) "50"
2100 PRINT "A", "B", "C", "D", "E", "F", "G"
2200 PRINT "A"; "B"; "C"; "D"; "E"; "F"; "G"
2300 END
```

Lines 1100 and 1200 print the values stored at variables A and B. Line 1300 prints a string constant and then the result of adding A and B.

Line 1400 shows that PRINT can operate on numeric constants by doing math for you. Note in line 1800, a plus sign between string constants concatenates (or joins together) strings.

Line 2000 shows how the TAB statement operates with PRINT. Line 2100 the comma's tabbing effect and line 2200 the semicolon's concatenating effect.

PRINT USING

The PRINT USING statement formats strings or numbers, depending on the punctuation that follows the statement.

FORMAT

PRINT USING format symbol;{list of expression|list of string\$}

NOTES

PRINT USING takes as its arguments a format symbol and a list either of numeric or string expressions. The format symbol shapes the expression into their format.

For example, the format symbol ("###.##")

```
"###.##";A
```

tells GRiDBASIC to put the number stored in variable A into a format with three digits to the left of the decimal point and two digits to the right. Thus the number 34.14735 appears in the formatted form

```
34.15
```

This section explores each format symbol and its results.

STRING EXPRESSIONS

You can modify string expressions with any one of three format symbols:

- The exclamation point (!)
- Double back slash enclosing space(s) (\n space\)
- The ampersand (&)

EXCLAMATION POINT (!)

The exclamation point returns only the first character in each string argument that follows it. See the example below.

DOUBLE BACK SLASH (\)

When you don't put space(s) between the two back slash (CODE-SHIFT-') characters, the double back slash prints two

characters from its string argument(s). Each space between the back slashes causes another character from the string(s) to print.

Double back slash prints one space character for each character you specify over the number of characters in the string. Thus if a string expression has five characters, and five space characters separate the two back slashes, two spaces will follow the printing of the five character string. See the example below with its companion printout.

AMPERSAND (&)

The ampersand causes the string to print exactly as it is stored. See the example below.

EXAMPLE (STRINGS)

```
1000 LET A$="Input"
1100 LET B$="Output"
1200 PRINT USING "!";A$;B$
1300 PRINT USING "\\";A$;B$
1400 PRINT USING "\ ";A$;B$
1500 PRINT USING "\  ";A$;B$
1600 PRINT USING "\   ";A$;B$
1700 PRINT USING "\    ";A$;B$
1800 PRINT USING "&";A$;
1900 END
```

```
IO
InOu
InpOut
InpuOutp
InputOutpu
Input Output
Input
```

NUMERIC EXPRESSIONS

The numeric format symbols include:

- The number or "pound" sign (#)
- The decimal point (.) and comma (,)
- The plus (+) and minus (-) signs
- The double asterisk (**)
- The double dollar (\$\$)
- The double asterisk-dollar (**\$)
- The character string

NUMBER SIGN (#)

Each number sign reserves one digit of space for PRINT USING. Thus to reserve space for a five digit number followed by three decimal places, you write

```
#####.###
```

Such a format handles numbers like

```
12345.567 and -2345.987
```

Note that the minus sign takes one of the character positions. If you try to print a number with more digits than your format allows, a percent sign (%) will precede the first character (whether it's a sign or a number). We call this the "overflow symbol." See Figure 7-3 below. Thus trying to put the number -99999.01 in the format #####.## would result in

```
%-99999.01
```

Whenever a number has fewer digits than the PRINT USING format, PRINT USING puts these extra spaces at the front of the number. Figure 7-3, shows a five digit format (to the left of the decimal) and three numbers in that format. Both the five digit positive number and the four digit negative number take up all allotted digits. The extra two digits pad the three digit number to its left.

```
#####.##  
23468.91  
    576.08  
-3418.99
```

Figure 7-3. How Format Characters Pad Digits

COMMA (,)

If you want your number to display a comma every three digits, you can include the comma anywhere to the left of the decimal point. The comma also specifies another digit in the string. The following examples are all legal.

```
#####,.##    ##,#####.##    ,#####.##
```

If you place a comma to the right of the decimal point, the comma prints as a literal at the end of the number. For example:

2345.56,

You can pad numeric output with surrounding spaces by putting space characters between either end of the string and the nearest quotation mark. See Figure 7.4 below for an example.

The program below with its output illustrate these facts.

```
1000 LET A=.912345
1100 LET B=7
1200 LET C=-1234.567891
1300 PRINT "A = ";A
1400 PRINT "B = ";B
1500 PRINT "C = ";C
1600 PRINT
1700 PRINT USING "#####.##";A, B, C
1800 PRINT USING "#####.##  ";A, B, C
1900 PRINT USING "#####.##";A, B, C
2000 PRINT USING "##.##";A, B, C
2100 END
```

A = 0.912345		
B = 7		
C = -1234.567891		
0.91	7.00	-1234.57
0.91	7.00	-1234.57
0.91	7.00	-1,234.57
0.91	7.00	%-1234.57

Figure 7-4 Basic formatting for PRINT USING

PLUS (+) AND MINUS (-) SIGNS

GRiDBASIC accepts a format with a plus or a minus sign at either the front or rear of the format string. All the following are legal:

```

+####.##
-####.##
####.##+
####.##-

```

Placing the plus sign on either end of the format string causes positive numbers to display the plus sign in the position indicated by the format. Negative numbers print the minus sign in this same position. Either sign adds an extra space to its number. See Figure 7-5 below, it shows a program with formatted output.

```

1000 LET A=-213.14
1100 LET B=2130.14
1200 LET C=-2130.14
1300 PRINT "A = ";A
1400 PRINT "B = ";B
1500 PRINT "C = ";C
1600 PRINT
1700 PRINT USING "#####.##";A, B, C
1800 PRINT USING "+#####.##";A, B, C
1900 PRINT USING "#####.##+";A, B, C
2000 PRINT USING "#####.##-";A, B, C
2100 PRINT USING "-#####.##";A, B, C
2200 END

```

A = -213.14		
B = 2130.14		
C = -2130.14		
-213.14	2130.14	-2130.14
-213.14	+2130.14	-2130.14
213.14-	2130.14+	2130.14-
213.14-	2130.14	2130.14-
-213.14	2130.14	%-2130.14

Figure 7-5. The PRINT USING Format with Signs

DOUBLE ASTERISKS (**)

Placing two asterisks in front of the format string fills leading spaces with asterisks. NOTE: Leading spaces appear when the number of digits take less space than the number of positions specified by the format. The program and printout in Figure 7-6 illustrate double asterisk, double dollar, double asterisk-dollar formatting.

DOUBLE DOLLAR (\$\$)

Placing two dollar signs before a format string causes a dollar sign to print to the left of the formatted number. Double dollar creates two format spaces, one of which the printed dollar sign takes. See Figure 7-6 for examples.

DOUBLE ASTERISK-DOLLAR (**\$)

Double asterisk-dollar combines the effects of double asterisk and double dollar: It prints a dollar sign to the left of the number and fills the field with asterisks whenever the number contains fewer digits than the format specifies. See Figure 7-6 immediately below.

EXAMPLE (NUMBERS)

```
1000 LET A=.912345
1100 LET B=7
1200 LET C=-1234.567891
1300 PRINT "A = ";A
1400 PRINT "B = ";B
1500 PRINT "C = ";C
1600 PRINT
1700 PRINT USING "$$#####.##";A, B, C
1800 PRINT USING "$$#####.##-";A, B, C
1900 PRINT USING "**$#####.##";A, B, C
2000 PRINT USING "**$#####.##-";A, B, C
2100 PRINT USING "*****.##";A, B, C
2200 END
```

A = 0.912345		
B = 7		
C = -1234.567891		
\$0.91	\$7.00	\$-1234.57
\$0.91	\$7.00	\$1234.57-
*****\$0.91	*****\$7.00	*\$-1234.57
*****\$0.91	*****\$7.00	*\$1234.57-
*****0.91	*****7.00	*-1234.57

Figure 7-6. Asterisk and Dollar Formatting

CHARACTER STRINGS

You can also include character strings between either set of

quotation marks and the format string. Remember: The space is a character. All the following are legal:

```
" #####.##"  
"#####.##      "  
"   #####.##    "  
"Your account contains $#####.##"  
"#####.## after deductions"  
"  You get #####.## shares for each hundred"
```

SEMICOLON

The semicolon character formats PRINT and INPUT data.

FORMAT

expression; expression[;]

NOTES

The semicolon character (;) serves to link expressions following PRINT and INPUT statements. Placed between expressions, the semicolon can link variables and strings. Unlike the comma, it provides no space between expressions. Placed at the end of a program line, the semicolon suppresses the carriage return-line feed characters issued by PRINT.

When a semicolon follows an INPUT statement, it suppresses the carriage return-line feed pair. As a result, you can request multiple items for INPUT on the same line.

EXAMPLE

```
1000 INPUT "Your first name is"; Name$
1100 PRINT
1200 INPUT; "Your City"; City$
1300 INPUT; " State"; St$
1400 INPUT " ZIP"; ZIP$
1500 PRINT
1600 PRINT "Ahhh, you mean "; City$; ", "; St$; " "; ZIP$; " and not
";
1700 PRINT Name$;City$;St$;ZIP$
1800 END
```

In lines 1000, 1200, 1300, and 1400 the semicolons following the prompt string cause a question mark to print immediately after the prompt. The semicolons after INPUT on lines 1200 and 1300 suppress the carriage return-line feed pair, so that the program requests city, state, and ZIP code information all on the same line.

Line 1600 shows how you can place semicolons to link a mix of strings and variables. Line 1700 prints as one long line, because semicolons provide no spacing.

TAB

This function operates with the PRINT statement to tab horizontally a specified number of character positions or spaces.

FORMAT

TAB(expression)

NOTES

TAB understands expression as the column number where it should position whatever item follows it. For example,

```
1000 PRINT TAB(17) "Top Drawer"
```

prints the string "Top Drawer" at the 18th column of the current line. NOTE: The first display position on a line is 0; TAB(1) is the second character position on a line.

The expression must be a positive number. If the current print position is already beyond that specified by expression, TAB goes to the specified expression on the next line. If the specified value is greater than the length of a display line (52 characters), TAB simply keeps counting character positions on subsequent lines to arrive at the specified column position. Thus,

```
PRINT TAB(52) "Here"
```

would print HERE beginning in the first character position of the next display line.

TAB operates only with the PRINT statement -- it does not work with the PRINT# statement.

EXAMPLE

```
1000 PRINT TAB(0) "0"; TAB(10) "10"; TAB(20) "20"; TAB(30) "30";  
TAB(40) "40"; TAB(50) "50"  
1100 FOR Position=0 TO 10  
1200 PRINT TAB(Position) "Tab ";Position  
1300 NEXT Position  
1400 END
```

This program prints the word "Tab" and the tab's number at columns 0 through 10.

TIME\$

The time function.

FORMAT

TIME\$

NOTES

TIME\$ returns the current time as a 13 character string from the Compass Computer system's real-time clock. The string takes the form hh:mm:ss a.m. or hh:mm:ss p.m. where hh is the hour (00 through 12), mm is the minutes (00 through 59) and ss is the seconds (00 through 59). NOTE: These characters are string, not numeric, characters. For a program to use them numerically, you must convert them to numbers (see Chapter Six, the VAL statement).

EXAMPLE

```
1000 PRINT "The time is "; TIME$
1100 FOR Loop=1 TO 5
1200 FOR Rest=1 TO 128: NEXT Rest
1300 LET Second$ = MID$(TIME$,7,2)
1400 PRINT: PRINT "The current seconds are "; Second$
1500 NEXT Loop
1600 END
```

After printing the time in line 1000, the program illustrates that you can take any particular element from the time string and work with it separately. In this case, the current seconds print every second, five times. The loop at line 1200 provides a one second (approximately) pause between printouts.

CHAPTER EIGHT: SEQUENTIAL FILES STATEMENTS

This chapter describes the statements necessary for writing, reading, and manipulating sequential files. Many of these commands also come into play when dealing with random access files (described in Chapter Nine).

The PRINT# and INPUT# statements, described in this chapter, transfer data to and from sequential files. The files created by the PRINT# statement are in a format called "interchange file format" and the INPUT# statement expects files it reads to be in this same format.

The interchange file format enables GRiD applications to place data in columns and rows for tabular or cell-based applications such as GRiDPLAN, GRiDFILE and GRiDPLOT. Because GRiD applications can read files in interchange format, they can process data generated by GRiDBASIC programs.

NOTE: Many of the examples in this book write or read data from a floppy drive and a subject called "'Testing'." For example:

```
1000 OPEN "D",1,"'f0'Testing'Weekly"
```

If you prefer to put your test files on your bubble, replace "'f0" with "'b0." For the hard disk, substitute "'w0." The GRiDBASIC OPEN command can create a title (like "'Weekly") but not a subject. If you want to create a special subject for your examples and test programs, you must do this beforehand.

If you prefer not to deal with pathname syntax, put the GETFILE\$ statement in your program. It presents the standard application file form. See the GETFILE\$ statement later in this chapter.

NOTE: You need only specify a file's device, subject, and/or kind when any of those designations change. For example, if you set usage (CODE-U) to the

floppy drive and then select the subject "'Testing" to write a BASIC program in, you could write

```
1000 OPEN "0",1,"'Weekly"
```

instead of

```
1000 OPEN "0",1,"'f0'Testing'Weekly~Text"
```

Like GRiDWRITE, GRiDBASIC assigns "Text" as the kind for file's created with the OPEN and PRINT# statements.

CLOSE

This statement closes a file or files previously enabled for program access by an OPEN statement.

FORMAT

```
CLOSE [[#]fileTag][,[#]fileTag][, ... ]
```

NOTES

The OPEN statement assigns a file tag number to a particular file name. The CLOSE statement disassociates the tag from this file name, so that you can reassign it to another file. With sequential files, you must close a file to change its mode.

For example, you have to close a program operating a file in the output mode before you can append to it. You can reopen the same file again with its previous tag or a different tag. For a further discussion of fileTags and modes, refer to the OPEN statement later in this chapter.

If you fail to give the CLOSE statement a fileTag, GRiDBASIC closes all open files. NOTE: An END statement automatically closes all files, but a STOP statement does not. GRiDBASIC allows the optional number sign (#) that precedes the fileTag to provide compatibility with other versions of BASIC.

EXAMPLE

```
1000 OPEN "I",1,"f0`Testing`Weekly~"
1100 WHILE NOT EOF(1)
1200 INPUT# 1, Day$
1300 PRINT Day$
1400 WEND
1500 CLOSE 1
1600 PRINT: PRINT "Those are the days of our lives."
1700 END
```

In this example (taken from the OPEN statement below) line 1500 closes the "'Testing" file opened in line 1000. You can also close multiple files with the same statement for example:

```
1000 CLOSE 3,4,15
```

EOF

The end of file function.

FORMAT

EOF(fileTag)

NOTES

The EOF function returns a value that indicates if an end of file has been reached on a specified file. If the end of file has been reached, EOF returns a -1 (true) value. If the end of file has not been reached, a 0 (false) is returned.

The fileTag parameter is the number you specified when you opened the file for input.

EXAMPLE

```
1000 OPEN "I",1,"f0'Testing'Weekly~"
1100 WHILE NOT EOF(1)
1200 INPUT# 1, Day$
1300 PRINT Day$
1400 WEND
1500 CLOSE 1
1600 PRINT: PRINT "Those are the days of our lives."
1700 END
```

NOTE: This example is one of three illustrating the OPEN command. To make this work program work, you will have to type and run the OPEN Output example first (see the OPEN statement later in this chapter).

EOLN

The end of line function.

FORMAT

EOLN(fileTag)

NOTES

The EOLN function returns a value that indicates if an end of line within a specified file has been reached. An end of a line within a file is indicated by the carriage return-line feed combination. If the most recent character read from a file is followed by a carriage return-line feed, or if the end of file has been reached, the EOLN function returns a -1 (true) value; otherwise, it returns a 0 (zero).

This function is especially useful when reading interchange files from other GRiD applications.

The fileTag parameter is the number you specified when the file was opened for input.

EXAMPLE

```
1000 OPEN "I",1,"f0'Testing'AllChecks~MyKind"
1100 INPUT "Take balance from what row (2-13)"; Row
1200 LET Lines=0
1300 WHILE NOT EOF(1)
1400     IF EOLN(1) THEN LET Lines=Lines+1
1500     INPUT# 1, Record$
1600     IF Lines = Row THEN LET Goal$ = Record$
1700 WEND
1800 PRINT
1900 PRINT "The balance at row ";Row;" is ";Goal$
2000 END
```

This program lets you take any number from the balance column of the worksheet shown below in Figure 8-1. For example, if you select row 5, the program will return the amount 479251.43 (the amount after the 02/03 Deposit).

If you want to set up Figure 8-1 in a worksheet, make all but the first item in the balance column a formula that adds current line's Amount to the previous Balance column amount. The first item is the absolute amount, 491084.00. The second Balance item (478605.47) results from adding -12478.53 to the absolute amount.

Check no	Payee	Amount	Balance
	start balance		491084.00
1000	Acme Realty	-12478.53	478605.47
1001	Local Power	-5601.89	473003.58
1002	Telephone	-3016.92	469986.66
23457	02/03 Deposit	9264.77	479251.43
1003	Fass Freight	-1032.14	478219.29
1004	Ace Credit	-15629.01	462590.28
1005	Fleet Rents	-4912.30	457677.98
1006	Personnel	-35971.95	421706.03
1007	A-1 Cleaning	-856.75	420849.28
1008	StarInsurance	-1478.42	419370.86
1009	Heavy Equip	-25819.66	393551.20

Figure 8-1. Worksheet Figures for Example Program

The EOLN function works by searching for the carriage return-line feed combination. In the case of this example program, line 1400 increments a line counter (the variable "Line") each time it encounters an EOLN. When the value of Line equals the value of Row (input by the user), the program prints the last field.

NOTE: Record\$ reads one record at a time, not one line. One cell, begun and/or ended by the Tab character constitutes a record.

GETFILE\$

The get file statement.

FORMAT

```
string$=GETFILE$("promptMessage")
```

NOTES

Normally, programmers specify file pathnames with the program development syntax --

```
'Device'Subject'Title~Kind~
```

For example: 1000 OPEN "I",1,"f0'Testing'Weekly~Text~"

The GETFILE\$ statement lets you bypass this syntax by bringing you the standard file form. Filling in the form and confirming it brings you the desired file.

You may prefer GETFILE\$ over the pathname syntax if you have trouble understanding pathname syntax or if you want your program to work with different files. On the other hand, if your program uses just one file (or only a few -- you could change a parameter before running the program), go with pathname syntax. Likewise, if you value quick access time, choose pathname syntax.

EXAMPLE

```
1000 MyFile$=GETFILE$("Select file and confirm")
1100 OPEN "I",1, MyFile$
1200 WHILE NOT EOF(1)
1300     INPUT# 1, Day$
1400     PRINT Day$
1500 WEND
1600 END
```

In this example, line 1000 assigns the GETFILE\$ function to the string variable, MyFile\$ along with the prompt

```
    Select file and confirm
```

The prompt appears in the message line when you run the program. Once you give the file information to the form and confirm, the string variable delivers that information to the program (see line 1100).

INPUT#

This statement assigns values to program variables by reading data items from a sequential file.

FORMAT

INPUT# fileTag,variablesList

NOTES

The fileTag parameter is the number you specified when the file was opened for input.

Data items read from the file are assigned to the variables specified in the variablesList. Each data item read from the file must be of the same type as that specified by the corresponding variable name. The variable names in variablesList can be any mix of numeric and string variable names, including subscripted variables.

INPUT# expects the file to be in GRiD's standard interchange file format: data items are separated by Horizontal Tabs or Carriage Return-Line Feed pairs. The PRINT# creates this interchange file format, as do GRiD's cell-based applications such as GRiDFILE and GRiDPLAN.

If the end of a file is reached while an item is being input, the item is terminated. If a type mismatch occurs between the data item and the variable that it is being assigned to, or the file has an insufficient number of items, the program halts and an appropriate error message appears.

The INPUT# statement can obtain data from the keyboard; you open the keyboard just as you would any other file. The keyboard's filename is "CI" (for Console Input). For example,

```
OPEN "I",1,"CI"
```

If you choose keyboard input, you must write a prompt for your user(s); unlike the INPUT statement, the INPUT# statement does not print a question mark or a prompt message.

EXAMPLE

```
1000 OPEN "I",1,"'f0'Testing'Weekly~"  
1100 WHILE NOT EOF(1)  
1200 INPUT# 1, Day$  
1300 PRINT Day$  
1400 WEND  
1500 CLOSE 1  
1600 PRINT: PRINT "Those are the days of our lives."  
1700 END
```

NOTE: This example is one of three illustrating the OPEN command. To make this work program work, you will have to enter and run the OPEN Output example first.

INPUT\$

Prefer INPUT\$ over INPUT# for handling communications files or for reading large sections of files.

FORMAT

INPUT\$(tag#, bytes)

NOTES

INPUT\$ fetches the number bytes (or characters) assigned to it in its argument from the file represented by the file tag number. You can assign part or all of the characters read from a communications or other file into one string with INPUT\$.

NOTE: If you give the statement a greater number of characters to fetch than exist within the file, INPUT\$ quits when it reaches the end of file character.

EXAMPLE

```
1000 OPEN "I",1, "'f0'Testing'AnotherDay~"
1100 INPUT "Get how many characters from this file"; HowMany
1200 WantToSee$=INPUT$(1,HowMany)
1300 PRINT
1400 PRINT WantToSee$
1500 END
```

This example has the INPUT\$ statement fetch as many character from the file of weekdays as the user specifies. Unlike INPUT#, INPUT\$ does not convert the end-of-line characters (carriage return-line feed). Rather, it prints the entire string of characters without breaking at the end of lines (except for the right margin).

NOTE: The OPEN output example creates a text file with the days of the week in it. You can create the same file by invoking GRIDWRITE and typing the days in a vertical list. The INPUT\$ program above can read it, as it can read any text file.

KILL

The KILL statement erases a file.

FORMAT

KILL filename

NOTES

Follow the KILL statement with the file name of the file you want to erase. You can present this in the form of a string variable. In fact, the most efficient way to issue a KILL is with a file form created with the GETFILE\$ statement (discussed earlier in this chapter). GETFILE\$ delivers its data to a string variable.

EXAMPLE

```
1000 OPEN "Q",1,"'f0'Testing'NewFile": CLOSE 1
1100 PRINT "NewFile created!"
1200 PRINT "KILL NewFile by selecting it."
1300 LET Joy%=GETFILE$ ("Select FloppyDisk-Testing-NewFile-Text and
confirm")
1400 KILL Joy%
1500 PRINT: PRINT "NewFile KILLED. See if NewFile is still there."
1600 Search%=GETFILE$ ("Press ESC after viewing files")
1700 PRINT: PRINT "KILL erased the file."
1800 END
```

Line 1000 creates a file. In line 1300, the GETFILE\$ statement presents a file form. We recommend GETFILE\$ over typing file name syntax. Line 1400 erases the file named in the form. Line 1600 presents second file form, so that you can see for yourself that KILL indeed erased the file.

LOC

The locating statement.

FORMAT

expression=LOC(tag#)

NOTES

LOC locates a portion of a file by returning a number from a file. What that number represents depends on the type of file involved.

Random	The record number of the last record read or written.
Sequential	The number of records read or written since the last OPEN.
Communications	The number of characters waiting to be read in the input buffer.

EXAMPLE

```
1000 OPEN "I",1,"f0'Testing'AnotherDay~"
1100 PRINT "Record","Byte":PRINT
1200 WHILE NOT EOF(1)
1300     LET MyByte=LOC(1)
1400     INPUT# 1, Day$
1500     PRINT Day$, MyByte
1600 WEND
1700 CLOSE 1
1800 END
```

This example reads a sequential file. This example gets the byte number of each record in a days-of-the-week file. If you want to run this file, you can create a text file called "AnotherDay" by typing in the days of the week in GRIDWRITE, putting each day on its own line.

When you run this example, the "S" beginning Sunday appears as the byte 1. The "M" in Monday as the ninth character. Why? Because in addition to the six characters in "Sunday," LOC also counts the two invisible characters at the end of the line -- carriage return and line feed. Remember: LOC returns the absolute position of each byte.

LOF

The length of file statement.

FORMAT

expression=LOF(fileTag)

NOTES

LOF returns a file's length in bytes. You must supply the file's file tag in parentheses.

EXAMPLE

```
1000 OPEN "I",1, "'f0'Testing'AnotherDay~"
1100 Length=LOF(1)
1200 PRINT
1300 PRINT "The length of this file is "; Length; " characters."
1400 END
```

OPEN

This statement opens a file for a particular kind of access.

FORMAT

```
OPEN "accessMode"[#]fileTag,"fileName"
```

NOTES

Here is a typical OPEN statement:

```
1000 OPEN "I", 3, "'wO'Taxes'January"
```

The accessMode parameter specifies the way that subsequent PRINT# and INPUT# statements in a program can access this file. Further, we can make subsequent references to this program with the number three, instead of with the pathname "'wO'Taxes'January."

GRiDBASIC has three access modes:

"I"	specifies sequential input
"O"	specifies sequential output
"A"	specifies sequential output to be appended

Note that a single OPEN statement can only establish access for one sequential activity at a time. A sequential file cannot be OPEN for both input and output at the same time. To change the type of access you have assigned to a file, you must first CLOSE the file, then execute another OPEN statement specifying the new type of access.

The fileTag parameter is a number that you specify to be associated with this fileName for a particular OPEN operation. Subsequent accesses to the file with PRINT# or INPUT# statements can then refer to the file simply by the fileTag number; you need not specify the file name or type of access. NOTE: GRiDBASIC allows the optional number sign (#) that precedes the fileTag for compatibility with other versions of BASIC.

A file can be open under only one fileTag number at a time. You cannot have a file simultaneously OPEN for input and output, for multiple inputs, or for multiple outputs. To perform two access operations, you need two open operations and two tag numbers. For example:

```
1000 OPEN "I",1 "'MyFile"  
1100 OPEN "O",2 "'YourFile"
```

This opens the file titled "'MyFile" for input and gives it tag number 1. Line 1100 opens a second file to receive this data (output), "'YourFile," with the tag number 2.

The fileName parameter can be any name you have specified up to 80 characters in length. For details, see "File Naming Conventions" near the end of Chapter 2. You can also use the standard Compass file form to get file names for your BASIC programs; see the GETFILE\$ statement earlier in this chapter.

EXAMPLE (OUTPUT)

```
1000 OPEN "O",1,"f0'Testing'Weekly~"
1100 DATA Sunday, Monday, Tuesday, Wednesday, Thursday, Friday,
Saturday
1200 FOR Week=1 TO 7
1300 READ Day$
1400 PRINT Day$
1500 PRINT# 1, Day$
1600 NEXT Week
1700 CLOSE 1
1800 PRINT: PRINT "The Weekly file is closed."
1900 END
```

This example creates a file with the title "'Weekly" and writes the names of the days of the week into it.

EXAMPLE (INPUT)

```
1000 OPEN "I",1,"f0'Testing'Weekly~"
1100 WHILE NOT EOF(1)
1200 INPUT# 1, Day$
1300 PRINT Day$
1400 WEND
1500 CLOSE 1
1600 PRINT: PRINT "Those are the days of our lives."
1700 END
```

This example opens the previous file, retrieves the days of the week from it, and prints them on the screen.

EXAMPLE (APPEND)

```
1000 OPEN "A",1,"`f0`Testing`Weekly~"
1100 DATA Yesterday, Today, Tomorrow, The Day after that
1200 FOR Now = 1 TO 4
1300 READ Day$
1400 PRINT# 1, Day$
1500 PRINT Day$
1600 NEXT Now
1700 CLOSE 1
1800 PRINT: PRINT "Those were the days, my friend."
1900 END
```

This example appends four lines to the original file -- Yesterday, Today, Tomorrow, and The Day after that. If you re-run the INPUT example, you will see these new additions.

PRINT#

The PRINT# statement writes data to a sequential file.

FORMAT

PRINT#fileTag,expression[(,|;)][expression] ... [(,|;)]

NOTES

The fileTag parameter is the number you specified when you opened the file for output. It identifies the sequential file that is to receive the data.

PRINT# writes the data contained in the expression(s) to the file with appropriate delimiting characters automatically inserted. Your choice of punctuation (either a comma or a semicolon) between expressions determines the delimiting characters written to the file to separate the items in each expression. You can use either commas (,) or semicolons (;) as separators.

NOTE: If you intend to print to an Epson printer and want your commas to perform a tabbing function, see the "Epson Notes" at the end of this discussion.

If you place a semicolon between two expressions, PRINT# writes the values of the two expressions with no delimiting character between them.

If you place a comma between two expressions, a horizontal tab character is written to the file separating the contents of the first expression from the contents of the second expression.

If a list of expressions terminates without a comma or semicolon, PRINT# writes a carriage return-line feed at the end of the list. If a comma terminates a list of expressions, PRINT# places a horizontal tab character after the last expression. If a semicolon terminates a list of expressions, it suppresses any delimiting character. Thus a subsequent PRINT# statement begins writing data to the file beginning at the point where the last PRINT# left off.

NOTE: The format of the file created by the PRINT# statement is compatible with the interchange file format. As a result, cell-based GRiD applications such as GRiDPLOT, GRiDFILE and GRiDPLAN can work with these files.

Choose the INPUT# statement to input data from a file that you created with the PRINT# statement.

EPSON NOTES

For the Epson to interpret GRiDBASIC's commas correctly -- providing tabs -- you must follow the PRINT# command with the file tag number, an ESC D (represented by CHR\$(27)+"D") and the column number of each tab preceded by the CHR\$ statement. Concatenate these tab positions with the plus sign (+). All such statements must end with the null character (CHR\$(0)). Do NOT exceed an 80-character line. An example command assigning 15 character-wide tabs follows:

```
PRINT# 1, CHR$(27)+"D"+CHR$(15)+CHR$(30)+...+CHR$(0)
```

EXAMPLE

```
1000 OPEN "D",1,"f0'Testing'Weekly~"
1100 DATA Sunday, Monday, Tuesday, Wednesday, Thursday, Friday,
Saturday
1200 FOR Week=1 TO 7
1300 READ Day$
1400 PRINT Day$
1500 PRINT# 1, Day$
1600 NEXT Week
1700 CLOSE 1
1800 PRINT: PRINT "The Weekly file is closed."
1900 END
```

This example (from the OPEN statement) writes the days of the weeks to a file called "'Weekly.'" The PRINT# statement in line 1500 transmits this data one string (or day!) at a time.

PRINT# USING

The PRINT# USING statement writes data to sequential files or to the printer in a specified format.

FORMAT

PRINT# fileTag, USING; formatstring; {numericvar|stringvar list}

NOTES

PRINT# USING takes the same arguments as PRINT USING. The only syntactical difference between the two is the presence of the number sign (#) and tag number following the word "PRINT." For details on this statement's many formatting possibilities, see PRINT USING in Chapter Seven.

EXAMPLE

```
1000 OPEN "Q",1, "epson"
1100 FOR Times = 1 TO 12
1200 LET Number = 100000*RND(1)
1300 PRINT Number;" ";; PRINT USING "$#####.##"; Number
1400 PRINT# 1, Number;
1500 PRINT# 1, USING " $#####.##";Number
1600 NEXT Times
1700 CLOSE 1
1800 END
```

This example generates 12 random numbers and then prints and formats them -- to the screen and to the printer. Line 1400 prints the unformatted number; line 1500 does the formatting. In this case, we have turned the number into a dollar amount and preceded that with spaces to separate the unformatted and formatted numbers. Figure 8-2 below is a typical printout.

Note that to print, you must open your printer as a file (line 1000) and give it a tag number. The PRINT#, PRINT# USING and CLOSE statements all take advantage of this tag number.

This example creates a printout to the screen and on paper that resembles the printout below in Figure 8-2. The left column displays the random number we generated. The right column shows how PRINT# USING formatted the same number.

6001.3733119707	\$6,001.37
62092.0119020371	\$62,092.01
38194.8577096208	\$38,194.86
82920.5767910277	\$82,920.58
7277.02754253452	\$7,277.03
81165.7892729076	\$81,165.79
3288.31921873808	\$3,288.32
88273.4416723888	\$88,273.44
89706.2638284886	\$89,706.26
52875.5626764324	\$52,875.56
65941.8631265736	\$65,941.86
53293.6598764019	\$53,293.66

Figure 8-2. PRINT# USING formatting of Random Numbers

CHAPTER NINE: RANDOM FILE STATEMENTS

Random access (also called "direct access") files differ from sequential files in several important ways. First, each data unit or record is of a fixed length (specified by the programmer). Second, you can go directly to any record within a random access file, rather than having to go through the entire file. This is true for both read and write activities. Third, random files have a buffer in RAM memory. Your program interacts with the buffer, rather than directly with your storage device.

Random access files share statements with sequential files. Note, however, that some of these statements don't behave exactly the same. For example, the LOF statement in a sequential file returns the length of that file in bytes. In a random file, LOF returns the total number of records in the file. These two numbers only equal each other when a random file has one-byte long records!

Another example. The OPEN statement has only one access mode for opening a random access file -- "R" (Sequential files have three). With random files, it doesn't matter whether you are opening the file for INPUT or OUTPUT. Further, the OPEN statement in a random access file also takes an optional argument after the file name, the buffer length.

Although this chapter contains plenty of working examples, you may want to look at the basic steps involved in creating random access write and read files. First to create a random access file and write data to it, follow these steps.

A RANDOM ACCESS WRITE FILE

- ① OPEN the file with an "R" and an optional buffer size specification.

```
1000 OPEN "R",1,"'f0'Testing'Demograf.1~",30
```

- ② Define the sizes of the fields in your record buffer with the FIELD statement.

```
1100 FIELD 1, 12 AS Name$, 3 AS Age$, 15 AS City$
```

- ③ Gather the data you want to write. You can do this by reading from other files, with INPUT and/or READ DATA statements, for example.

```
1200 INPUT "Name"; N$
```

- ④ Put this data into the buffer with the LSET or RSET statements. Use the variable names you assigned to fields in the FIELD statement.

```
1600 LSET Name$=N$
```

- ⑤ Write the data to the file with the PUT statement.

```
2000 PUT 1
```

- ⑥ Close the file.

```
2200 CLOSE 1
```

Here's an outline for reading data from a file.

A RANDOM ACCESS READ FILE

- ① OPEN the file with an "R" and an optional buffer size specification.

```
1000 OPEN "R",1,"'f0'Testing'Demograf.1~",30
```

- ② Define record buffer field sizes in with the FIELD statement.

```
1100 FIELD 1, 12 AS Name$, 3 AS Age$, 15 AS City$
```

- ③ Use the GET statement to read your data from the file.

```
1500 GET 1, RecordNo
```

- ④ Process this data and send it to the screen or other device.

```
1600 PRINT Name$;
```

- ⑤ Close the file.

```
2400 CLOSE 1
```

CVI, CVS, CVD

The convert string to integer function.
The convert string to single-precision function.
The convert string to double-precision function.

FORMAT

CVI(2-byte string)
CVS(8-byte string)
CVD(16-byte string)

NOTES

Programmers often convert numeric values to strings so they can format these values with the LSET or RSET statements. However, the system cannot perform mathematical operations on string values. Only on numeric values. Therefore, they convert string values back to numbers. CVI converts a 2-byte string, CVS converts a 4-byte, and CVD converts an 8-byte string.

Choose the CV function that matches the MK\$ function that made the original number into a string. Table 9-1 below illustrates this.

MK\$ Form	CV Form	No. of Bytes
MKI\$	CVI	2
MKS\$	CVS	4
MKD\$	CVD	8

Table 9-1. Choosing MK\$ and CV Functions

EXAMPLE

```
1000 OPEN "R",1,"f0'Testing'MyNumbers~",8
1100 FIELD 1, 8 AS Number$
1200 FOR Count=1 TO 3
1300 INPUT "Any number"; N
1400 LSET Number$=MKD$(N)
1500 PUT 1
1600 NEXT Count
1700 PRINT:PRINT "MKD$ Form"; TAB(13) "After CVD": PRINT
1800 FOR Count=1 TO 3
1900 GET 1, Count
2000 PRINT Number$;
2100 PRINT TAB(13) CVD(Number$)
2200 NEXT Count
2300 PRINT
2400 END
```

This program asks you to enter any three numbers. It then converts them to string format in line 1400. It then writes these numbers to the screen and shows them in the form in which they are stored (the "MKD\$ Form") and the numeric form they take after conversion to double precision (CVD). See the MKD\$ function below for details on its operation.

FIELD

FIELD sets up a random file buffer.

FORMAT

FIELD [#] tag#, number AS string\$ [, number AS string\$] ...

NOTES

The FIELD statement breaks the buffer into individual fields. Thus the buffer is the length of the record that comprises these fields. To maximize efficient use of memory and storage space, add the numbers of characters for each field together and give the resulting sum as the optional buffer length parameter.

The AS statement assigns buffer space in characters (indicated by the number preceding AS) to a variable (following AS).

EXAMPLE

```
1000 OPEN "R",1,"'f0'Testing'Demograf.1~",30
1100 FIELD 1, 12 AS Name$, 3 AS Age$, 15 AS City$
1200 WHILE NOT EOF(1)
1300 GET 1
1400 PRINT Name$;
1500 PRINT Age$;
1600 PRINT City$
1700 PRINT
1800 WEND
1900 CLOSE 1
2000 END
```

This program reads in all the records from the 'Demograf.1 file. It allots space in the (random) buffer for its three fields as follows: 12 characters for the string variable Name\$, 3 characters for the string variable AGE\$, and 9 to the string variable ZIP\$.

GET

The GET statement retrieves data for random file access.

FORMAT

GET [#] tag#[, number]

NOTES

The GET statement reads one record at a time into the buffer. If you do not specify a number, any reading of these records causes their content to appear in the order in which they exist in the file. If you specify a number, the record belonging to that record number appears. Thus if you ask for

```
1500 GET 1,3
```

line 1500 will get the third record from the file you assigned the tag number of 1.

EXAMPLE

```
1000 OPEN "R",1,"f0\Testing\Demograf.1~",30
1100 FIELD 1, 12 AS Name$, 3 AS Age$, 15 AS City$
1200 LET Items=LDF(1)
1300 PRINT "Record number (between 1 and "; Items;
1400 INPUT ") please", RecordNo
1500 GET 1, RecordNo
1600 PRINT Name$;
1700 PRINT Age$;
1800 PRINT City$
1900 INPUT "This person's true age", A$
2000 PRINT
2100 LSET Age$=A$
2200 PUT 1, RecordNo
2300 GOTO 1300
2400 CLOSE 1
2500 END
```

This example reads whatever record number you specify (in line 1400) and prints the appropriate data on the screen. The program then gives you the opportunity to change the age parameter.

LOC

LOC locates a record.

FORMAT

LOC(tag#)

NOTES

LOC returns the record number of the next record that you can either GET or PUT. When this function sees the EOF marker, it looks no further.

EXAMPLE

```
1000 OPEN "R",1,"f0'Testing'Demograf.1~",30
1100 FIELD 1, 12 AS Name$, 3 AS Age$, 15 AS City$
1200 WHILE LOC(1)<=8
1300 PRINT LOC(1);" ";
1400 GET 1
1500 PRINT Name$;
1600 PRINT Age$;
1700 PRINT City$
1800 PRINT
1900 WEND
2000 CLOSE 1
2100 END
```

This example uses LOC to test whether the WHILE WEND should continue (line 1200) and to print each record's number before printing the contents of the record (line 1300).

LOF

The length of file statement

FORMAT

LOF(fileTag)

NOTES

LOF returns a file's length in records.

EXAMPLE

```
1000 OPEN "R",1,"'f0'Testing'Demograf.1~",30
1100 FIELD 1, 12 AS Name$, 3 AS Age$, 15 AS City$
1200 WHILE LOC(1) <= LOF(1)
1300 PRINT LOC(1);" ";
1400 GET 1
1500 PRINT Name$;
1600 PRINT Age$;
1700 PRINT City$
1800 PRINT
1900 WEND
2000 CLOSE 1
2100 END
```

In this example, line 1200 we test whether to continue the WHILE WEND loop by comparing the file record number (LOC) with the number of records in the file (LOF). If the record number is less than or equal to LOF, the WHILE WEND loop continues.

LSET and RSET

The LSET and RSET statements

FORMAT

```
LSET fieldString=programString  
RSET fieldString=programString
```

NOTES

LSET and RSET statements assign a string created within the current program to one of the string variables defined in the FIELD statement. In the event that a value does not take up all the string space allotted to it, LSET will left-justify the value within the space. Similarly, RSET right-justifies when space remains. NOTE: You must convert numeric variables to string variables before doing this. Either the MK\$ statement (see below) or the STR\$ can do the job.

CAUTION: Do not use a field variable in an input statement nor put it on the left side of an assignment (LET) statement. Either practice causes the variable pointer to point not to the random file buffer, but to string space. The result: garbage in your file.

EXAMPLE

```
1000 OPEN "R",1,"f0'Testing'Demograf.1~",30  
1100 FIELD 1, 12 AS Name$, 3 AS Age$, 15 AS City$  
1200 INPUT "Name"; N$  
1300 IF N$="" THEN GOTO 2200  
1400 INPUT "Age"; A$  
1500 INPUT "City"; C$  
1600 LSET Name$=N$  
1700 LSET Age$=A$  
1800 LSET City$=C$  
1900 PRINT  
2000 PUT 1  
2100 GOTO 1200  
2200 CLOSE 1  
2300 PRINT: PRINT "This input session is over"  
2400 END
```

This example writess data to a random access file. The LSET statements (lines 1600-1800), it assign the values in the input variables (lines 1200, 1400, and 1500) to the field variables (assigned in line 1100). If you wanted your output right-justified instead of left-justified, you would substitute RSET for each occurence of LSET.

MKI\$, MKS\$, MKD\$

The make string function.

FORMAT

MKI\$ (expression)

MKS\$ (expression)

MKD\$ (expression)

NOTES

The MK\$ function converts numeric expressions (including variables and numbers) into 4-byte strings. You must convert any numeric expressions before submitting them to the LSET or RSET. (You must choose one of these two to put data into the buffer).

As a general rule, choose MKD\$ to convert your strings. At 8 bytes, it yields the greatest precision and, with its corollary CVD, minimizes the possibility for returning an inaccurate number from storage.

EXAMPLE

```
1000 OPEN "R",1,"f0'Testing'MyNumbers~",8
1100 FIELD 1, 8 AS Number$
1200 FOR Count=1 TO 3
1300 INPUT "Any number"; N
1400 LSET Number$=MKD$(N)
1500 PUT 1
1600 NEXT Count
1700 PRINT:PRINT "MKD$ Form"; TAB(13) "After CVD": PRINT
1800 FOR Count=1 TO 3
1900 GET 1, Count
2000 PRINT Number$;
2100 PRINT TAB(13) CVD(Number$)
2200 NEXT Count
2300 PRINT
2400 END
```

This program asks you to enter any three numbers. It then converts them to string format in line 1400. Note that we put the result of our MKD\$ function in the string variable, Number\$. Lines 1900 to 2100 bring the numbers back from storage -- in the MK\$ string form and in the converted form. See the CVD function above for details on its use.

OPEN

The OPEN statement creates a buffer in RAM memory and prepares the system to write data to or read data from the specified file. If the file doesn't exist, OPEN creates the file title. It cannot, however, create a subject.

FORMAT

```
OPEN "R"[#]fileTag,"fileName",[bufferLength]
```

NOTES

Here is a typical OPEN statement:

```
1000 OPEN "R", 3, "'w0'Taxes'January", 63
```

When placed in the context of random access files, OPEN has only one accessMode parameter -- "R" for Random. This specifies the kind of file manipulation activities the file allows. NOTE: where sequential files have three possible letters -- "I," "O," and "A," random files have just one accessmode -- "R."

NOTE: Unlike sequential files that must CLOSE and issue a new OPEN statement before changing its access activity, random files can do both input (GET) and output (PUT) under the same OPEN statement. See the example under the MKD\$ command. For details on sequential files, see the OPEN statement in Chapter Eight.

The fileTag parameter is a number that you specify to be associated with this fileName for a particular OPEN operation. Subsequent accesses of the file with GET or PUT statements can then refer to the file simply by the fileTag number; you need not specify the file name or type of access. In the example above, this number is 3.

A file can be open under only one fileTag number at a time. NOTE: GRI BASIC allows the optional number sign (#) preceding fileTag for compatibility with other versions of BASIC.

The fileName parameter can be any name you have specified up to 80 characters in length. For details, see "File Naming Conventions" near the end of Chapter 2.

The optional bufferLength parameter sets the size of the buffer. For greatest efficiency, you should assign this the same number of bytes as the total number bytes in the field statement. In the example above, we defined the length of the buffer as 63 characters. The default length for the buffer is 128 bytes.

EXAMPLE

```
1000 OPEN "R",1,"f0'Testing'Demograf.1~",30
1100 FIELD 1, 12 AS Name$, 3 AS Age$, 15 AS City$
1200 INPUT "Name"; N$
1300 IF N$="" THEN GOTO 2200
1400 INPUT "Age"; A$
1500 INPUT "City"; C$
1600 LSET Name$=N$
1700 LSET Age$=A$
1800 LSET City$=C$
1900 PRINT
2000 PUT 1
2100 GOTO 1200
2200 CLOSE 1
2300 PRINT: PRINT "This input session is over"
2400 END
```

This example inputs data to a random access file. Its OPEN statement assigns this file the access mode parameter, "R" and file tag number "1." It then specifies the floppy drive as the device, "Testing" as the subject, and "Demograf.1" as the title. Finally, it sets aside 30 bytes for the file's buffer length. Note that 30 is the sum of the lengths of the three records given in the FIELD statement (line 1100).

PUT

The PUT statement writes data to a random buffer.

FORMAT

PUT [#]fileTag [, expression]

NOTES

The PUT statement writes data to a random file buffer for transfer to the appropriate storage medium. PUT understands the optional expression (whether a constant or a variable) as a record number. If another record already has the number you specify, PUT will write over it. If you fail to specify a number, PUT assigns the next available number.

EXAMPLE

```
1000 OPEN "R",1,"f0`Testing`Demograf.1~",30
1100 FIELD 1, 12 AS Name$, 3 AS Age$, 15 AS City$
1200 INPUT "Name"; N$
1300 IF N$="" THEN GOTO 2200
1400 INPUT "Age"; A$
1500 INPUT "City"; C$
1600 LSET Name$=N$
1700 LSET Age$=A$
1800 LSET City$=C$
1900 PRINT
2000 PUT 1
2100 GOTO 1200
2200 CLOSE 1
2300 PRINT: PRINT "This input session is over"
2400 END
```

This program puts data into a file called 'Demograf.1. Each time through the loop, it puts a record with three fields -- Name\$, Age\$ and City\$.



CHAPTER TEN: GRAPHICS STATEMENTS

This chapter discusses GRiDBASIC's graphics statements. With these statements you can draw, invert, erase, and four figures:

- The box
- The circle
- The dot
- The line

This manual has one example each for the circle, dot, and line statements (for DRAW, INVERT, and ERASE). Each example shows how the graphic appears after a particular statement by placing part of the graphic against the screen and part against a white box. See Figures 10-1, 10-2, and 10-3 below. We place these examples at the front of the chapter for easy access and comparison. The programs that generated these figures are listed later in this with each relevant statement.

You can also position character strings (DrawChars), create menus, move boxes, and place prompt messages.

In graphics syntax, *x* and *y* represent (respectively) the horizontal and vertical coordinates of the point being described. These points are screen bits or "pixels." The screen is 320 pixels wide and 240 pixels deep.

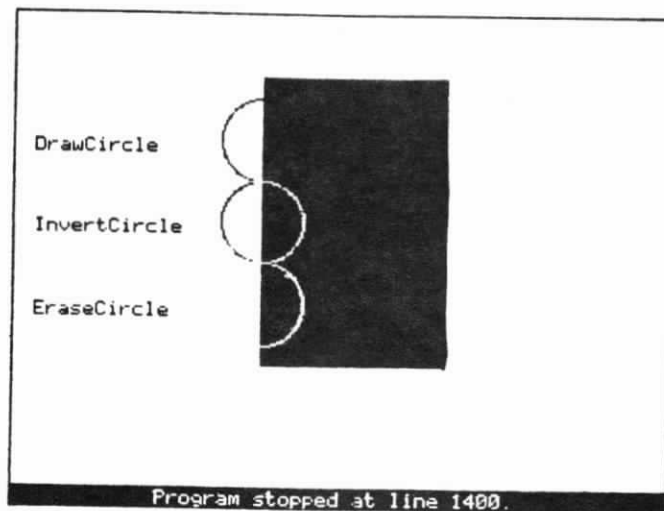


Figure 10-1. The Three Circle Graphics

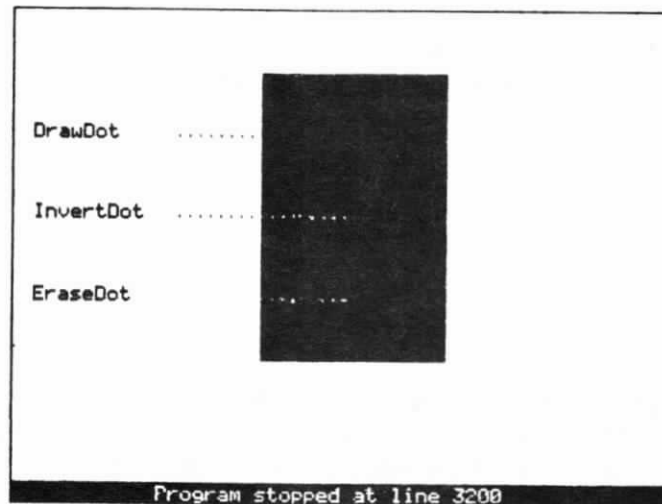


Figure 10-2. The Three Dot Graphics

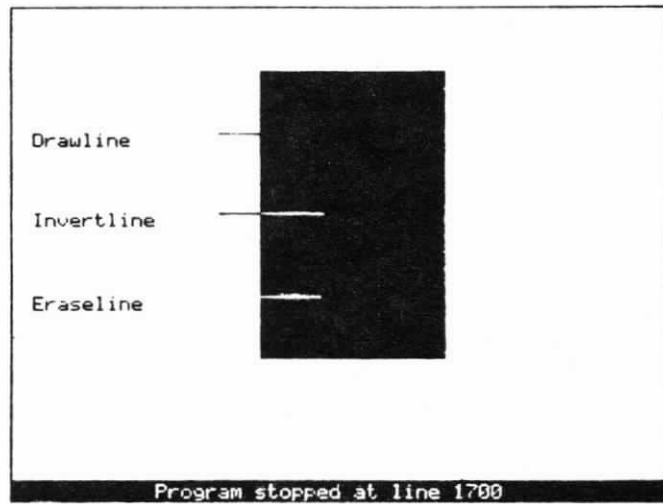


Figure 10-3. The Three Line Graphics

CLEARMSG

The clear message statement.

FORMAT

CLEARMSG

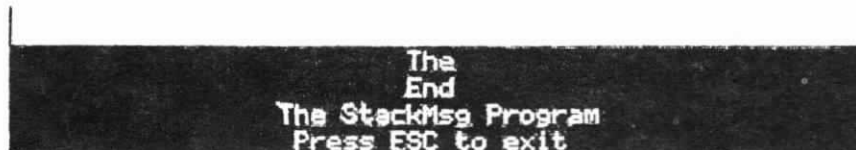
NOTES

CLEARMSG clears any prompt previously specified by the STACKMSG statement. If you include a STACKMSG prompt inside a loop or want to move on to a new message, you must clear the old message with CLEARMSG. If you don't, messages stack up, as shown in Figure 10-4. See STACKMSG later in this chapter for details.

EXAMPLE

```
1000 PRINT "Press a key"
1100 STACKMSG "Press ESC to exit"
1200 STACKMSG "The StackMsg Program"
1300 LET Key$=INKEY$
1400 IF Key$="" THEN GOTO 1300 ELSE PRINT "Key is: "; Key$
1500 CLEARMSG
1600 STACKMSG "End
1700 STACKMSG "The
```

If it weren't for line 1500, the prompt would look like the one in Figure 10-4. CLEARMSG clears the first prompt so that the final prompt looks like the one in Figure 10-5.



```
The
End
The StackMsg Program
Press ESC to exit
```

Figure 10-4. Before CLEARMSG



```
The
End
```

Figure 10-5. After CLEARMSG

DOMENU

This statement creates a menu.

FORMAT

```
variable=DOMENU(prompt$, choice$!choice$[!choice$] ...)
```

NOTES

The DOMENU statement draws a menu at the bottom of the screen. This menu resembles the ones you have seen in GRiD applications. DOMENU asks you to specify the prompt message (prompt\$) at the bottom of the screen and the various choices the menu will offer.

Separate each choice with a bar (!) by pressing CODE-SHIFT-; . DOMENU assigns a number to each choice, the first choice is 1, the second 2, etc. In this way, you can execute the choice with an ON GOTO or ON GOSUB statement. Only the size of the screen limits the number of choices you can present.

EXAMPLE

```
1000 LET Picky$="Make your play"
1100 LET Yours=DOMENU(Picky$,"Stand and fight!Flee and retreat!Buy
'em out")
1200 IF Yours=0 THEN LET Picky$="Oh no you don't: Choose"
1300 ON Yours GOTO 1500,1600,1700
1400 END
1500 PRINT "Fire when ready, Gridley":END
1600 PRINT "Come back, come back, come....":END
1700 PRINT "Okay. Let's talk, turkey":END
```

This example presents a menu as shown in Figure 10-6. "Picky\$" is string variable to which we assign the prompt ("Make your play").

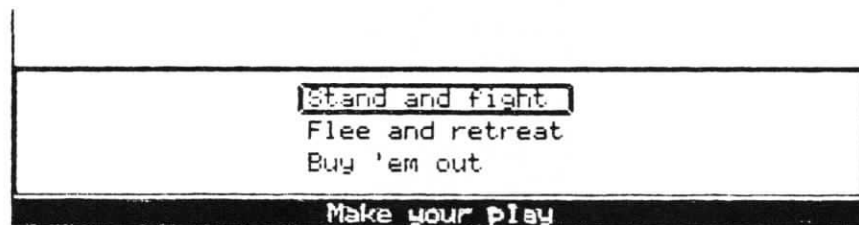


Figure 10-6. A Menu Created with DOMENU

DRAWBOX

The DRAWBOX statement draws a solid (light-colored) rectangle.

FORMAT

DRAWBOX topLeft (x,y) extent (x,y)

NOTES

DRAWBOX needs four coordinates. The first two describe the top left corner of the box. The second two describe the horizontal and vertical extensions from the starting point.

EXAMPLE

```
1000 INPUT "Top left horizontal coordinate";A
1100 INPUT "Top left vertical coordinate";B
1200 INPUT "Extend how far horizontally";C
1300 INPUT "Extend how far vertically";D
1400 EraseBox 0,0,320,240
1500 DrawBox A, B, C, D
1600 FOR Pause=1 TO 100: NEXT Pause
1700 LOCATE 5,210
1800 PRINT "This box has coordinates "; A;"", "B:""", "C:""", "D:""
1900 FOR Pause=1 TO 100: NEXT Pause
2000 END
```

This program asks you to describe a box and then draws that box. See Figure 10-7 for an example.

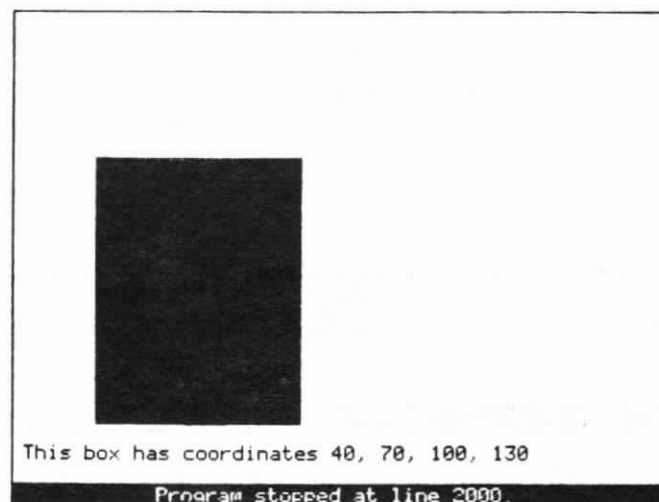


Figure 10-7. An Example of DRAWBOX

DRAWCHARS

DRAWCHARS places characters on the screen at the stated coordinates.

FORMAT

DRAWCHARS string x,y

NOTES

The coordinates in DRAWCHARS define the upper left pixel of the first character in the string. This statement accepts strings surrounded by quotation marks, strings defined by the CHR\$ statement and ASCII numbers, or a combination of the two. You cannot join strings with the semicolon (as with the PRINT statement. Instead, you must always concatenate them with the plus sign (+).

EXAMPLE

```
1000 DRAWCHARS "What's that ringing?",10,10
1100 FOR Starts=1 TO 100: NEXT Starts
1200 FOR Phone=1 TO 3
1300 LET Ring=1
1400 WHILE Ring<30
1500 DRAWCHARS CHR$(142)+CHR$(143),40,40
1600 DRAWCHARS CHR$(142)+CHR$(143),42,40
1700 LET Ring=Ring+1
1800 WEND
1900 FOR Time=1 TO 100: NEXT Time
2000 NEXT Phone
2100 DRAWCHARS "Only the phone",10,70
2200 FOR Pause=1 TO 100: NEXT Pause
2300 DRAWCHARS "Another "+ CHR$(137)+CHR$(138)+CHR$(139)+CHR$(140)+
" presentation",10, 90
2400 END
```

Lines 1000 and 2100 demonstrate placement of a string enclosed in quotation marks. Lines 1500 and 1600 show concatenation of individual ASCII characters (Line 1600 the string repositions the string for an animation effect). Line 2300 combines both quotes and ASCII codes to print the program's final message. The loops at lines 1100 and 2200 delay execution of the program for another effect. See Figure 10-8 for a picture of this program.

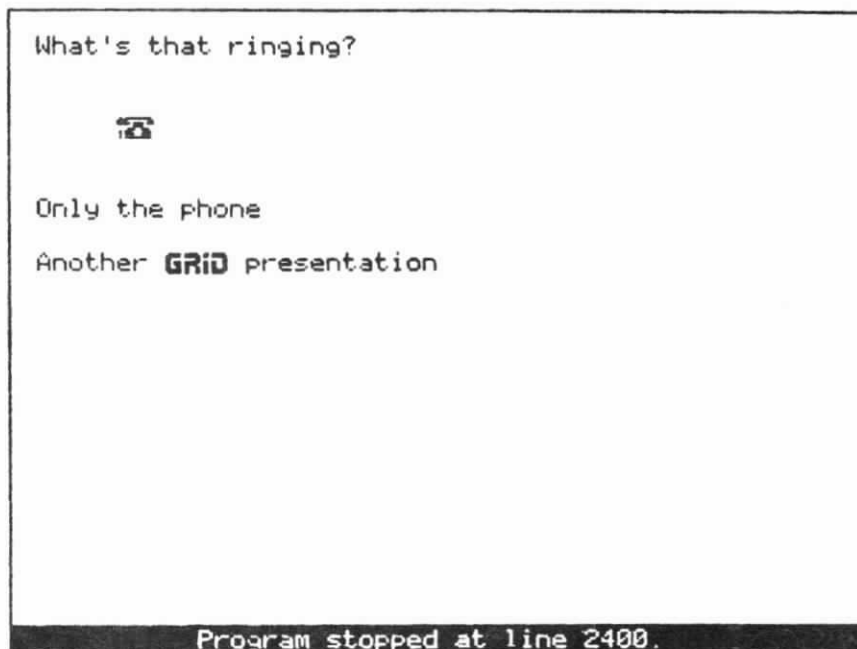


Figure 10-8. The DRAWCHARS Example

DRAWCIRCLE

This statement positions and draws the outline of a circle.

FORMAT

DRAWCIRCLE x,y, radius

NOTES

The x,y coordinates specify the center of the circle. The radius is measured in screen bits.

EXAMPLE

```
1000 DRAWBOX 120, 30, 90, 140
1100 DRAWCIRCLE 120,60,20
1200 LOCATE 10, 60: PRINT "DrawCircle"
1300 INVERTCIRCLE 120,100,20
1400 LOCATE 10, 100: PRINT "InvertCircle"
1500 ERASECIRCLE 120, 140, 20
1600 LOCATE 10,140: PRINT "EraseCircle"
1700 END
```

DRAWDOT

The DRAWDOT statement turns on one screen bit (also known as a "pixel").

FORMAT

DRAWDOT x,y

NOTES

The two arguments are the dot's horizontal and vertical coordinates.

EXAMPLE

```
1000 DRAWBOX 120, 30, 90, 140
1100 LOCATE 10, 55: PRINT "DrawDot"
1200 LOCATE 10, 95: PRINT "InvertDot"
1300 LOCATE 10, 135: PRINT "EraseDot"
1400 REM The DrawDot Routine
1500 LET X=80: Y=60
1600 WHILE X<=160
1700 DRAWDOT X,Y
1800 LET X=X+5
1900 WEND
2000 REM The InvertDot Routine
2100 LET X=80: LET Y=100
2200 WHILE X<=160
2300 INVERTDOT X,Y
2400 LET X=X+5
2500 WEND
2600 REM The EraseDot Routine
2700 LET X=80:Y=140
2800 WHILE X<=160
2900 ERASEDOT X,Y
3000 LET X=X+5
3100 WEND
3200 END
```

This program differs from programs for the other figures in order to put five pixels between the dots. Without these spaces, you cannot tell the difference between similar statements for DOT and LINE.

DRAWLINE

The DRAWLINE statement draws a line.

FORMAT

DRAWLINE startPoint (x,y) endpoint (x,y)

NOTES

DRAWLINE needs four arguments -- the horizontal and vertical points for the start of the line and the horizontal and vertical points for the end of the line.

EXAMPLE

```
1000 DRAWBOX 120, 30, 90, 140
1100 DRAWLINE 100, 60, 150, 60
1200 LOCATE 10, 60: PRINT "Drawline"
1300 INVERTLINE 100, 100, 150, 100
1400 LOCATE 10, 100: PRINT "Invertline"
1500 ERASELINE 100, 140, 150, 140
1600 LOCATE 10, 140: PRINT "Eraseline"
1700 END
```

ERASEBOX

This statement erases a box in the position described by its coordinates.

FORMAT

ERASEBOX topLeft (x,y) extent (x,y)

NOTES

You cannot see ERASEBOX working against a dark background. Only against a light background. The act of erasing only turns screen bits off.

EXAMPLE

```
1000 LET A=120: B=80: C=80: D=80
1100 DrawBox A,B,C,D
1200 FOR Pause=1 TO 200: NEXT Pause
1300 LET A=140: B=100: C=40: D=40
1400 PRINT "This erases the center of the box"
1500 GOSUB 2000
1600 PRINT "And this erases everything."
1700 LET A=0: B=0: C=320: D=240
1800 GOSUB 2000
1900 END
2000 FOR Pause=1 TO 200: NEXT Pause
2100 EraseBox A,B,C,D
2200 FOR Pause=1 TO 200: NEXT Pause
2300 RETURN
```

This example draws a box, erases its center, and then clears (erases) the entire screen. Figure 10-9 shows the program run through the first erasure.

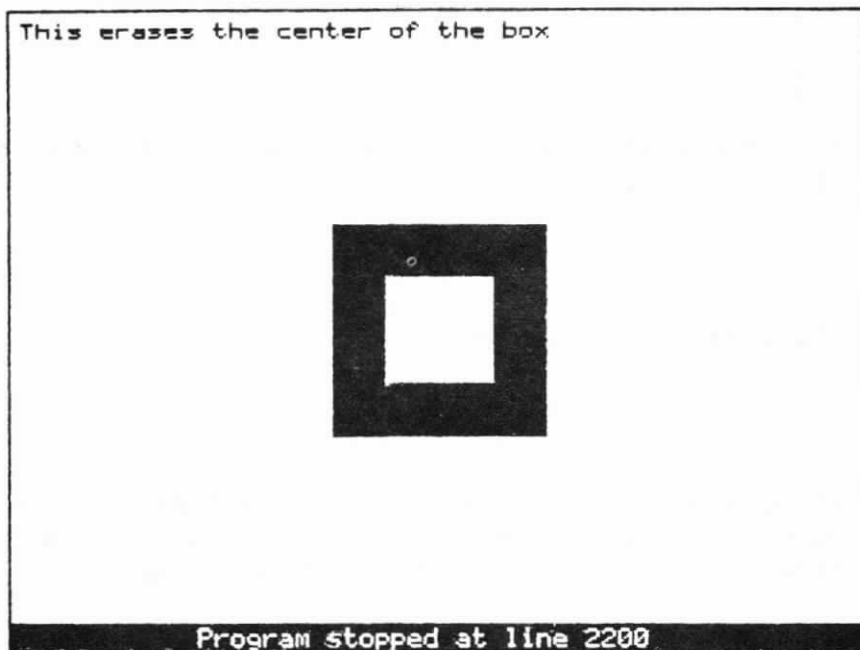


Figure 10-9. An Example of ERASEBOX

ERASECIRCLE

This statement erases a circle of the size and position described by its coordinates.

FORMAT

ERASECIRCLE x,y, radius

NOTES

You cannot see the circle described by ERASECIRCLE unless you erase over a white area. The x,y coordinates specify the center of the circle. The radius is measured in screen bits.

EXAMPLE

```
1000 DRAWBOX 120, 30, 90, 140
1100 DRAWCIRCLE 120,60,20
1200 LOCATE 10, 60: PRINT "DrawCircle"
1300 INVERTCIRCLE 120,100,20
1400 LOCATE 10, 100: PRINT "InvertCircle"
1500 ERASECIRCLE 120, 140, 20
1600 LOCATE 10,140: PRINT "EraseCircle"
1700 END
```

ERASEDOT

This statement erases a dot.

FORMAT

ERASEDOT X,Y

NOTES

ERASEDOT turns off one screen bit. It is only visible when the dot resides on a light background.

EXAMPLE

```
1000 DRAWBOX 120, 30, 90, 140
1100 LOCATE 10, 55: PRINT "DrawDot"
1200 LOCATE 10, 95: PRINT "InvertDot"
1300 LOCATE 10, 135: PRINT "EraseDot"
1400 REM The DrawDot Routine
1500 LET X=80: Y=60
1600 WHILE X<=160
1700 DRAWDOT X,Y
1800 LET X=X+5
1900 WEND
2000 REM The InvertDot Routine
2100 LET X=80: LET Y=100
2200 WHILE X<=160
2300 INVERTDOT X,Y
2400 LET X=X+5
2500 WEND
2600 REM The EraseDot Routine
2700 LET X=80:Y=140
2800 WHILE X<=160
2900 ERASEDOT X,Y
3000 LET X=X+5
3100 WEND
3200 END
```

This program differs from programs for the other figures in order to put five pixels between the dots. Without these spaces, you cannot tell the difference between similar statements for DOT and LINE.

ERASELINE

This statement erases a line in the position described by its coordinates.

FORMAT

ERASELINE startPoint (x,y) endPoint (x,y)

NOTES

ERASELINE needs four arguments -- the horizontal and vertical points for the start of the erasure and the horizontal and vertical points for ending erasure.

EXAMPLE

```
1000 DRAWBOX 120, 30, 90, 140
1100 DRAWLINE 100, 60, 150, 60
1200 LOCATE 10, 60: PRINT "Drawline"
1300 INVERTLINE 100, 100, 150, 100
1400 LOCATE 10, 100: PRINT "Invertline"
1500 ERASELINE 100, 140, 150, 140
1600 LOCATE 10,140: PRINT "Eraseline"
1700 END
```


INVERTBOX

This statement inverts the "colors" of a box in the position described by its coordinates.

FORMAT

INVERTBOX TopLeft (x,y) Extent (x,y)

NOTES

The inversion here amounts to turning off bits that are on and turning on bits that are off. As a result, INVERTBOX describes a light rectangle on a dark background and a dark rectangle against a light background. See Figure 10-10 below.

EXAMPLE

```
1000 DRAWBOX 80,80,100,100
1100 LET A=40: B=100: C=1: D=1
1200 WHILE A<200
1300 INVERTBOX A,120,5,20
1400 A=A+5
1500 WEND
1600 END
```

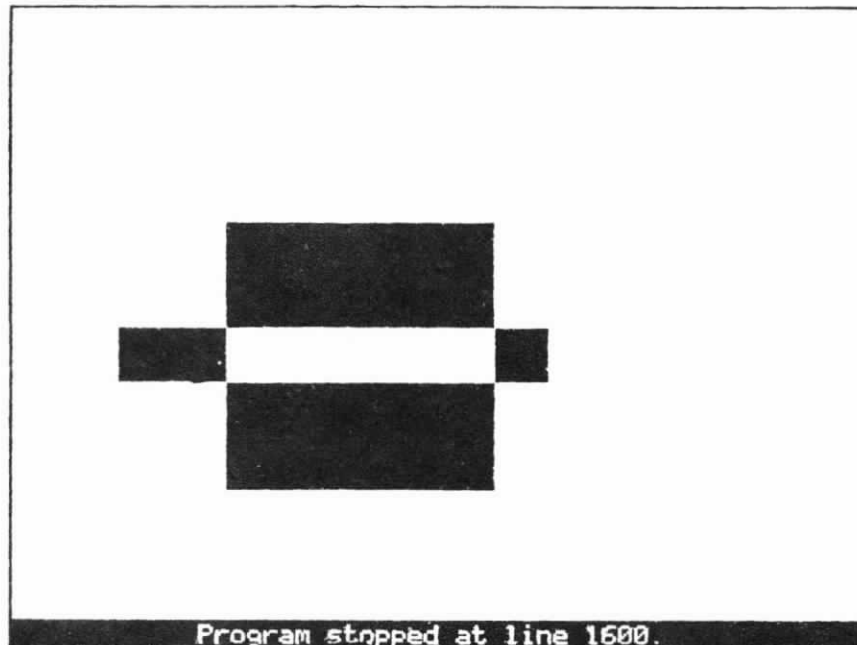


Figure 10-10. An Example of INVERTBOX

INVERTCIRCLE

This statement positions and draws a circle the "colors" of which are the opposite of the background area.

FORMAT

INVERTCIRCLE x,y, radius

NOTES

The inversion here amounts to turning off bits that are on and turning on bits that are off. As a result, INVERTCIRCLE describes a light circle on a dark background and a dark circle against a light background. The x,y coordinates specify the center of the circle. The radius is measured in screen bits.

EXAMPLE

```
1000 DRAWBOX 120, 30, 90, 140
1100 DRAWCIRCLE 120,60,20
1200 LOCATE 10, 60: PRINT "DrawCircle"
1300 INVERTCIRCLE 120,100,20
1400 LOCATE 10, 100: PRINT "InvertCircle"
1500 ERASECIRCLE 120, 140, 20
1600 LOCATE 10,140: PRINT "EraseCircle"
1700 END
```

INVERTDOT

INVERTDOT draws a dot of a "color" opposite that of its background.

FORMAT

INVERTDOT topLeft (x,y) extent (x,y)

NOTES

INVERTDOT places a light dot on a dark background and a dark dot on a light background. If a screen bit is on, INVERTDOT turns it off. If off, it turns it on.

EXAMPLE

```
1000 DRAWBOX 120, 30, 90, 140
1100 LOCATE 10, 55: PRINT "DrawDot"
1200 LOCATE 10, 95: PRINT "InvertDot"
1300 LOCATE 10, 135: PRINT "EraseDot"
1400 REM The DrawDot Routine
1500 LET X=80: Y=60
1600 WHILE X<=160
1700 DRAWDOT X,Y
1800 LET X=X+5
1900 WEND
2000 REM The InvertDot Routine
2100 LET X=80: LET Y=100
2200 WHILE X<=160
2300 INVERTDOT X,Y
2400 LET X=X+5
2500 WEND
2600 REM The EraseDot Routine
2700 LET X=80:Y=140
2800 WHILE X<=160
2900 ERASEDOT X,Y
3000 LET X=X+5
3100 WEND
3200 END
```

This program differs from programs for the other figures in order to put five pixels between the dots. Without these spaces, you cannot tell the difference between similar statements for DOT and LINE.

INVERTLINE

INVERTLINE draws a line of a "color" opposite that of its background.

FORMAT

INVERTLINE startPoint (x,y) endPoint (x,y)

NOTES

INVERTLINE needs four arguments -- the horizontal and vertical points for the start of the line and the horizontal and vertical points for the end of the line.

The inversion here amounts to turning off bits that are on and turning on bits that are off. As a result, INVERTLINE describes a light line on a dark background and a dark line against a light background.

EXAMPLE

```
1000 DRAWBOX 120, 30, 90, 140
1100 DRAWLINE 100, 60, 150, 60
1200 LOCATE 10, 60: PRINT "Drawline"
1300 INVERTLINE 100, 100, 150, 100
1400 LOCATE 10, 100: PRINT "Invertline"
1500 ERASELINE 100, 140, 150, 140
1600 LOCATE 10,140: PRINT "Eraseline"
1700 END
```

MOVEBOX

This statement copies an existing box to a second set of coordinates.

FORMAT

MOVEBOX topLeft (x,y) extent (x,y) destination topLeft (x,y)

NOTES

MOVEBOX copies, but does not erase, an existing box. If you need to give the illusion of movement you must follow your MOVEBOX statement with an ERASEBOX statement (the erase coordinates should be those of the original box).

EXAMPLE

```
1000 LET A=100: B=80: C=80: D=80
1100 DrawBox A,B,C,D
1200 FOR Pause=1 TO 200: NEXT Pause
1300 MoveBox A,B,C,D,200,80
1400 EraseBox A,B,C,D
1500 END
```

STACKMSG

This statement places a message in inverse video at the bottom of the screen.

FORMAT

```
STACKMSG "PromptString"  
[STACKMSG "PromptString"]
```

NOTES

STACKMSG takes only one parameter -- the string character that constitutes the prompt message. If you have a second STACKMSG statement, the message area will expand to hold both messages. NOTE: When writing two messages, place the first message second. For example, the messages in

```
1000 STACKMSG "Press ESC to exit"  
1100 STACKMSG "The StackMsg Program"
```

appear in reverse vertical order:

```
The StackMsg Program  
Press ESC to exit
```

STACKMSG messages remain on the screen for a split second. To make them stay longer, you can follow the STACKMSG statement with some kind of loop. The most common loop displays the prompt until someone presses a key. The message then disappears as program execution continues. NOTE: Do not include a STACKMSG prompt inside a loop unless you follow it immediately with a CLEARMSG statement. Otherwise, the prompt area scrolls up the screen. Adding a third and/or fourth message without CLEARMSG also causes scrolling. See CLEARMSG earlier in this chapter for details.

EXAMPLE

```
1000 STACKMSG "Press ESC to exit"  
1100 STACKMSG "The StackMsg Program"  
1200 LET Key$=INKEY$  
1300 IF Key$ <> "" THEN PRINT Key$; " gets me out of the loop" ELSE  
GOTO 1200  
1400 END
```

In this example, lines 1000 and 1100 set up a two-line prompt. Lines 1200 and 1300 create an INKEY\$ loop that waits for a key press to occur. When someone presses a key, line 1300 prints the key's character.

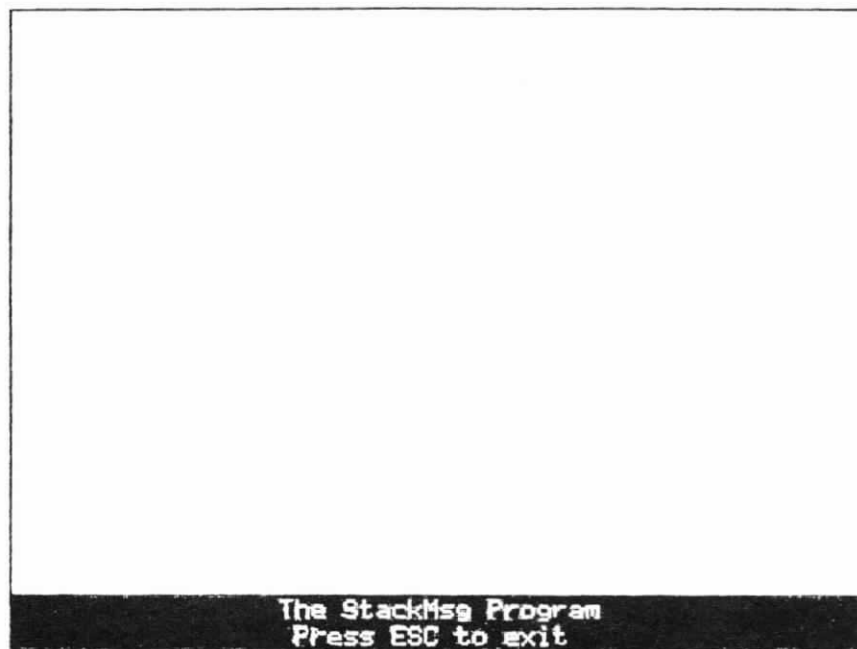


Figure 10-11. A STACKMSG Prompt Line

APPENDIX A: ERROR MESSAGES

GRiDBASIC error messages are listed here in alphabetical order.

Array is too large

What happened You tried to put over 65,535 bytes into an array.

What to do Redimension the array so that its size falls within legal limits.

Array reference is out of range

What happened You probably have a subscript of 0; you dimensioned an array with a variable and still haven't assigned a number to that variable. Or, you have assigned a number greater than the subscript allows.

What to do Check your subscripts, especially those that are variables. Remember: You can only have ten items in an array without dimensioning.

Attempt to read past end of file

What happened An INPUT# statement is executed after all the data in a file has already been input, or the file is a null (empty) file.

What to do Place the EOF function in your program to detect end

of file and avoid this error.

ELSE encountered without matching IF

What happened You programmed an IF THEN ELSE, but managed to leave out the IF.

What to do Put in the IF statement.

Empty line

What happened This is a system-level error.

What to do Nothing. You won't see this error.

Expression error

What happened This is a system-level error.

What to do Nothing. You won't see this error.

File already open

What happened This run-time error occurs when you try to reopen a file you've already opened.

What to do Check to see what file is open and its tag number. Also, is it a random access file? Sometimes you try to open a different file, but give a tag that is already in use.

You can either write a CLOSE statement for the file or (if the tag is the problem) give the correct number. If the file is random access, make sure you haven't tried to read or write to it as you would with a sequential file.

File is not open for random access.

What happened This run-time error indicates you've tried to read from or write to a file (with GET or PUT), but you haven't opened it as a random access file.

What to do Check to see if you've opened the file. If you have, check the file tag number and the number you've used with the your access command. Also, did you assign the file the "R" (for "random") mode? And remember:

Random access files require a FIELD statement to allot buffer space.

File not open

- | | |
|---------------|---|
| What happened | This run-time error indicates you've tried to read from or write to a file you haven't opened. |
| What to do | Check to see if you've opened the file. If you have, check the file tag number and the number you've used with the your access command. |

FOR encountered without matching NEXT

- | | |
|---------------|--|
| What happened | You began a FOR NEXT loop, but failed to complete it with a NEXT statement. |
| What to do | Locate the place where the loop should end and put in the NEXT statement with appropriate variable. Or erase the FOR TO [STEP] if you no longer want the loop in your program. |

Generation error

- | | |
|---------------|------------------------------------|
| What happened | This is a system-level error. |
| What to do | Nothing. You won't see this error. |

Illegal character

- | | |
|---------------|---|
| What happened | This error message is reserved for later use. |
| What to do | Nothing. It won't happen. |

Illegal value

- | | |
|---------------|---|
| What happened | A number that is either too large or too small causes this at the system level. |
| What to do | Nothing. You won't see this error. |

Improper expression

- | | |
|---------------|------------------------------------|
| What happened | This is a system-level error. |
| What to do | Nothing. You won't see this error. |

Improper function call

- What happened This message covers a multitude of sins -- from using non-existent or unimplemented functions.
- What to do Check to see that your program contains only current functions. Are their names correctly spelled? If everything looks okay, try re-running the program. If that fails, reboot the system and then re-run the program.

Improper loop nesting

- What happened You written an inner loop and an outer loop so that they overlap. A run-time error.
- What to do Untangle the offending loops so that no overlapping takes place. See FOR NEXT and WHILE WEND for details.

Improper parameter in function call

- What happened This run-time error usually indicates an improper number of parameters or parentheses.
- What to do Check to see that you gave the correct number of parameters and parentheses to the function.

Improper syntax

- What happened This is the catch-all phrase for any syntax problem. It occurs while programming, when you press RETURN, CODE-RETURN, or either vertical arrow key.
- What to do Check the syntax of all statements and functions on the current line and confirm the line again to see if the error remains.

Invalid variable

- What happened The interpreter has encountered either a variable with an illegal character in it (see Chapter Two) or the name of a file the system can't find.
- What to do Check the variables on the current line. If you are making a file reference, make sure that it is on the

device, under the subject, and of the kind you have named.

Mismatched quotes

- What happened This error occurs while you're programming and indicates you don't have the proper number of quotes on the current line.
- What to do Check quotation marks (") to make sure you have the correct number. NOTE: Don't try to put double quotes within double quotes. You can, however, put single quotes (') within double quotes.

Missing parameter in array reference

- What happened During programming, you have omitted one of the dimensions that you declared when dimensioning (DIM) the array.
- What to do Find the erring array and insert the missing parameter.

Missing parameter in function reference

- What happened While programming, you have omitted a required parameter from a function.
- What to do Find the function and determine which parameter is missing. Then insert the parameter.

NEXT encountered without matching FOR

- What happened You have failed to include the upper portion of the FOR NEXT loop -- FOR TO [STEP]. Or you have given the wrong variable after NEXT.
- What to do Put the FOR TO [STEP] portion of the loop at its proper place in your program. Or, if the wrong variable follows NEXT, correct it. Or, if the NEXT is an unwanted leftover, erase it.

Number of array dimensions disagrees with definition

- What happened You have given an array the wrong number of dimensions. This message can occur while programming or at run-time.

What to do Check any array(s) in the current line and determine which has an improper number of dimensions.

Not implemented

What happened You have used a word that GRiDBASIC has reserved for later use, but which does not yet work as a statement, function, or constant.

What to do Figure out some other way to accomplish the purpose achieved by the unimplemented word.

Number of parameters disagrees with definition

What happened You have given an array or a function the wrong number of parameters. This message can occur while programming or at run-time.

What to do Check any array(s) or functions in the current line and determine which has an improper number of parameters.

Out of memory

What happened You dimensioned an array so that it takes more memory than the system offers.

What to do Redimension the offending array.

Ran out of data

What happened A READ statement read all the available DATA items.

What to do Add more data. Or put in a counter that causes the program to stop reading before exhausting the items in the data statement. Or put in a RESTORE statement to cause the data to be reread.

RETURN encountered outside subroutine

What happened You have a RETURN statement that lacks a preceding matching GOSUB statement.

What to do Either write the appropriate GOSUB statement or erase the RETURN.

Statement with syntax errors encountered

- What happened This message repeats at run-time what you saw as "Improper syntax" while programming. This means that you didn't correct the error.
- What to do Check the syntax of all statements and functions on the current line and confirm the line again to see if the error remains.

Type mismatch

- What happened Every variable and most operations expect a particular "type" of data -- string, numeric, or Boolean. Giving a foreign datum to a variable or operation causes this error. For example, giving a string to a numeric operator or variable.
- What to do Find the offending datum (or its source). Then change either the datum or the receiving statment so that a type match occurs.

Undefined line number

- What happened You have placed a line number in a statement (such as GOTO or GOSUB) for which no matching line exists.
- What to do Change the line number to point to the proper line. Or erase the pointer statement.

Variable expected here

- What happened You type a statement requiring a variالبة (such as INPUT) while programming, but didn't include its variable.
- What to do Find the statement and enter the variable(s).

WEND encountered without matching WHILE

- What happened A WHILE WEND loop lacks its WHILE statement.
- What to do Either insert the WHILE statement with its condition or erase WEND statement.

WHILE encountered without watching WEND

What happened A WHILE WEND loop lacks its WEND statement.

What to do Either insert the WEND statement with its condition or erase WHILE statement.

APPENDIX B: ASCII CHARACTERS

This appendix contains the ASCII (American Standard Code for Information Interchange) character codes. Programmers use these codes in everything from string handling functions (see Chapter Six) to communications work.

DEC	HEX	GRPH	ABBR	NAME	PRESS
00	00	N	NUL	null	CTRL-SHIFT-2
01	01	S	SOH	start of heading	CTRL-A
02	02	T	STX	start of text	CTRL-B
03	03	E	ETX	end of text	CTRL-C
04	04	T	EOT	end of transmission	CTRL-D
05	05	E	ENQ	enquiry	CTRL-E
06	06	A	ACK	acknowledge	CTRL-F
07	07	B	BEL	bell	CTRL-G
08	08	S	BS	backspace	CTRL-H
09	09	H	HT	horizontal tab	CTRL-I, TAB
10	0A	L	LF	linefeed	CTRL-J
11	0B	V	VT	vertical tab	CTRL-K
12	0C	F	FF	form feed	CTRL-L
13	0D	C	CR	carriage return	CTRL-M
14	0E	S	SO	shift out	CTRL-N
15	0F	S	SI	shift in	CTRL-O
16	10	D	DLE	data link escape	CTRL-P
17	11	D	DC1	device control 1(XON)	CTRL-Q
18	12	D	DC2	device control 2	CTRL-R
19	13	D	DC3	device control 3(XOFF)	CTRL-S
20	14	D	DC4	device control 4	CTRL-T
21	15	N	NAK	negative ack	CTRL-U
22	16	S	SYN	synchronous idle	CTRL-U
23	17	E	ETB	end trans. block	CTRL-W
24	18	C	CAN	cancel	CTRL-X
25	19	E	EM	end medium	CTRL-Y
26	1A	S	SUB	substitute	CTRL-Z
27	1B	E	ESC	escape	CTRL-;
28	1C	F	FS	file separator	CTRL-SHIFT-;
29	1D	G	GS	group separator	CTRL-=
30	1E	R	RS	record separator	CTRL-SHIFT-;
31	1F	U	US	unit separator	CTRL-SHIFT-hyphen
32	20	S	SP	space	
33	21	!		exclamation	
34	22	"		quotation marks	
35	23	#		number sign	
36	24	\$		dollar sign	
37	25	%		percent sign	
38	26	&		ampersand	
39	27	'		apostrophe	
40	28	(		opening parenthesis	
41	29	)		closing parenthesis	
42	2A	*		asterisk	
43	2B	+		plus	
44	2C	,		comma	
45	2D	-		hyphen	
46	2E	.		period	
47	2F	/		slash	
48	30	0			
49	31	1			
50	32	2			
51	33	3			
52	34	4			
53	35	5			
54	36	6			
55	37	7			
56	38	8			
57	39	9			
58	3A	:		colon	
59	3B	;		semicolon	
60	3C	<		less than	
61	3D	=		equal to	
62	3E	>		greater than	
63	3F	?			
64	40	@		commerical at sign	

DEC	HEX	GRPH	ABBR	NAME	PRESS
65	41	A			
66	42	B			
67	43	C			
68	44	D			
69	45	E			
70	46	F			
71	47	G			
72	48	H			
73	49	I			
74	4A	J			
75	4B	K			
76	4C	L			
77	4D	M			
78	4E	N			
79	4F	O			
80	50	P			
81	51	Q			
82	52	R			
83	53	S			
84	54	T			
85	55	U			
86	56	V			
87	57	W			
88	58	X			
89	59	Y			
90	5A	Z			
91	5B	[		opening bracket	CODE-,
92	5C	\		backslash	CODE-SHIFT-,
93	5D	]		closing bracket	CODE-
94	5E	^		circumflex	
95	5F	_		underline	
96	60	`		back quote	CODE-'
97	61	a			
98	62	b			
99	63	c			
100	64	d			
101	65	e			
102	66	f			
103	67	g			
104	68	h			
105	69	i			
106	6A	j			
107	6B	k			
108	6C	l			
109	6D	m			
110	6E	n			
111	6F	o			
112	70	p			
113	71	q			
114	72	r			
115	73	s			
116	74	t			
117	75	u			
118	76	v			
119	77	w			
120	78	x			
121	79	y			
122	7A	z			
123	7B	{		left curly bracket	CODE-SHIFT-,
124	7C			vertical line	CODE-SHIFT-,
125	7D	}		right curly bracket	CODE-SHIFT-,
126	7E	~		tilde	CODE-,
127	7F	■	DEL	delete	CODE-SHIFT-hyphen



INDEX

A (Append) 8-14
ABS 5-3
Access mode 8-14
ACOS 5-4
AND 2-8, 5-5
Append access mode 8-14
Arrays 2-7, 3-2
AS 9-5
ASC 6-2
ASCII characters 6-2, B-1 ff
ASIN 5-6
ATN 5-7
Automatic line numbering 1-6

b0 8-1
Boolean Constants 5-13, 5-33
Buffer (RAM) 9-1, 9-5, 9-11

CDBL 5-8
CHR\$ 6-3
CINT 5-2, 5-9
CLEARMSG 10-4
CLOSE 8-3
CODE-? 1-6
CODE-C 1-3
CODE-E 1-3
CODE-R 1-3
COMMA 7-2, 7-6, 7-7, 7-10
Communications files 8-10
Concatenation 2-11, 7-2, 7-11, 8-11
Constants 2-4
 Boolean 5-13, 5-33
 Numeric 2-5
 String 2-4
Continuing a Program 1-3

Convert 9-3
Convert to double precision 5-8
Convert to integer 5-9
Convert to single precision 5-11
COS 5-10
Cosine 5-10
CSNG 5-11
Cursor 1-5
CVD 9-3
CVI 9-3
CVS 9-3

DATA 3-6
DATE\$ 7-4
Delimiters 2-13, 7-7
DIM 2-7, 3-2
DOMENU 10-5
DRAWBOX 10-6
DRAWCHARS 10-7
DRAWCIRCLE 10-9
DRAWDOT 10-10
DRAWLINE 10-11

e (exponential function) 5-12
ELSE 4-9
END 4-2, 8-3
E notation 2-5
EOF 8-4
EOLN 8-5
Epson 7-2 f, 8-18
ERASEBOX 10-12
ERASECIRCLE 10-14
ERASEDOT 10-15
ERASELINE 10-16
Erasing Line(s) 1-3

Error messages A-1 ff.
ESCAPE key 1-3
EXP 5-12
Exponential 5-12
Expressions 2-7

FO 8-1
FALSE 5-13
FIELD 9-5
Field variables 9-9
File access mode 8-14
File kinds 2-13
File naming conventions 2-12
file tag 8-3
FIX 5-2, 5-14
FOR 4-3
FOR TO NEXT [STEP] 4-3

GET 9-6
GETFILE\$ 2-12, 8-7
GOSUB RETURN 4-6
GOTO 4-8
Greater than (>) 2-8
Greater than or equal to (>=) 2-8
GRiDBASIC environment 1-4
GRiDBASIC, Invoking 1-1
GRiDBASIC screen 1-4

Highlighting 1-5

I (Input) 8-14
IF 4-9
IF THEN [ELSE] 4-9
INKEY\$ 7-5
INPUT# 8-8
INPUT\$ 8-10
INPUT 7-6
Input access mode 8-14
INSTR 6-4
INT 5-2, 5-15
Integer division 2-4, 2-8, 5-16
Integer functions 5-2
Integers 2-6, 5-2
INVERTBOX 10-17
INVERTCIRCLE 10-18
INVERTDOT 10-19
INVERTLINE 10-20

Keyboard 7-5
KILL 8-11

LEFT\$ 6-6
LEN 6-7
Less than (<) 2-8
Less than or equal to (<=) 2-8

LET 3-4
Line number field 1-4
Lines and line numbering 1-6
LOC 8-12, 9-7
LOCATE 7-9
LOF 8-13, 9-8
LOG10 5-18
Logarithm 5-17, 5-18
Logical operators 2-8 f, 5-5, 5-20, 5-21, 5-35
LOG 5-17
LSET 9-9

Make string function 9-10
Manual line numbering 1-6
Menu 10-5
Message line 1-5, 10-4, 10-22
Messages 10-4, 10-22
MID\$ 6-8
MKD\$ 9-10
MKI\$ 9-10
MKS\$ 9-10
MOD 2-8, 5-19
Mode
 Access 8-14, 9-11
 Direct 1-4
 Indirect 1-2
 Programming 1-2
MOVEBOX 10-21
Multiple statements 1-7

Nesting 4-4, 4-14
NEXT 4-3
NOT 2-8, 2-9, 5-20
Not equal to (<>) 2-8
Numeric constants 2-5
Numeric operators 2-7, 5-1

O (Output) 8-14
ON GOSUB 4-11
ON GOTO 4-11
OPEN (Random files) 9-11
OPEN (Sequential files) 8-1, 8-14
Operators 2-7
OR 2-8, 5-21
Outline 1-4
Output access mode 8-14

Padding digits 7-14
Pathname syntax 2-12, 8-7
PI 5-22
pixel 7-9, 10-1
Precedence, Order of 2-7, 2-8
PRINT# 8-17
PRINT# USING 8-19
PRINT 7-10
Printer 7-2 f, 8-18
PRINT USING 7-12
Program editing screen 1-2, 1-5
Prompt 10-5, 10-22
PUT 9-13

R (Random) 9-1
 Random access mode 9-1
 RANDOMIZE 5-23
 Random numbers 5-24 ff
 READ 3-5
 READ DATA [RESTORE] 3-5
 Real-time clock 5-23, 7-4, 7-21
 Reformatting listings 1-7
 Relational operators 2-8
 REM 3-9
 Renumbering 1-6
 Reserved words 2-3
 RESTORE 3-6
 RETURN 4-6
 RIGHT\$ 6-9
 RND 5-24
 ROUND 5-2, 5-28
 RSET 9-9
 Running a program 1-3

Screen dimensions 9-1
 Scrolling 1-2
 SEMICOLON 7-6, 7-7, 7-10, 7-19
 SGN 5-29
 SHIFT-RETURN 1-8
 SIN 5-30
 SPACE\$ 6-10
 SQR 5-31
 STACKMSG 10-22
 Statement field 1-4
 STEP 4-3
 STOP 4-13
 Stopping a program 1-3
 STR\$ 6-11
 STRING\$ 6-12
 String constants 2-4
 String operators 2-11
 Syntax diagrams 2-1

TAB 7-20
 Tabbing 7-2, 7-10
 TAN 5-32
 THEN 4-9
 TIME\$ 7-21
 TO 4-3
 TRUE 5-33
 TRUNC 5-2, 5-34

VAL 6-13
 Variables 2-6

W0 8-1
 WEND 4-14
 WHILE 4-14
 WHILE WEND 4-14
 Worksheet example 8-6

XOR 2-8, 5-35, 7-12

" 2-4, 7-10
 # 2-4, 7-14, 8-3, 8-14
 \$\$ 7-17
 \$ 2-4, 2-6, 6-1
 % 2-4, 2-6, 7-15
 & 7-13
 ' 3-9
 (2-4, 2-8
) 2-4, 2-8
 \$\$\$ 7-17
 \$\$ 7-16
 * 2-4, 2-8, 5-1, 7-17
 + 2-4, 2-8, 2-11, 5-1, 7-15
 - 2-4, 2-7, 2-3, 5-1, 7-15
 / 2-4, 2-8, 5-1
 , 7-2 ff, 7-14
 : 1-7, 2-4, 2-13
 ; 2-4, 7-19
 < 2-4, 2-8
 <= 2-8
 <> 2-8
 = 2-4, 2-8
 > 2-4, 2-8
 >= 2-8
 \ 2-4, 2-8, 5-16, 7-13
 ^ 2-5
 ' 2-4, 2-12, 2-13
 ! 10-5
 ~ 2-4, 2-12, 2-13

