GRiDBASIC REFERENCE MANUAL

March 9, 1983

Model Number 21020-40

2535 Garcia Avenue
Mountain View, CA 94043
(415) 961-4800

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

TABLE OF CONTENTS

CHAPTER TWO: GENERAL INFORMATION ABOUT GRiDBASIC

CHAPTER THREE: ASSIGNMENT AND DEFINITION STATEMENTS

CHAPTER FOUR: STATEMENTS THAT CONTROL PROGRAM FLOW

CHAPTER FIVE: ARITHMETIC AND LOGIC

CHAPTER SIX: STRING FUNCTIONS

CHAPTER SEVEN: INPUT/OUTPUT STATEMENTS

CHAPTER EIGHT: SEQUENTIAL FILES STATEMENTS

CHAPTER NINE: RANDOM FILE STATEMENTS

CHAPTER TEN: GRAPHICS STATEMENTS

APPENDICES

LIST OF FIGURES

LIST OF TABLES

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 disucssion 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.

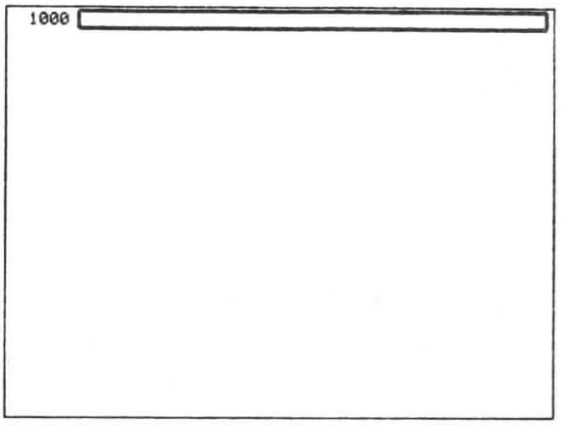

Figure 1-1. The Initial Program Editing Screen

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.

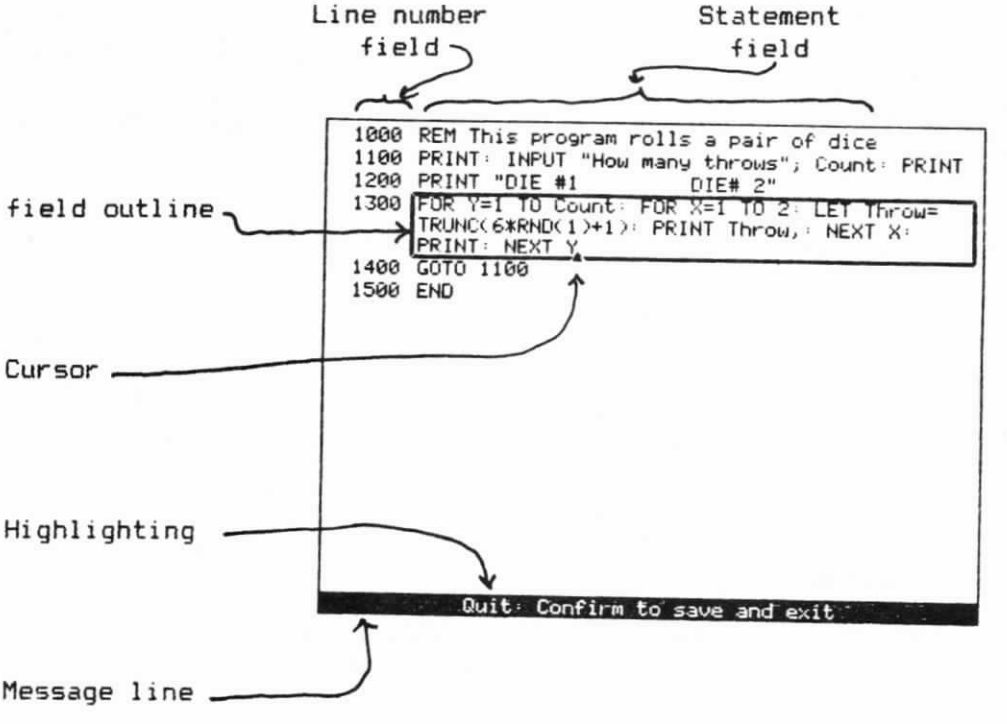

Figure 1-2. The Program Editor Screen

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.

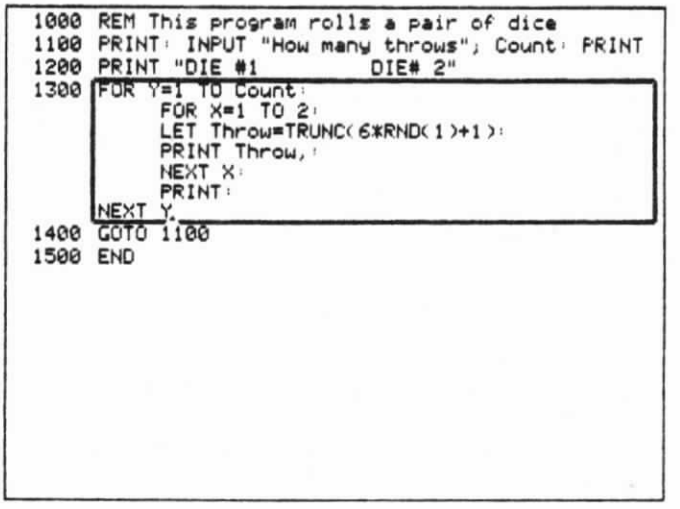

Figure 1-3. Line 1300 after Inserting SHIFT-RETURNs

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 multilple 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 statments 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"
"%@*&?"
"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 (%). GRiDBASIC 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 immediatley 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 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 excecution 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 be 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 pi). 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(expession)
```

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(expession)

NOTE$

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
1100 PRINT: PRINT
1200 FOR X = 1 TO Number
1300 PRINT, 10*RND(1)
1400 NEXT X
1500 PRINT: PRINT
1600 GOTO 1000
1700 END
```

```
 1.00343327992676
 4.02151522087434
 9.8295567254139
 7.92690928511482
 6.04562447547112
 1.51720454718853
 1.85885404745556
 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 occurence 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 before beginning its search. Include this expression when you want to move past the string just located to find another occurence 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.