BASIC/47

Preface

This manual provides detailed programming information for Intelligent Terminal BASIC. It is written to provide the information needed to use the BASIC interpreter to develop application programs to run on the terminal.

All material in this manual is applicable to both the HP 2647F and the HP 2647A terminals, except where specifically stated as applying to one terminal or the other.

This manual assumes that you are familiar with your terminal. Operating and programming information for the 2647F terminal is given in the manual titled *Getting Started with Your HP 2647F* (02647-90036) and the HP 2647F *Reference Manual* (02647-90037). Operating and programming information for the HP 2647A is contained in the prior versions of both manuals.

Other manuals documenting the 2647F are:

Word/47, part no. 13257-90019 — This manual describes the text editing and formatted output features available as options on the 2647F.

Forms/47, part no. 13257-90020 — This manual describes how to enter data in forms and how to create these forms using the Forms Design Mode feature, available as an option.

2647F Quick Reference Guide, part no. 02647-90039 — This pocket sized manual provides brief descriptions of terminal commands, functions, and escape sequences as well as a summary of BASIC and AGL syntax.

Autoplot/47, part no. 02647-90042 — This manual describes how to creat pie, bar, and linear charts and how to produce overhead transparencies using the AUTOPLOT/47 application program. It also provides detailed information on plotters and printers frequently used to obtain hardcopy output.

2647F Service Manual Supplement, part no. 02647-90043 — This manual lists replaceable part numbers, provides disassembly instructions, and gives test procedures for the terminal. This manual is available at extra cost.

If you are not familiar with programming or the BASIC language, you should refer to a tutorial manual explaining the BASIC language and the elements of programming. Manuals explaining BASIC are available in most bookstores. Note that the version of BASIC used with the terminal is slightly different than that described in most tutorial manuals.

How To Use This Manual

This manual describes the BASIC interpreter used in Hewlett-Packard Intelligent Terminals. The various functional statement groups such as strings, arrays, and files are described in separate sections. If you are already familiar with the BASIC programming language, you can refer to Section 12 which contains the detailed syntax of the various BASIC commands and statements. Section 10 contains many unique statements that allow you to directly access or modify the terminal operating system.

HP Computer Museum
www.hpmuseum.net

For research and education purposes only.

This manual is made up of the following sections and appendices:

Section 1. Introduction — This section provides a brief overview of the BASIC interpreter and describes how it is loaded and accessed.

Section 2. Data — This section describes the various data structures and formats used by the BASIC interpreter.

Section 3. Operators, Functions, and Expressions — This section describes the arithmetic and string operators as well as the various functions available in the interpreter.

Section 4. Statements — This section provides a brief tutorial description of some of the BASIC statements.

Section 5. Formatted Output — This section describes how to produce specially formatted output from your BASIC program.

Section 6. Arrays — This section describes how to access and modify array data.

Section 7. Strings — This section describes how to access and modify string or textual data.

Section 8. Subprograms — This section describes how to write subprograms.

Section 9. Files — This section describes how to store and access data stored in terminal device files.

Section 10. Terminal Operations — This section describes the special statements and functions that allow you to access the terminal's operating system from your BASIC program.

Section 11. A Graphics Language (AGL) — This section describes a special set of statements that allow you to perform graphics output to the terminal display or a graphics peripheral.

Section 12. BASIC Syntax — This section provides the detailed syntax for the BASIC commands and statements.

Appendix A. ASCII Character Set — This appendix lists the characters available in the ASCII character set together with their numeric code values.

Appendix B. Compatibility — This appendix describes the differences between BASIC/47 and other BASIC languages.

Appendix C. Reserved Words — This appendix lists letter groups that cannot be used as variable names.

Appendix D. Summary of BASIC — This appendix lists the commands and statements available in the interpreter.

Appendix E. Error Messages — This appendix lists all of the error messages generated by the terminal together with a description of the error.

Terms And Conventions

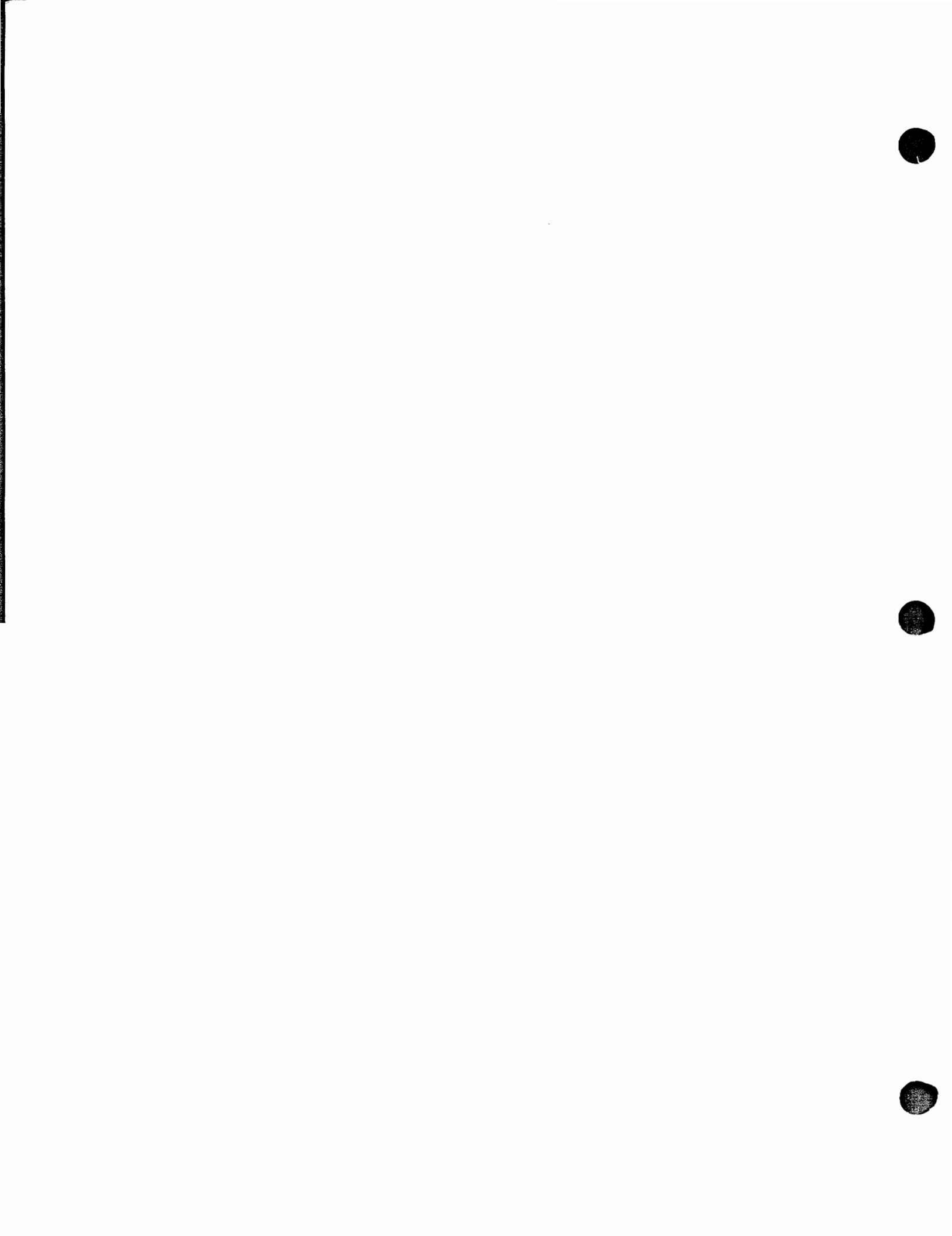The descriptions in this manual use the following conventions:

] [— The right and left bracket are used to enclose a parameter that is optional. The brackets themselves should not be entered.

Example: The RUN command is shown as RUN [line number]. This means that a line number is optional. To RUN a program at line 100, enter RUN 100.

< > — The less than and greater than signs are used to set off parameters. The less than and greater than signs themselves should not be entered.

Example: The SET command is shown as SET < condition>. This means that one of the <condition> parameters, SHORT, LONG, etc., should be used. To set the default data type to LONG, enter SET LONG.

Control Characters are indicated in three ways. They may be shown as a character with a superscript "C" (A^C), the word control followed by a letter (CONTROL-A), or as a two letter graphic symbol (ξ). The graphic symbols are listed in Appendix A.

CONTENTS

Section 12

BASIC SYNTAX

Appendix A

ASCII CHARACTER SET

Appendix B

COMPATIBILITY

Appendix C

RESERVED WORDS

Appendix D

SUMMARY OF BASIC

Appendix E

ERROR MESSAGES

Appendix F

GRAPHICS APPLICATIONS PROGRAMS

Index

Introduction

Computer
Museum

This section provides an overview of BASIC programming on the terminal. It briefly describes selected commands and statements used in elementary BASIC programming. Sample programming sessions will show you how to create, edit, run, and save BASIC programs. Specialized topics such as arrays, strings, files, formatted output, and graphics are discussed as separate topics in other sections. Section 12 contains detailed descriptions of each of the BASIC commands and statements.

What is BASIC/47?

BASIC/47 is a powerful version of the BASIC programming language. It allows you to write and execute application programs on your terminal. These programs can interact with programs on a host computer or run independently with the terminal off line.

BASIC/47 contains most of the standard BASIC statements together with special statements that allow you to monitor and control terminal operation. In addition, a complete set of AGL (A Graphics Language) statements are included. These statements provide a high level language for controlling the terminal's graphic functions. AGL statements are used in the same way as normal BASIC statements.

BASIC program statements present in the terminal's memory are executed by a BASIC interpreter. This interpreter allows your program to interact with the terminal's input/output file system. The file system gives your program access to printers, plotters, and other devices connected to the terminal. This is in addition to the terminal's own keyboard, display, flexible discs, cartridge tapes, and data communications capabilities (see figure 1-1).
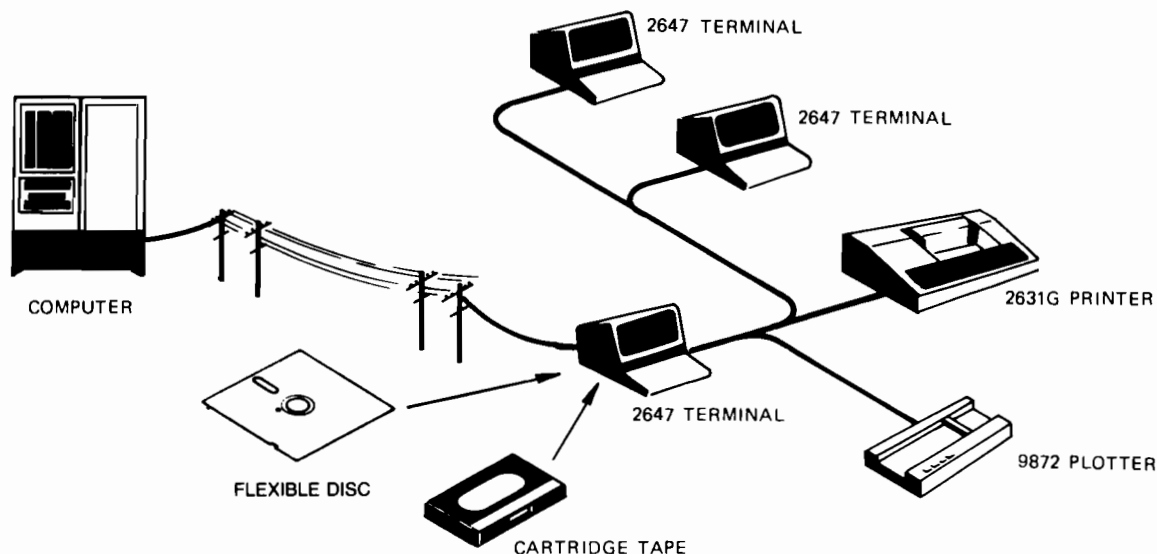
Figure 1-1. Terminal Network Capabilities

How Do You Use BASIC?

To use BASIC the interpreter must be present (loaded) in the terminal. Normally it will not be necessary to reload the interpreter unless the terminal has been turned off. (The interpreter is lost when the terminal is turned off.)

The normal procedure for using BASIC is to request the BASIC interpreter using the COMMAND channel, type in a program from the keyboard, or read a program from a flexible disc or cartridge tape, and then run the program.

Start the terminal by setting the power switch to ON. After about 20 seconds "TERMINAL READY" will appear in the upper left corner of the display screen.

Before you can enter a BASIC program, the BASIC interpreter must be present in memory. If the BASIC interpreter is already loaded you can start it by pressing the COMMAND key and entering BASIC ⬛RETURN. The terminal will display the available workspace and the BASIC prompt character (" > "). If the terminal responds with the message:

PROGRAM NOT FOUND or CHECKSUM ERROR IN BASIC

the BASIC interpreter is not present and must be loaded. (The error message can be cleared by pressing the ⬛RETURN key.) The procedure for loading the BASIC interpreter is as follows:

Step 1. Insert the BASIC/AUTOPLOT/47 disc in the drive.

Step 2. Make sure that the REMOTE key is in the "up" position. (The terminal must be set for local operation.)

The following display will appear on the screen:

```
        BASIC/AUTOPLOT/47

   (c) HEWLETT-PACKARD CO 1982

         02647 13401

       Rev A-2142-42
```

```
                                                          | LORD
   Select AUTOPLOT program f1 - f7                        | BASIC
  ┌─────┬─────┬──────┬──────┐    ┌─────┬──────┬──────┐ ┌──────┐
  | f1  | f2  |  f3  |  f4  |    | f5  |  f6  |  f7  | |  f8  |
  | PIE | BAR |LINEAR|LOG/LOG|   |Y LOG|X-LOG | SLIDE| |BASIC |
  └─────┴─────┴──────┴──────┘    └─────┴──────┴──────┘ └──────┘
```

Figure 1-2. How to Load BASIC

Step 3. Press softkey **f8** , corresponding to the key labeled BASIC which appears in the display across the bottom of the screen as shown below. When the BASIC interpreter has been loaded, the BASIC welcome message below will appear.

> **NOTE:** If you want to use one of the AUTOPLOT/47 programs, press a softkey from **f1** through **f7** to load the program labeled on that key. The BASIC interpreter is automatically loaded along with the program selected (unless BASIC is already present in the memory). See the HP 2647F AUTOPLOT/47 Manual (part number 02647-90042) for instructions for running AUTOPLOT/47.

```
TERMINAL READY
Hewlett-Packard BASIC/47 Interpreter Rev. A-2142-42

HP BASIC/47 - REV. A-2141-42
 XXXXX BYTES STACK SPACE
 XXXXX BYTES WORK SPACE
>
```

You can now begin entering your BASIC program. Pressing the **HOME** and **CLEAR DSPLY** keys will clear the display. If you want to return to normal terminal operation, enter "EXIT" **RETURN** following the BASIC prompt ("**>**"). Note that if the terminal is subsequently turned off, BASIC must be reloaded using steps 1 through 3. If you simply "exit" BASIC and do not turn the power off, you may reenter BASIC at any time by typing BASIC after the asterisk in the command channel. (Press the **COMMAND** key to bring the command channel onto the screen. Press it again to remove it.)

Once you are in the BASIC interpreter you can use a wide selection of commands and editing features to aid you in program preparation.

Commands and Statements

The BASIC interpreter accepts commands and statements. BASIC commands instruct the interpreter to perform some action on your program or modify the workspace or interpreter operation. Commands differ from BASIC statements in both purpose and form. A command causes the interpreter to perform some action immediately. A statement is an instruction to perform a particular function only when the program containing it is run. A statement must always have a line number while a command never does. Commands can be entered at any time except when the current program is executing. Commands are either executed immediately or rejected with an appropriate error message. Additional information on statements is given later in this section under "Programs". Refer to the 2647F *Quick Reference Guide* (part number 02647-90039) for a summary of syntax parameters for both commands and statements.

Each command is a single word. Some commands also have optional or required parameters following them. If parameters are used, they are separated from the command by a space. Multiple parameters are separated from each other by commas.

The line number range in the LIST, SAVE, CSAVE, and DELETE commands is expressed with use of the slash (/) or hyphen (-).

For example, to list lines 10 through 80 of a program enter:

LIST 10-80 or LIST 10/80

Command entry is terminated by pressing the ⬛**RETURN** key. If the command is misspelled or otherwise unrecognized, the interpreter will respond with the message "SYNTAX ERROR".

Many of the commands will not produce a response on the display. Their completion is indicated by the prompt ("›") character. Others display one or more lines of information. In these cases you can stop command operation with the Break character (normally Control-A). The appearance of the "›" prompt indicates that the interpreter is ready to accept another command or statement.

The following paragraphs describe some of the commands used to manipulate programs in the BASIC workspace. Section 12 contains a full description of all BASIC commands.

Workspace

The BASIC interpreter has a workspace which is set aside for the user. When statements are typed at the terminal keyboard, they are entered into this workspace and become part of the current BASIC program. The workspace also holds all of the variables used by the program. Whenever the BASIC interpreter is loaded, this workspace is cleared.

To determine the size of the workspace available for your program on a 2647A, refer to "PROGRAM CONTROL" in Section 10.

If you require additional workspace, the way to get it will differ depending on whether you have an HP 2647F or HP 2647A terminal. To obtain additional workspace on an HP 2647F terminal, shorten the program or change the program structure to require less variable storage. On an HP 2647A terminal, you can adjust the size of the workspace by using the SET SIZE or REMOVE command (refer to Section 12). Increasing the workspace on an HP 2647A decreases the memory space available for display memory.

You can control the workspace in a variety of ways. You can change, display, renumber, delete, or save, program statements. To illustrate these capabilities, enter the following statements:

```
10 REM...FIND THE PRODUCT
20 PRINT "ENTER TWO NUMBERS"
30 INPUT B,C
40 LET A=B*C
50 PRINT A
60 END
```

To change line 20, simply re-enter the line:

```
20 PRINT "ENTER TWO NUMBERS SEPARATED BY A COMMA"
```

To list the changed program, enter the LIST command:

```
LIST RETURN
10 REM...FIND THE PRODUCT
20 PRINT "ENTER TWO NUMBERS SEPARATED BY A COMMA"
30 INPUT B,C
40 LET A=B*C
50 PRINT A
60 END
```

Note that when you enter program statements, spaces are not required except within quoted text. You can omit spaces from the BASIC statements or put in more than one space. When the program is listed, the interpreter shifts all BASIC statements to upper case (only the first letter in variable names) and uses a standard spacing between terms. Also if you have entered statements out of order, the interpreter will always list them in numeric order.

To add lines, you can insert them by using line numbers between those already used. For example, to insert a statement between lines 50 and 60:

```
55 IF A>0 THEN 70
```

At least one statement must now be added at line 70:

```
70 LET X=A/2
71 PRINT "A DIVIDED BY 2 IS";X
72 END
```

The RENUM command allows you to renumber lines in the workspace. When no parameters are used, this command renumbers each line starting at 10 and incrementing each line by 10. It will automatically adjust any line number references used in program statements to the new line numbers. Renumber and list the edited program. Note that the line entered as 55 is now line 60 and references line 80.

```
RENUM RETURN
LIST RETURN
10 REM...FIND THE PRODUCT
20 PRINT "ENTER TWO NUMBERS SEPARATED BY A COMMA"
30 INPUT B,C
40 LET A=B*C
50 PRINT A
60 IF A>0 THEN 80
70 END
80 LET X=A/2
90 PRINT "A DIVIDED BY 2 IS";X
100 END
```

You can renumber lines starting at a line number other than 10 and using any increment you want. For example, to start with line number 1 and increment by 5, enter:

```
RENUM 1,5 [RETURN]
LIST [RETURN]

1 REM...FIND THE PRODUCT
6 PRINT "ENTER TWO NUMBERS SEPARATED BY A COMMA"
11 INPUT B,C
16 LET A=B*C
21 PRINT A
26 IF A>0 THEN 36
31 END
36 LET X=A/2
41 PRINT "A DIVIDED BY 2 IS";X
46 END
```

You can also renumber only a portion of your program. Refer to Section 12 for additional RENUM parameters that provide this feature.

You can delete any line from your program by entering the line number and a RETURN. If you want to delete a group of lines, you can use the DELETE command. For example, to delete lines 26 through 41, enter:

```
DELETE 26-41 [RETURN]
```

If the last number to be deleted is also the last line in the program, you need only enter the starting line number to be deleted, followed by a dash. The DELETE command will then delete all lines from the starting line specified to the end of the program. If you now renumber your program and list it, you will be back to the version of the program that you originally entered.

```
RENUM [RETURN]
LIST [RETURN]

10 REM...FIND THE PRODUCT
20 PRINT "ENTER TWO NUMBERS SEPARATED BY A COMMA"
30 INPUT B,C
40 LET A=B*C
50 PRINT A
60 END
```

So far the LIST command has been used to list the entire program. It can also be used to list selected portions of the program by specifying the first and last lines to be listed. If only one number is entered following the LIST command, only that single line will be listed.

Example: List lines 30 through 50.

```
LIST 30-50 [RETURN]

30 INPUT B,C
40 LET A=B*C
50 PRINT A
```

Example: List only line 30.

LIST 30 `RETURN`

30 INPUT B,C

At any time when you are preparing a program, you can run it to check for problems. This is done by entering the RUN command. Your program will begin executing with the first line number.

RUN `RETURN`

ENTER TWO NUMBERS SEPARATED BY A COMMA
?375,4 `RETURN`
1500

An executed program will remain in the workspace and can be run again as desired until it is either removed using the DELETE or SCRATCH commands or until a new program is loaded. The program will be lost if the terminal is turned off or if BASIC is reloaded, but not if BASIC is exited and reentered.

You can save a program on either disc or tape using the SAVE command. The SAVE command writes a copy of the program in the workspace into a designated file.

For example, to save the program which is currently in the workspace, proceed as follows:

Disc

1. Insert an unprotected (no tab covering side notch) disc into the drive. Make sure the disc is already named and formatted.

2. Enter SAVE "<filename>" `RETURN` after the BASIC prompt.

3. You may verify the operation by pressing:

`COMMAND` `f3` SHOW, `f3` FILES, `RETURN`

to check the file listing.

You can reload the program using the GET command and can follow that with a LIST or RUN command following the next BASIC prompt. On the 2647F, you may also enter RUN "<filename>" folllowing the BASIC prompt, eliminating the need to GET the program first.

Tape

1. Put a blank cartridge tape in the left tape slot. Make sure the tab on the tape cartridge is set to the RECORD position.

2. Record the program onto the cartridge tape by entering:

SAVE "L" `RETURN`

3. To rewind the tape, press:

`COMMAND` , `f5` , `f5` , `RETURN`

4. When the green light by the tape slot stops blinking, press:

`COMMAND`

You can reload the program from the cartridge tape using the GET command.

The GET command reads a program from a specified cartridge tape and enters it into the workspace. The GET command clears any program that is in the workspace before entering the new program from tape.

For example, to get the program which you have just stored on the cartridge tape, proceed as follows:

1. Transfer the information from the left tape into the workspace by typing:

`GET "L" RETURN`

2. List what is now in the workspace.

```
LIST RETURN
10 REM...FIND THE PRODUCT
20 PRINT "ENTER TWO NUMBERS SEPARATED BY A COMMA"
30 INPUT B,C
40 LET A=B*C
50 PRINT A
60 END
```

The MERGE command can be used to combine two programs. The first program is loaded into the workspace. The second program is then loaded from tape or disc using the MERGE command. The second program is added to the lines already in the workspace just as if it was entered from the keyboard. Note that if any of the line numbers in the program on tape are the same as those in the workspace, the line on tape will replace the one in the workspace.

The MERGE command is useful for incorporating standard subroutines into a program being developed in the workspace. The RENUM command can be used to adjust line numbers so that the program segments can be easily merged.

The SCRATCH command is used to clear the workspace. You can enter SCR or SCRATCH. You should use the SCRATCH command to delete programs from the workspace before entering a new program from the keyboard. If you do not, a few lines from the earlier program may remain and cause program errors.

For example, to clear the workspace before going on to the next program, Enter:

`SCR RETURN`

Programs

Programs control the operations that you want performed. They can perform arithmetic functions, manipulate text data, or control devices.

A BASIC program is a sequence of BASIC language statements. The lines containing statements are numbered to indicate the order in which the statements are to be executed. Line numbers also allow one statement to reference another. The following BASIC statements make up a simple program that calculates a salary given the hours worked and an hourly wage.

```
10 PRINT "WEEKLY PAY CALCULATOR"
20 INPUT Hours,Wage
30 IF Hours = 0 THEN STOP
40 LET Pay = Hours * Wage
50 PRINT Hours;"hours at $";Wage;"per hour = $";Pay
60 GOTO 20
70 END
```

When this program is executed, it prompts you for values for the Hours and the Wage by displaying a question mark ("?") on the screen. If you enter 40 for the Hours and 3.00 for the Wage (40,3), the output is:

```
40 hours at $ 3 per hour = $120
```

The program stops when the number of hours given is 0. To stop the program enter 0,0.

The lines can be entered in any order. A good practice is to number them by tens so that in the future additional lines can be inserted easily. The computer puts the lines in numerical order no matter how they are entered. For example, if lines are entered in the sequence 10,40,30,50,20; the BASIC interpreter arranges them in the order 10,20,30,40,50. Line numbers can range from 1 through 9999.

Line numbers can be automatically generated using the AUTO command. See Section 12 for a description of the AUTO command.

You can make a second copy of a program line by listing the line, typing over the line number with the line number for the second copy and pressing RETURN. Remember that this does not delete the original line.

You can assign more than one statement for a single line number by separating statements on a line with the backslash ("\") character.

Example:

```
10 REM THIS PROGRAM PRINTS HI BYE
20 DIM A$[80],B$[80] \ A$="HI" \ B$="BYE" \ PRINT A$;B$
30 END
```

To continue a statement on more than one line, end the line with an ampersand (&). The BASIC interpreter will generate a double prompt (>>) for the additional input. Lines longer than approximately 250 bytes will trigger the error message BUFFER OVERFLOW. When the line is listed, the ampersand will appear as the 79th character in the line.

```
>20 REM ANY LINE WHICH ENDS WITH AN &
>>CAN BE CONTINUED ON ANOTHER LINE.   &
>>BASIC WILL RETURN A DOUBLE PROMPT &
>>WAITING FOR THE NEXT LINE.
>
>LIST
20 REM ANY LINE WHICH ENDS WITH AN CAN BE CONTINUED ON ANOTHER LINE. BASIC W &
ILL RETURN A DOUBLE PROMPT WAITING FOR THE NEXT LINE.
>

>100 IF Entry<0 THEN &
>>PRINT "VALUE ENTERED MADE POSITIVE"\ &
>>Entry=ABS(Entry)
>list
100 IF Entry<0 THEN PRINT "VALUE ENTERED MADE POSITIVE"\Entry=ABS(Entry)
>
```

Entering Programs

Once BASIC is loaded and enabled you can do one of the following:

- Enter a program from the keyboard.
- Enter a program from disc.
- Enter a program from tape.
- Enter a program from a remote computer.

Entering Programs from the Keyboard

The terminal should be set for local operation (REMOTE key up) during program generation. Otherwise keyboard input will be sent to the host computer as well as the interpreter.

When data is input to a program from the keyboard, BASIC does not process the data until the █████ key is pressed. If the BASIC statement is entered without a line number it will be executed immediately. (Refer to Direct Computation.) If the statement has a line number it will be stored in memory. When the █████ key is pressed, the display line that contains the cursor is processed. You are free to edit the data in any way you wish before pressing the █████ key. This means that if you enter data and then move the cursor to another line before pressing the █████ key, the wrong data will be entered. Refer to the descriptions of the INPUT and LINPUT statements for additional information on keyboard input.

Example: Write a program that adds up the cost of items purchased and calculates a 6% sales tax on the total.

Step 1. If you are not already in the BASIC interpreter, call it as described under "HOW DO YOU USE BASIC?".

Step 2. Enter the following program from the keyboard:

```
10 REM Add up the cost of items sold
20 REM and then compute sales tax
30 INPUT "Enter item cost $",Item
40 Price=Price+Item
50 IF Item>0 THEN 30
60 TAX=INT(6*Price)/100
70 PRINT LIN(1);"Cost =$";Price
80 PRINT "Tax = $";Tax
90 PRINT LIN(1);"Total = $";Price+Tax
100 END
```

Step 3. Type "RUN `RETURN`".

The program will output a message. Enter the values:

```
95.96  RETURN
33.00  RETURN
17.95  RETURN
0      RETURN
```

You should see the following display:

```
>run RETURN

Enter item cost  $95.96
Enter item cost  $33.00
Enter item cost  $17.95
Enter item cost  $0

Cost  = $ 146.91
Tax   = $    8.81

Total = $ 155.72
>_
```

Step 4. Type "RUN `RETURN`" again and enter your own values. Enter a "0" to total your entries and end the program. (Note that this simple program will not print trailing zeros after the decimal point.)

Loading Programs from Flexible Discs

A program previously saved on a disc can be loaded into the BASIC workspace by using the GET command followed by the filename in quotes in response to the BASIC prompt. The last program entered into the workspace remains loaded in the interpreter until it is scratched or the terminal is turned off. It is ready to be run.

On the 2647F you have the option of typing RUN "<filename>" following the BASIC prompt, eliminating the GET "<filename>" step. Both RUN "<filename>" and GET "<filename>" delete the current program in the workspace.

Example: Write a program to compute and plot SIN(X)/X for values of X ranging from 0 to 40. Store and retrieve the program using a disc file.

Step 1. Enter the BASIC interpreter. Type SCR [RETURN]. This will clear any old program from the BASIC workspace. Enter the following program statements:

```
10 REM Plot SIN(X)/X
20 PLOTR
30 LOCATE (100,180,50,100)
40 SCALE (0,50,-1,1)
50 FXD (2,2)
60 LGRID (5,.2,0,0,2,2)
70 FOR X=.01 to 40 STEP .1
80 PLOT (X,SIN(X)/X)
90 NEXT X
100 END
```

Step 2. Record the program on a file by entering SAVE "GRAPH" [RETURN] . Press:

[COMMAND] , [f5] , [f6] , [RETURN]

Step 3. Press the COMMAND key.

Step 4. Load the SIN(X)/X program by entering GET "GRAPH" [RETURN] . The program will be read into the workspace.

Step 5. Type "RUN [RETURN]". The program will execute and give the following display:
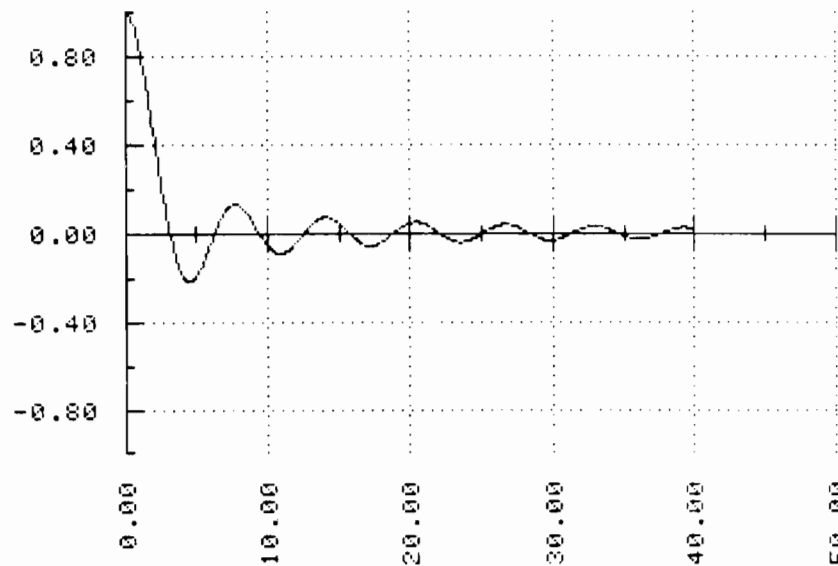
NOTE: To clear the graphics display press the SHIFT and CLEAR keys simultaneously.

[SHIFT] [CLEAR]

Loading Programs from Cartridge Tape

A program stored on a cartridge tape can be entered by placing the tape in the left drive and typing GET ▮▮▮. This causes the current file on the tape to be loaded into memory.

You can also load programs from the right tape drive. The left tape is selected unless you tell BASIC to read from the right tape drive. Entering GET **"R"** would cause the program to be loaded from the right tape drive.

If you select the wrong tape drive or if there is no tape in the selected drive you will get a NO TAPE error message. If there is a tape present, whatever is on the tape will be loaded as if it were your program.

Refer to the 2647A *User's Manual* for additional information on locating files and selecting tape drives.

Loading Programs from a Computer

Programs can be transferred from a computer to the terminal's BASIC interpreter either directly or indirectly. To transfer programs directly, simply access the terminal's BASIC interpreter and then have the computer list the program to the terminal. The exact procedure will vary depending on your computer system but it will be similar to the following:

1. Log onto your computer system.

2. Run the program or utility that is to transfer the program listing. This may be an editor, file transfer program, or a BASIC interpreter.

3. Enter the command that will cause the transfer or listing to take place, but do not enter in the final carriage return or other terminating character. This will keep your command from executing.

4. Release the REMOTE key. This will return the terminal to LOCAL operation.

5. Access the terminal's BASIC interpreter. (▮▮▮▮ , BASIC ▮▮▮▮ .)

6. Enter the SCR command to clear the terminal workspace.

7. Press the REMOTE key down and enter the carriage return or other terminating character to complete your computer command. This will cause your program to be listed to the terminal screen and to be entered into the terminal's BASIC workspace.

You can then release the REMOTE key (LOCAL operation), and exit BASIC/47. Once the terminal has been returned to normal operation, again press the REMOTE key and log off of the computer system. Your program is still present in your terminal. You can run the program by accessing BASIC/47 and entering the RUN command.

Programs can also be loaded indirectly by using a computer editor, utility program, or BASIC interpreter to list the program after placing the terminal in "Data Logging" mode. This will cause the program to be recorded onto a disc or tape. The file can then be entered using the procedures explained under "Loading A Program From A Flexible Disc." Refer to the *2647F Reference Manual,* Section 8, for an explanation of Data Logging.

Editing Programs

It is very easy to make changes to a program stored in the terminal. You can use any of the normal terminal editing features:

- character delete
- character insert
- character overstrike

- line delete
- line insert
- line replace

To change a statement, simply make the needed modifications to the statement on the display and press the ▮RETURN▮ key. The new line will be entered, replacing the old line. (If you wish to retype the entire line you can do this as well.)

Remember that when you press the ▮RETURN▮ key the entire line is read. Make sure that the line you enter contains only those characters that you want read. If there are any extra characters displayed in the line, they will be read also and may cause an error.

Example: Using the plot SIN(X)/X program, change the step size to .4.

Step 1. List the program and move the cursor to the line containing statement 70.

Step 2. Move the cursor to the step size (.1). The cursor should be under the "1".

```
>70 FOR X=.01 TO 40 STEP .1
```

Step 3. Overstrike the "1" with a "4". The line should now appear as follows:

```
>70 FOR X=.01 TO 40 STEP .4
```

Step 4. Press the ▮RETURN▮ key. This will cause the edited line to be entered, replacing the old statement 70.

You can now rerun the program with the new step size by typing RUN ▮RETURN▮

Running Programs

To run a program, enter "RUN" followed by ▮RETURN▮. Syntax and other programming errors are checked at the time a program is run. If an error is detected in your program, a message will be displayed to inform you of the cause of the error. A list of error messages is given in Appendix E.

If the terminal is set for local operation (REMOTE key up), the BASIC program cannot request input from, or print data to, the datacomm line using the terminal's file system (PRINT #, LINPUT #, etc.).

If the terminal is set for remote operation, the BASIC program can interact with the host computer, requesting input and sending output.

A running BASIC program can be halted by entering the terminal break character (normally CONTROL A). This causes a program break. You can then display and make changes to the values of variables using "direct mode" techniques. You can resume program execution from the point at which the break occurred by entering the GO command.

Example: Write a program to compute the square roots of a string of numbers.

Step 1. Clear the workspace by entering the SCRATCH command.

    SCR **RETURN**

Step 2. If there are graphics on the screen, clear the graphics display by pressing:

    SHIFT **CLEAR**

Step 3. Enter the following program:

```
10 INPUT X
20 LET Srx=SQR(X)
30 PRINT "THE SQUARE ROOT OF";X,"IS";Srx
40 LET X=X+1
50 GOTO 20
60 END
```

Step 4. Execute the program by entering the run command. When the question mark appears on the screen, type in the number 1.

    RUN **RETURN**
    ? 1

Step 5. The following output will begin running on your screen:

```
THE SQUARE ROOT OF 1   IS 1
THE SQUARE ROOT OF 2   IS 1.41421
THE SQUARE ROOT OF 3   IS 1.73205
THE SQUARE ROOT OF 4   IS 2
THE SQUARE ROOT OF 5   IS 2.23607
THE SQUARE ROOT OF 6   IS 2.44949
            .
            .
            .
```

Step 6. Stop the program by pressing:

    CNTL **A**

Step 7. To see what value of X is currently being computed, enter:

    PRINT X or ?X **RETURN**

Step 8. To change the value of X to 100 enter:

    Let X=100 **RETURN**

Step 9. To continue the program, enter:

    GO **RETURN**

You can repeat this process of finding the value of X and/or assigning a new value to X as often as you like. To terminate the program, press **CNTL** **A**.

When a program finishes execution (STOP or END statement), the final values assigned to program variables are available for direct computation. The values for these variables are lost when you enter the next RUN command or modify a program statement.

Saving Programs

After generating or modifying your program, you can save it on a flexible disc or cartridge tape. The SAVE command is used to store a copy of the current BASIC program to a designated file. The filename selected must not already exist on the disc. CAUTION: A program saved without a filename will be saved as a CTUR file. (See Files, Section 9.)

Disc

Step 1. Insert an unprotected (no tab on side notch) disc into the drive. Make sure the disc is already named and formatted.

Step 2. Enter SAVE "<filename>" RETURN in response to the BASIC prompt.

Step 3. You may verify the operation by pressing:

COMMAND , f3 SHOW, f3 FILES, RETURN .

Tape

Step 1. Insert a cartridge tape in either tape slot. Make sure the tab on the cartridge is in the record position.

Step 2. Assign the tape as the destination device.

COMMAND , f4 , f5 or f6 , RETURN

Step 3. Get out of the command channel by pressing:

COMMAND

Step 4. Type "SAVE" followed by RETURN . Your program will be stored on the cartridge tape. Since the program is copied from your workspace and not from the display, you do not need to list your program or modify the display before saving your program.

Using An Execute File

You can control the loading and execution of a BASIC program using a terminal Execute File. The Execute File can contain terminal and BASIC commands as well as a BASIC program. (Additional information on Execute Files is given in the *2647F Reference Manual*, Section 5, part number 02647-90037.) The following is a simple example of an Execute File.

Disc

Step 1. Enter the SCRATCH command to clear the BASIC workspace.

SCR RETURN

Step 2. Take the terminal out of BASIC by typing:

EXIT **RETURN**

The following will appear on the screen:

TERMINAL READY

Step 3. Insert an unprotected (no tab in notch) disc. Make sure disc is already named and formatted.

Step 4. Assign a filename as the destination device, press:

COMMAND , **f4** ASSIGN, **f4** more, **f6** D:FILE
SQR, **RETURN** .

Step 5. Take the terminal out of the command channel by pressing: **COMMAND**

Step 6. Clear the display screen by pressing the **HOME** and **CLEAR DSPLY** keys.

To clear graphics from the display screen, press the **SHIFT** and **CLEAR** keys simultaneously.

Step 7. Enter the following program on the screen:

```
BASIC
SCR
10 COMMAND "SUSPEND COMMAND FILE"
20 INPUT I
30 IF I=0 THEN 60
40 PRINT SQR(I)
50 GOTO 20
60 COMMAND "RESUME COMMAND FILE"
70 END
RUN
EXIT
```

Step 8. To transfer information from the display screen to the disc, press:

RECORD

Step 9. Press **COMMAND** ; enter EXECUTE SQR **RETURN**

Tape

Step 1. Enter the SCRATCH command to clear the BASIC workspace.

SCR **RETURN**

Step 2. Take the terminal out of BASIC by typing:

EXIT `RETURN`

The following will appear on the screen:

TERMINAL READY

Step 3. Insert a cartridge tape in the left tape slot. Make sure the tab on the cartridge is in the record position.

Step 4. Assign the left tape drive as the destination device, press:

`COMMAND` , `f4` , `f5` , `RETURN`

Step 5. Take the terminal out of the command channel by pressing:

`COMMAND`

Step 6. Clear the display screen by pressing the "home up" `▨` and `CLEAR DSPLY` keys.

To clear graphics from the display screen press the `SHIFT` and `CLEAR` keys simultaneously.

Step 7. Enter the following program on the screen:

```
BASIC
GET "L"
10 COMMAND "SUSPEND COMMAND FILE"
20 INPUT I
30 IF I=0 THEN 60
40 PRINT SQR(I)
50 GOTO 20
60 COMMAND "RESUME COMMAND FILE"
70 END
RUN
EXIT
```

Step 8. To transfer the information from the display screen to the cartridge tape, press:

`RECORD`

Step 9. Rewind the tape by pressing:

`COMMAND` , `f5` , `f5` , `RETURN`

Step 10. While still in the command channel enter "Execute L `RETURN`".

This file, on either disc or tape, will cause the terminal command "BASIC" to be read from the disc or tape. The commands that follow are then acted on by the BASIC interpreter. The SCR command clears the workspace so that the numbered lines from this file can be loaded. The RUN command starts program execution. The "SUSPEND" in statement 10 turns off the command file execution while the BASIC program is running. You can now enter any positive number, and the program will print the square root of the number.

To end the program, enter 0. This causes statement 60 to be executed. Statement 60 turns on the command file and reads the EXIT command from the file. You then leave BASIC and return to normal terminal operation.

Remote Operation

Refer to the description of the ASSIGN statement in Section 12 and to Section 10 under Data Communications for information on remote operation.

Sample Program Session

The following paragraphs will show you how to enter and run BASIC programs. These examples do not cover all of the terminal's programming features, nor do they teach you how to write a BASIC program. You should have read the manual titled *Getting Started with Your HP 2647F* and be familiar with the terminal keyboard.

The remainder of this manual assumes that you have experience with a high level programming language such as BASIC or FORTRAN. If you were able to read and understand the programs listed above, then you will be able to understand the following program. If not, you should probably first read a manual or text intended for beginning BASIC programmers.

Following the examples is a list of some of the BASIC statements and functions. If you are an experienced BASIC programmer and are developing fairly simple programs, this first section may contain all the information that you need. Refer to other sections of this manual for more detailed explanations of all of the BASIC commands, statements, and functions available in the terminal.

Access BASIC and clear the work area with the SCR command.

[COMMAND], BASIC [RETURN]
SCR [RETURN]

Enter the "Weekly Pay Calculator" program shown earlier in this section under "Programs." Figure 1-3 shows how the display should appear after the sample program has been entered.

```
Hewlett-Packard BASIC/47 Interpreter Rev. A-2142-42

HP BASIC/47 - REV. A-2141-42
 XXXXX BYTES STACK SPACE
 XXXXX BYTES WORK SPACE
 >SCR
 10 PRINT "WEEKLY PAY CALCULATOR"
 20 INPUT Hours,Wage
 30 IF Hours = 0 THEN STOP
 40 LET Pay = Hours * Wage
 50 PRINT Hours;"hours at $";Wage;"per hour = $";Pay
 60 GOTO 20
 70 END
```

Figure 1-3. Weekly Pay Calculator Program.

Once you have loaded the sample program, enter "RUN `RETURN`". This will cause the program to be executed. If you have entered the sample "Weekly Pay Calculator" program properly, the screen will appear as shown:

```
>run RETURN
WEEKLY PAY CALCULATOR
?
```

Enter 40 for the number of hours worked and 3.00 for the hourly wage.

```
? 40,3 RETURN
 40 hours at $ 3 per hour = $ 120
?
```

The above display shows how the screen might look after execution of the sample program. The number of hours entered first was 40 and the hourly wage was 3.00. The second time around use 20 for the number of hours and an hourly wage of 3.25.

```
? 20,3.25 RETURN
 20 hours at $ 3.25 per hour = $ 65
?
```

If you enter 0 as the number of hours, the program will stop and control will return to the BASIC interpreter.

```
?0,0 RETURN
>
```

The program that just executed is displayed at the top of the screen. The program dialog is on the next few lines and the BASIC prompt ("`>`") is in the last line.

If your program has run successfully:

Disc

1. Place an unprotected disc in the drive.

2. Enter SAVE "PAY" `RETURN`. This will record a copy of your program.

3. Enter EXIT `RETURN`. This will end the BASIC interpreter and return the terminal to normal operation.

Tape

1. Place a scratch or blank tape in the right tape drive. Make sure that the record tab on the tape is set to the record position.

2. Enter SAVE "R" `RETURN`. This will record a copy of your program.

3. Enter `COMMAND`, `f5`, `f6`, `RETURN`. This writes a file mark and rewinds the tape.

4. Enter `COMMAND` to close the Command channel and then Enter EXIT `RETURN`. This will end the BASIC interpreter and return the terminal to normal operation.

The program is now stored and may be loaded again with the GET command. If you do not want to rerun this program, you can clear the program copy in terminal memory by entering the command SCRATCH while you are in the BASIC interpreter. If you do not clear the program, it will still be in the workspace the next time you enter BASIC. This may interfere with any new program you might want to run. On the 2647F, either the GET or RUN commands will delete the current program in the workspace prior to loading the program requested.

Exiting BASIC

To return to normal terminal operation, type " EXIT **RETURN** ". Pressing **RESET TERMINAL** twice (a full reset) will also return the terminal to normal operation.

Pressing **RESET TERMINAL** (soft reset) causes you to enter the command mode of BASIC. If you have entered a BASIC program, it will remain in the BASIC workspace but all program variables will be set to zero.

If you enter a program and then leave BASIC without scratching or deleting the program, it will still be there if you re-enter BASIC at a later time. If you turn the terminal off, your program will be lost along with the BASIC interpreter.

Direct Computation

BASIC/47 can be used for direct computation without the need for a program. Numeric operations can be performed by entering a "?" followed by a numeric expression. Entering most BASIC expressions or statements without a statement number will cause the operation to be executed as soon as the **RETURN** key is pressed.

Example: Calculate $(2*(4+2)^3)$.

```
?2*(4+2)^3 RETURN
432
```

Example: Set X=3, Y=-8*X, and print the absolute value of X*Y.

```
LET X=3 RETURN
LET Y=-8*X RETURN
PRINT ABS(X*Y) RETURN
72
```

The following statements cannot be used in direct computation:

CALL	IMAGE	RESUME
DATA	ON	RETURN
FOR	NEXT	SUB
GOSUB	READ	SUBEND
GOTO	RESTORE	WAKEUP

Note that when variables are assigned values using the LET statement, the values are not displayed. Only a PRINT statement or expression evaluation causes the output to be displayed. Statements or expressions entered without line numbers are not stored. Values assigned to variables are lost if a program is subsequently run or if BASIC is exited.

You can pass direct computation values to your program by using the GO command to resume execution of a program that has been interrupted by the break character or a STOP statement.

This allows you to run a program, interrupt the program, print out program variable values, change the values of the variables, and then resume program execution with the new variable values.

You cannot assign a value to a variable using direct computation and pass it to a program run afterward (using the RUN command).

List Of BASIC Statements

Appendix D contains a list of the BASIC commands, statements, and functions.

Is That All There Is?

This section has discussed only a few of the features available in BASIC/47. The remainder of this manual explains the many other capabilities available.

- You can for instance, reconfigure the keyboard so that one or all of the keys will generate a different character. As an example, you can change the graphics control keys (ZOOM, CURSOR FAST, etc.) to a numeric keypad (0-9,.) with the following statements:

```
10 FOR I=1 TO 11
20 READ Kee, Code
30 KEYCDE(2,Kee,Code)
40 NEXT I
50 DATA 86,48,78,46,6,49,14,50,22,51,7,52
60 DATA 15,53,23,54,95,55,87,56,79,57
```

When this routine is executed, the graphics keys will generate the same characters as a numeric keypad whenever the right shift key is pressed at the same time as a graphic control key.
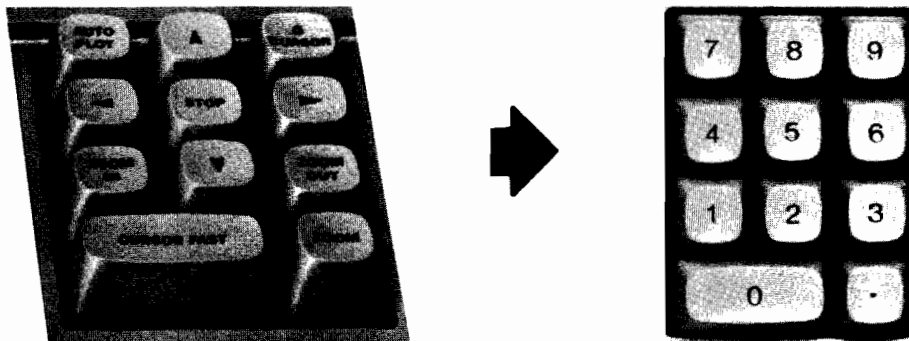
Introduction

- You can control all of the graphics capabilities of the terminal with a high level graphics language. For example, the single statement LGRID (5,5,0,0,2,2) can be used to generate a labeled grid.

LGRID (5,5,0,0,2,2)
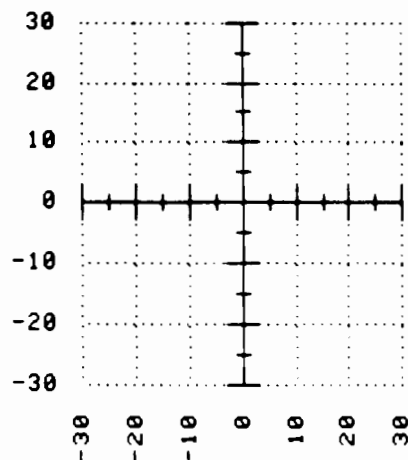
- You can directly access peripheral equipment such as a graphics digitizer to allow you to input coordinate data or a graphic plotter to output data. The following program generates a bar code pattern on a graphics printer at HP-IB address 4.

```
10 ASSIGN "HP-IB#4" TO #1
20 FOR I=1 TO 50
30 PRINT #1;"Ec*b10W";RPT$(CHR$(90),10):
40 NEXT I
```
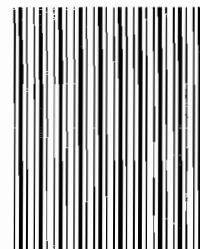
- You can put your program to "sleep" and resume normal terminal operation until a special key is struck at which time your program will "wakeup", perform some function, and then go back to "sleep".

Data_____2

The statements in BASIC/47 use data in the form of decimal numbers, character strings, and logical values. Any of these types of data can be combined into expressions. The data can be a specific value or a constant. It may also be referred to by name as a variable. A constant has only one value while a variable can be assigned different values at different times.

Constants

BASIC/47 programs can be used in a variety of applications such as data entry and retrieval, scientific and business calculations, and data display. Each application has its own requirements for the data it uses. Some programs work only with numeric data while others use only alphabetic data. Some programs perform calculations on very large numbers, while others use small numbers with many digits of precision. In the BASIC/47 language, you can specify the types of data you are going to use in a program or you can simply let the program use the standard (default) types. The purpose of this section is to describe the types of data that can be processed by BASIC and to introduce files and arrays, two features of BASIC designed to help organize and manipulate large sets of data.

BASIC uses two major classes of data:

- numeric data
- string data

A numeric data item can be any number between 10^{-38} and 10^{+38}, such as: 34, -22.999, 897564, and .0000001. Numbers that exceed 10^{+38} are not accepted. Numbers that are smaller than 10^{-38} are converted to 0. The precision of numbers is 6 digits for numbers of type short and 16 digits for numbers of type long. Numbers that exceed the available precision are rounded. A description of number types is given in the following paragraphs.

A string data item is a sequence of characters, such as:

```
THIS IS A STRING
246 Washington Blvd.
#X!*   JULY 4, 2076
```

The digits 0, 1, ... 9 are included in the list. The difference between the numeric digits 0, 1, ... 9 and the corresponding ASCII characters is that arithmetic calculations can be performed only on the numeric digits. Arithmetic calculations cannot be performed on digits that are a part of strings. Typical data items that contain digits that are not used in arithemtic calculations are addresses, zip codes, part numbers, and dates. Note that it is easy to convert strings of numeric characters to numeric data using the VAL function (see Section 12).

The characters used by the terminal consist of the 128 ASCII (American Standard Code for Information Interchange) characters. The ASCII character set is made up of both printing and control characters. Table 2-1 contains a list of the ASCII characters. Note that a space " " is a valid character.

Table 2-1. List of ASCII Characters

⁰		●	`		⁰	0	P	p
	!	A	a			1	Q	q
	"	B	b			2	R	r
	#	C	c			3	S	s
	$	D	d			4	T	t
	/	E	e			5	U	u
	&	F	f			6	V	v
	'	G	g			7	W	w
	(	H	h			8	X	x
	)	I	i			9	Y	y
	*	J	j			:	Z	z
	+	K	k			;	[	{
	,	L	l			<	\		
	-	M	m			=	]	}
	.	N	n			>	^	~
	/	O	o			?	_	■

Numeric Data

BASIC uses three types of numeric values: INTEGER, SHORT, and LONG. The types used in a program affect the speed of execution, the precision of the results, and the amount of space needed to store the values. If you do not specify the type(s), then SHORT values are assumed. Arithmetic with LONG values provides the maximum possible range of values and allows up to 16 decimal places, but execution speed is slower than for SHORT values. Also, LONG values occupy more storage than INTEGER or SHORT values.

The following paragraphs describe characteristics of the various numeric data types to help you determine the types most appropriate for your application. Once you have decided, you can use type declaration statements to specify the data types in your program.

CAUTION

When operators or numeric functions other than $+$, $-$, $/$, or $*$ are used, the operands are converted to type Short before the calculation is made. This may cause the result to vary from Long precision results.

Integers

Range: -32768 through 32767 (including 0)
Precision: not applicable
Size: 2 bytes

INTEGERS are positive or negative whole numbers and can range from -32768 to 32767 including 0. They do not allow the use of a decimal point or a comma. If you enter a decimal number (i.e. 10.3) in response to a program request for an INTEGER, the input will be rounded to an INTEGER. When INTEGER values are required in a program calculation, BASIC will automatically round the data to the nearest integer value before performing the calculation.

The following are INTEGER values:

        1   5600   -78 0 32767

The following are not INTEGER values because they include decimal positions:

        5.25   -7.999999   1.0 0.00001 -0.175

The following are not INTEGER values because they are outside the range of INTEGER values:

        -32769   32768   100000   -40000

INTEGER values are generally used in programs that involve counting such as keeping track of the number of items in inventory or assigning employee numbers. The execution speed of arithmetic operations is faster with INTEGER values than with any other numeric data type.

Note that when values exceed 1000, commas CANNOT be used to separate groups of three digits. For instance, the value ten thousand must be entered as 10000 rather than 10,000. Commas are used to separate one data item from another, so 10,000 would be read by BASIC as two values: 10 and 000.

Short

Range: 10^{-38} to 10^{38} (including 0)
Precision: 6 Significant Digits
Size: 4 bytes

Examples

The following are SHORT values:

        0.1   -103.75 9.12345   -0.222333   32123.9   1.000000   3000000000

The following are not SHORT values because they require more than six significant digits:

        9.1234567   -0.2223334   2.0000001

The following are not SHORT values because they are outside the range of SHORT values:

        1.6E92 2.13E-41

SHORT values provide a much greater range than INTEGER values and also allow for six significant digits of accuracy in calculations. All values included in the ranges of INTEGERs are also included in the range of SHORT values. Arithmetic with SHORT values is somewhat slower than with INTEGERs. SHORT values are useful for any arithmetic calculations that require just a few decimal places of accuracy.

Long

Range: 10^{-38} to 10^{38}
Precision: 16 Significant Digits
Size: 8 bytes

LONG values offer much greater precision than SHORT values. They also occupy twice as much storage space, and take longer to execute. Note that SHORT values are included in the range of LONGs.

The following can be LONG values:

        0.1 -103.75 9.12345 9.1234567 -0.2223334 3000000000 3.14159

The following are not LONG values because they use more decimal places than the LONG values allow (extra digit positions will be rounded off):

        1. 12345678901234567 -255.000000000000000001

The following are not LONG values because they are outside the range available for LONG values:

        1.6E92 2.13E-41

NOTE: Only $+$, $-$, $/$, or $*$ operations retain the precision of LONG values.

E-Notation

BASIC represents very large and very small numbers in E-notation (similar to scientific notation). Numbers are printed using E-notation when the number of digits needed exceeds 16. Numeric values that exceed 16 digits of precision or values of 10E38 or greater are represented as 9.999...E38. Values smaller than 1E-38 are represented as 0.

Rather than print 17,000,000,000,000,000

You should use: 17E15 (17×10^{15})

Scientific	Long Hand Notation	E-Notation
400,000,000,000,000,000,000,000	4×10^{23}	4E23
0.0000000000000000235	2.35×10^{-17}	$2.35E^{-17}$

An implied 1 is not allowed, since E9 is a valid variable name. Instead use 1E9 to represent 1000000000.

Octal And Hexadecimal Data

The terminal will accept octal or hexadecimal data as input. The data must be in the form ***Q or ***H. The input will be automatically converted to a decimal equivalent before being passed to your program.

Example: 377Q = 0FFH = 255

This means that if your input data contains a "Q" or "H" following the numeric value, you will not receive an error message, and the data will not be interpreted as a decimal number.

Example:

```
10 FOR I=1 TO 3 STEP 1
20 INPUT N
30 PRINT N
40 NEXT I
>RUN
?10 RETURN
10
?10Q RETURN
8
?10H RETURN
16
```

Note that hexadecimal numbers must begin with a numeric digit (e.g. OFFH). If you attempt to enter a hexadecimal number that begins with a letter, it will be seen as a variable name (ABH = Abh).

String Data

A string can be from 0 to 255 ASCII characters long. When you use a string in a BASIC program, you must always enclose it in quotation marks. The quotation marks around the string are not considered part of the string. Any ASCII character except for the double quotation mark and the carriage return can be enclosed in quotation marks. To include quotation marks or carriage returns in a string you must use the CHR$ function (refer to Section 3). A detailed description of strings is given in Section 7.

Examples	Notes
"THIS IS A STRING"	This string is 16 characters long.
"#X-!"	This string is 4 characters long.
" "	This is a string of 4 blanks or space characters.
""	This is the null string. The null string has no characters. Its length is 0.
THIS IS A STRING	Invalid. Quotes are missing.
"#X-!	Invalid. End quotes are missing.
#X-!"	Invalid. Beginning quotes are missing.
"Quote is ""	Invalid. String cannot use a quote character.
"12.95"	This string contains digits. It can be converted to the equivalent numeric data 12.95 using the VAL function. As shown it is a string of five characters.

BASIC requires that all quote marks must occur in pairs, even if they are in a REM statement or comment. If individual quotes are required, the CHR$ function can be used to generate a quote character.

Variables

Variable names are used to represent numeric or string values in a program. For instance, in the statement

        10 LET Limit=999

"Limit" is the name of a variable and 999 is the value to be assigned to Limit. Numeric values are represented by numeric variables; strings are represented by string variables. Unlike constants, the value associated with a variable can change during program execution. When referencing a variable the program will always use the current value of the variable.

Numeric Variables

A numeric variable name is composed of an upper case letter, A through Z, which may be followed by any combination of up to 14 digits, lower case letters, and the symbol "_" (underscore). You may enter characters in upper or lower case. BASIC will automatically convert them to the variable name format (the first character uppercase and the remainder lowercase). The variable name cannot start with a string of characters used by BASIC to identify commands, statements, or functions (i.e. RUN, END, COS, etc.). A list of these "reserved words" is given in Appendix C.

Examples

X	Valid.
Y2	Valid.
A12345678901234	Valid.
Program_cntr	Valid
9X	Invalid. Digit must follow letter.

If you use a numeric variable name that is longer than 15 characters, the extra characters are ignored. For instance, the variables

        Program_number_1 and Program_number_2

are equivalent since only the first 15 characters of each are retained.

Variable names are terminated by a blank or any disallowed character (#, %, etc.).

> **NOTE:** Array names and subscripted variables are special kinds of variables. They are discussed in this section under Arrays.

Numeric variables are set to 0 each time the RUN command is executed.

Type Declaration Statements

A numeric variable is assumed to refer to a value in the SHORT range unless it appears in a type declaration statement. There are three forms of the type declaration statement corresponding to the three types of numeric variables. These statements specify the range that a variable may represent.

Example	Notes
10 INTEGER N,C	Declares that the variables N and C will have values in the INTEGER range.
30 SHORT S,Title	The variables S and Title are type SHORT.
40 LONG X,Y,Z	X, Y, and Z are type LONG.

Note that statement 30 is not necessary since any numeric variables that do not appear in a type declaration statement are assumed to be type SHORT. However, for documentation purposes, it is recommended that all numeric variables that are used in a program appear in type declaration statements.

Type declaration statements are executable statements. If a variable in a type declaration statement has been listed in a previously executed type declaration statement or was used in any previously executed statement, a REDECLARED VARIABLE error will occur. This error will also occur if the same type declaration statement is re-executed.

It may be helpful to use the program listed below to become familiar with the different types of numeric data. Enter the program by following the same steps used to enter the sample program in Section 1. This program calculates the sum (Sum) and the mean (Mean) of five numbers (A,B,C,D, and E). When you enter RUN ⬛, the program will execute.

A "?" character will be displayed and the program will wait for you to enter the five numbers. If the values you provide are not in the SHORT range, they are automatically converted to SHORT values before any calculations are performed, and the results are SHORT values.

Sample Program

```
10 REM THIS PROGRAM CALCULATES THE SUM AND
20 REM THE MEAN OF FIVE NUMBERS (A,B,C,D,E)
30 INPUT A,B,C,D,E
40 LET Sum=A+B+C+D+E
50 LET Mean = Sum/5
60 PRINT A,B,C,D,E
70 PRINT "SUM =";Sum
80 PRINT "MEAN =";Mean
```

RUN `RETURN`

```
? 98,94,82,86,78
98              94              82              86              78

SUM = 438
MEAN = 87.6
```

RUN `RETURN`

```
?1.234567,2.345678,3.456789,4.567890,5.678901
1.23457     2.34568     3.45679     4.56789     5.6789

SUM = 17.2838
MEAN = 3.45677
```

Now change the program by inserting a type declaration statement as statement 25. For instance, what is the result if you add this statement?

```
25 INTEGER A,B,C,D,E,Sum,Mean
```

Does the result change if statement 25 is

```
25 INTEGER Sum, Mean ?
```

String Variables

A string variable name is formed by including a dollar sign ("$") as the last character of a valid numeric variable name. String variable names are limited to 14 characters followed by a "$". If more than 14 characters are used, BASIC will use the first 14 characters and the "$". This means that "String-number-11$" and "String-number-12$" are both equivalent to "String-number$".

	Examples	Notes
	X$	Valid.
	Y8$	Valid.
	String-1$	Valid.
	Stringvariable$	Valid.
	$X	Invalid. $ must come last.
	String-variable$	Valid, but exceeds 15 character limit. Extra characters ignored, therefore this is the same as String-variabl$.

Although string variable names are very similar to numeric variable names, these two variable types are completely independent of each other. For instance, the variable names X8 and X8$ would both be allowed in the same program. Section VII contains a detailed description of strings and string functions.

DIM Statement

For string variables which will contain strings longer than 18 characters, you must specify the string variable name and the maximum length of the string, in brackets, in a DIM statement.

Example

string
names

10 DIM S$[25],C$[35]   This statement reserves data space for a 25 character string S$ and a 35 character string C$.

maximum
length

A string variable defined by a DIM statement can contain a string shorter than the length specified. However, if a string is longer than the specified length, all extra characters to the right are ignored (truncated).

Sample Program

```
10 DIM S$[25]
20 S$ = "THIS STRING IS LONGER THAN 25 CHARACTERS"
30 PRINT S$
40 END

RUN RETURN

THIS STRING IS LONGER THA
```

A string variable that does not appear in a DIM statement defaults to a maximum length of 18 characters. Thus, string variables that will always contain strings shorter than 19 characters do not have to be dimensioned. For documentation purposes, and more efficient memory allocation, however, it is recommended that you dimension all string variables.

Sample Program	Notes

```
10 DIM A$[12],B$[5]
20 LET A$="STRING"
30 LET B$="STRING"
40 LET C$="STRING"
50 PRINT A$
60 PRINT B$
70 PRINT C$
80 END
```

Statement 10 reserves space for a 12 character string A$ and a 5 character string B$. The string C$, by default, has a maximum length of 18 characters.

RUN `RETURN`

```
STRING
STRIN
STRING
```

Notice that because the string "STRING" is longer than the 5 character maximum specified for B$, only the first five characters of "STRING" are printed.

DIM statements are executable statements. If a variable in a DIM statement was used in a previously executed statement, a REDECLARED VARIABLE error will occur when the DIM statement is executed. This error will also occur if the DIM statement is re-executed.

Arrays and Subscripted Variables

Variable names can also refer to arrays of data. An array is generally used for processing an entire set of data rather than a single data item, but you can access an individual data item within an array. The individual data items in arrays are called array elements. Subscripted variables are used to refer to individual array elements.

An array may have from 1 to 32 dimensions. Examples of 1-, 2-, and 3-dimensional arrays follow. Arrays of four or more dimensions are not so conveniently represented on paper but they can be set up and manipulated easily in a BASIC program. The complete description of how to use arrays is in Section 6.

Examples

One-dimensional array:

```
1.5 2.3 3.4 4.7 10.7 .8 3.5 4.6 2.0 1.1 2.3
```

Two-dimensional array:

```
12.95 12.95 11.50 11.50 11.50 11.50
 3.95  3.50  3.50  3.50  3.50  3.00
  .80   .80   .80   .80   .80   .80
```

Two-dimensional array:

```
McConnoll    Pat       123-4577   Palo Alto
Stearne      Bill      890-2345   Santa Clara
Allan        Walter    678-9023   Saratoga
Spencer      Dave      456-7890   Cupertino
Wilson       Ed        134-5788   San Jose
Greene       Dave      901-2456   Mountain View
Underwood    Doug      789-0123   Los Altos
Olsen        Mary      456-7890   Menlo Park
```

Three-dimensional array A(3,3,3):

```
ROW          ROW          ROW
 COL          COL          COL
  DPTH          DPTH          DPTH
─────        ─────        ─────
123          012          987
456          345          654
789          678          321
```

Subscripted variables are used to refer to individual elements in an array. Elements in 2-dimensional and larger arrays are identified by 2 or more subscripts, separated by commas. For instance, A(2,3) is a subscripted variable that refers to the element in row 2, column 3 of array A. The subscripts can be numeric expressions, numeric variables, or constants. The subscript is rounded to an integer if necessary.

String variables can also be subscripted. In this case the subscripts select character positions from within the string. For example, if A$="MORNING", then A$[5;1]="I". Additional information on strings and subscripted strings is given in Section 7.

Files

You will want to store large sets of data in files. Files allow you to organize and store data in a manner that provides access to individual items. The data in a file can be of a single type or the file can contain many data types. The amount of data that can be stored in a file is not limited by BASIC. It is limited only by the amount of storage space available on the system. A data file may be stored on a flexible disc, cartridge tape, or on a remote computer. Section 9 describes the way files are created and accessed in BASIC.

Logical Values

When a variable or numeric expression is used as a logical value, the variable or expression is evaluated as true if it is non-zero and false if it is 0. This means that $-.1$ for example would evaluate to true.

Example:

```
10 FOR I = -3 TO 2
20 IF I THEN 50
30 PRINT "I=";I;" IS FALSE"
40 GOTO 60
50 PRINT "I=";I;" IS TRUE"
60 NEXT  I
RUN RETURN
I=-3  IS TRUE
I=-2  IS TRUE
I=-1  IS TRUE
I= 0  IS FALSE
I= 1  IS TRUE
I= 2  IS TRUE
```

Note that if a noninteger data type is used, round off errors may prevent you from obtaining exact values for variables. In the above example, if a step size of .2 is used, round off errors will produce a value for I of 3.8743E-07 instead of zero. To avoid this type of error you should check for a range of values that can serve as your 0". For example:

```
20 IF ABS(I)>.001 THEN 50
```

Operators, Functions and Expressions _____ 3

This section describes the various operations and functions that can be performed on BASIC numeric and string data. It includes four types of operators and five types of functions. These operators and functions can be used to create expressions. Expressions are evaluated to obtain numbers, strings, or logical values. Most applications will use only a small subset of the available operators and functions.

Operators	Functions
• Arithmetic	• Numeric
• Relational	• String
• Logical	• Print
• String	• Input/Output
	• Other

Operators

An operator indicates a mathematical or logical operation to be performed on one or two values (operands) resulting in a single value. The combination of one or two operands with an operator is called an expression.

expression

```
      10 + 5
     /   |   \
operand  |  operand
         |
      operator
```
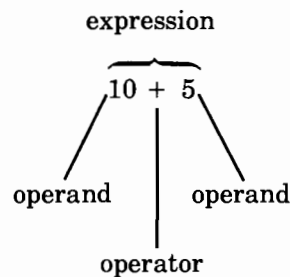
Expressions in BASIC are similar in form to normal algebraic expressions. This allows most arithmetic operations to be programmed directly without changing their form.

The term operand can refer to a number, a string, or a variable. Generally, an operator is between two operands but an operator can also precede a single operand. For instance, the minus sign is an operator which indicates subtraction when it appears between two operands (e.g., 512-88) and a negative number when it appears before a single operand (e.g., -1).

Operators may be divided into four classes depending on the kind of operation performed: arithmetic, string, relational, and logical (Boolean).

CAUTION

When operators or numeric functions other than $+$, $-$, $/$, or $*$ are used, the operands are converted to type Short before the calculation is made. This may cause the result to vary from Long precision results.

Expressions	Notes
A - B	
X + 1	
-1	
+2.14	
2.14	
N$	Strings or string variables can also be considered expressions.
''STRING''	
"AB" & "CD"	The string operator "&" is used to combine two strings into one.
(P+5)/27	P is a variable that must have been previously assigned a value. 5 and 27 are constants. The slash is the divide operator. Parentheses group the portions of the expression to be evaluated first. Suppose that the current value of P is 49. The expression is then (49+5)/27 and evaluates to 2.
(N-(R+5))-T	N, R, and T are all numeric variables that must have been previously assigned values. The innermost parentheses are evaluated first. Suppose that N is 20, R is 10, and T is 5. The expression is (20-(10+5))-5 and evaluates to 0.

Arithmetic Operators

The arithmetic operators are:

Operators	Operations	Examples
+	add	$10 + 5 = 15$
-	subtract,	$10 - 5 = 5$
	negative number	-2
*	multiply	$10 * 5 = 50$
/	floating point divide	$15/10 = 1.5$
^	exponentiate	$8^3 = 512$
DIV	integer divide	$15 \text{ DIV } 10 = 1$
		$-15 \text{ DIV } 10 = -1$
MOD	modulo;	$38 \text{ MOD } 8 = 6$
	A MOD B = A -(B *	$-13 \text{ MOD } 2 = -1$
	INT(A/B))	$-13 \text{ MOD } -2 = -1$

Note that, unlike algebraic notation, implied multiplication does not exist in BASIC. Thus, A × B must be written as A * B rather than just AB. The operation of raising a number to a power also requires an explicit operator. Thus A^B is written as A^B.

There are two division operators: "/" and DIV. Division with the "/" operator (called floating point division) results in a value in the LONG range if either of the operands is type LONG. Otherwise, the result will be a SHORT value. When the DIV operator is used (integer division), the result is an INTEGER value.

The operands for MOD and DIV are rounded to integers before the operations are performed. This means that MOD and DIV can only be used on numbers in the INTEGER range of values.

Examples

$3/2 = 1.5$
$3 \text{ DIV } 2 = 1$

$-10/5 = -2.0$
$-10 \text{ DIV } 5 = -2$

$9.999999999/1 = 9.999999999$
$9.999999999 \text{ DIV } 1 = 10$

The function INT(X) which is used to calculate A MOD B returns the greatest integer less than or equal to X. So, using the formula A MOD B = A − (B*INT(A/B)), we have:

```
38 MOD  6 = 38 -(6 * INT(38/6))
              = 38 -(6 * 6)
              = 38 -36
              = 2

-13 MOD  2 = -13 - (2 * INT(-13/2))
              = -13 - (2 * -6)
              = -13 - (-12)
              = -1

-13 MOD -2 = -13 - ((1-2) * INT(-13/-2))
              = -13 - ((1-2) * 6)
              = -13 - (1-12)
              = -1
```

Expressions with more than two values are evaluated according to the following hierarchy of operators:

^ (highest)
Unary +, −
*, /
DIV
MOD
+, − (lowest)

Examples

$5 + 6 * 7 = 5 + 42 = 47$
$5 * 6 + 7 = 30 + 7 = 37$

If operators are at the same level, the order is from left to right in the expression.

Examples

$30 - 40 + 100 = -10 + 100 = 90$
$2 + 3^2 - 1 = 2 + 9 - 1 = 11 - 1 = 10$

Parentheses can be used to override this order.

Examples

$30 - (40 + 100) = 30 - 140 = - 110$
$2 + 3^{\wedge}(2 - 1) = 2 + 3^{\wedge}(1) = 2 + 3 = 5$
$5 + 6 * 7 = 5 + 42 = 47$
$(5 + 6) * 7 = 11 * 7 = 77$
$14/7 * 6/4 = 2 * 6/4 = 12/4 = 3$
$14/(7 * 6)/4 = 14/42/4 = .333\ldots/4 = .083$

When parentheses are nested, operations within the innermost pair are performed first.

Examples

$(100/(4 + 6) * 2) = 100/(10 * 2) = 100/20 = 5$
$2 * ((3 + 4) - 5)/6 = 2 * (7 - 5)/6 = 2 * 2/6 = 4/6 = .6666\ldots$

String Operators

The string operator "&" is used to combine two strings into one. This is called string concatenation. For example, if A$ = "ABC" and B$ = "DEF", then A$ & B$ = "ABCDEF". The characters in B$ immediately follow the characters in A$.

Example:

```
5 DIM C$[23]
10 A$="TEST OF TERMINAL"
20 B$="COMPUTER"
30 C$=A$[1,8]&B$&" SYSTEM"
40 PRINT C$
RUN
TEST OF COMPUTER SYSTEM
```

Relational Operators

The relational operators are:

Operator	Operations	Example
<	less than	A < B
>	greater than	A > B
<=	less than or equal to	A< = B
>=	greater than or equal to	A> = B
=	equals	A = B
<>	not equal to	A <> B

When relational operators are used in a numeric expression, the value 1 is returned if the relation is found to be true; the value 0 is returned if the relation is false. For instance, A = B is evaluated as 1 if A and B are equal in value, or as 0 if they are not equal. If A = 1, B = 2, and C = 3, then $(A * B) < (A - C/3)$ is evaluated as 0 (false) because $A * B = 2$ which is not less than $(A - C/3) = 0$.

If the two operands are of different numeric types, the operand with the lower type is converted to the higher type, the operation is performed, and the result is in the range of the higher type. The hierarchy of numeric types is:

LONG   (highest)
SHORT
INTEGER (lowest)

Sample Program

```
10 INPUT A,B,C
20 Result = (A*B)<C
30 If Result THEN 60
40 PRINT "(A*B)<C is false -result";Result
50 GOTO 70
60 PRINT "(A*B)<C is true -result";Result
70 END

RUN [RETURN]

?3,4,5
(A*B)<C is false - result 0
```

Relational operators are also used to compare strings. Strings are compared according to their associated numeric values in the ASCII code (see Appendix A). The strings are compared character by character until a difference is found, or until the end of a string is reached. If the ends of both strings are found at the same time, the strings are equal. If the end of one string is reached, then that string is an initial substring of the other, and is considered to be less than the other.

Examples	Notes
"ABC" < "ABC "	"ABC" is an initial substring of "ABC ".
"AB " = "AB "	
"B" > "ABC"	"B" has a higher numeric equivalent than "A" in the ASCII code.
"AB$" < "AB*"	"$" has a lower numeric equivalent than "*" in the ASCII code.

The null string ("") is always less than every other string and equal only to another null string.

More information on substrings is presented at the end of this section.

Logical Operators

The logical operators (sometimes called Boolean operators) are:

- AND
- OR
- NOT
- XOR
- CMP

These operators are most frequently used as part of an IF . . . THEN statement. The operands used with logical operators are converted to type INTEGER before the logical operators are used. If the operand is outside of the range of INTEGERS (– 32,768 to 32,767) an error will result.

The expressions that the logical operators compare can be either relational or non-relational. If the expression is relational (like A<B), whether it is true or false is determined by the relation of the operands. If the expression is non-relational (like A), it is true if its arithmetic value is not zero and false if its arithmetic value is zero.

The AND, OR, XOR, and CMP operators act on the operands bit by bit. The operands are made up of a sign bit and 15 binary bits.

AND

AND compares two operands. The operands can be either INTEGER expressions or relational expressions. The result has both a numeric and a logical value. The numeric value of the comparison is obtained by comparing the values of the two operands, bit by bit. If both bits are 1 the resulting bit is 1. If either bit is 0 then the resulting bit is 0. For example, if operand A = 5 and operand B = 3, then A AND B = 1.

```
A = 5      = 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1
B = 3      = 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1
             - - - - - - - - - - - - - - - - - - - - - - -
A  AND  B  = 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 = 1
```

The logical value of the comparison is the same as all other logical values, false (0) if the result is zero, and true (1) if the result is non-zero.

OR

OR compares two operands. The operands can be either INTEGER expressions or relational expressions. The result has both a numeric and a logical value. The numeric value is obtained by comparing the operands, bit by bit. If either bit is 1 then the resulting bit is 1. If neither bit is 1 then the resulting bit is 0. For example, if A = 3 and B = 4, then A OR B = 7.

```
A = 3      = 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1
B = 4      = 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0
             - - - - - - - - - - - - - - - - - - - - - - -
A  OR  B   = 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 = 7
```

The logical value of the comparison is the same as all other logical values, false (0) if the result is zero, and true (1) if the result is non-zero.

NOT

NOT changes only the logical value of an expression. If the expression is true, NOT changes its logical value to false (0). If the expression is false, NOT changes its logical value to true (1). The NOT operator does not have a numeric value as a result.

XOR

The XOR operator performs an exclusive OR of the operands. The operation is applied bit by bit. If both bits are 1, or if both bits are 0, the result is 0. If the bits are not equal, the result is 1. For example, if A = 7 and B = 2, then A XOR B = 5.

```
A = 7     = 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1
B = 2     = 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0
            - - - - - - - - - - - - - - - - - - -
A XOR B = 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 = 5
```

The logical value of the comparison is the same as all other logical values, false (0) if the result is zero, and true (1) if the result is non-zero.

CMP

The CMP operator complements the operand. The operation is applied bit by bit. If a bit is 1, it results in 0. If a bit is 0, it results in 1. For example, if A = 5, then CMP 5 = -6.

```
A = 5 = 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1
        - - - - - - - - - - - - - - - - - - -
CMP A = 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 0 = -6
```

Examples:

Assume A = 0, B = 2, C = 4, and D = 4.

A < B AND C = D   True because both relational expressions A < B and C = D are true.

A AND C = D   False because the arithmetic value of A equals zero (false).

B AND C   False because B and C do not have any common bits.

A OR B   True because the arithmetic value of B is not zero (so B is true).

NOT A   Since A is zero (false), NOT A is true.

NOT B OR NOT C   False. NOT B is false and NOT C is false.

When arithmetic, relational, and logical operators appear in a single numeric equation, the operations are performed according to the following hierarchy:

```
^              (highest)
unary +, −
*, /
DIV
MOD
arithmetic +, − and & (string concatenation)
Relational (=, <, >, <=, >=, <>)
NOT
CMP
AND
OR
XOR            (lowest)
```

Functions

A function is a routine that manipulates numeric or string data and produces a numeric or string value as a result. Some of the commonly used functions, such as the one to compute the square root of a number are supplied as a part of BASIC. A function is identified by a three or four character name followed by optional operands in parentheses. Since a function results in a single value, it can be used in an expression wherever a constant or variable would be used. Some common functions are:

SQR(X) Where X is a numeric expression that results in a value or 0. When called the function returns the positive square root of X. For example, if N=2, SQR(N+2)=2.

ABS(X) Where X is a numeric expression. When called, the function returns the absolute value of X. For example, ABS(−33)=33.

TRIM$(T$) Where T$ is an ASCII text string. When called, the function returns the characters in the string with all leading and trailing blanks removed. For example, if T$=" Texas ", TRIM$(T$)= Texas.

Numeric Functions

Numeric functions return a numeric value as a result. BASIC provides the following numeric functions:

ABS(X)	Same type as X.	Absolute value of X.
ATN(X)	SHORT	Arctangent of X; result expressed in radians.
COS(X)	SHORT	Cosine of X; X expressed in radians.
EXP(X)	SHORT	E raised to the power of X.
INT(X)	INTEGER	Largest integer $<=$ X.
LOG(X)	Short	Natural logarithm; X $>$ 0.
LONG(X)	Long	X converted to LONG representation.
SGN(X)	Integer	The sign of X; -1 if X$<$0, 0 if X$=$0, and $+1$ if X $>$ 0.
SHORT (X)	Short	X converted to SHORT representation.
SIN(X)	Short	Sine of X; expressed in radians.
SQR(X)	Short	The positive square root of X.
TAN(X)	Short	Tangent of X; X expressed in radians.

Each numeric function consists of a function name followed by one parameter. The parameter may be a number (as in statement 10 below), a numeric variable (statement 20), or a numeric expression (statement 30). Since the result of a numeric function is always a single value, a numeric function can be used as an operand in an expression (statement 40), or as a parameter of a numeric function (statement 50).

Examples:

10 LET A = COS(0)	A equals the cosine of 0 = 1.
20 LET B = ABS(A)	B equals the absolute value of 1 = 1.
30 LET C = SQR(A + B)	C equals the square root of (1 + 1) = 1.414214.
40 LET D = COS(A)^2+SIN(A)^2	D equals the sum of COS(A) squared plus SIN(A) squared.
50 LET E = ABS(SIN(0))	SIN(0) = 0; absolute value of 0 = 0; therefore E = 0.

Other functions are used to manipulate strings or arrays and to control the format of program output. A complete list of the functions available in BASIC is given in Appendix D at the back of this manual.

Random Numbers

A pseudo random number generator, the RND function, is provided for programs that perform simulations. Each time the RND function is called, a random number between 0.0 and 1.0 is returned. The RND function is called a pseudo random number generator because the same sequence of random numbers is generated each time the BASIC interpreter is loaded.

Sample Program

```
5  DIM A(10)
10 FOR N = 1 TO 10
20 A(N) = RND
30 PRINT A(N)
40 NEXT N

RUN RETURN

.8970581
.779265
.35281
.757005
.8578032
.7452424
.7903184
.507
.11975
.859175
```

String Functions

CHR$(X)ST   Returns a string character with the value of the numeric expression X. The value of X must be between 0 and 255.

```
10 LET T$=CHR$(32)  ! T$ = " " (blank)
```

DATE$ Returns the current date as an ASCII text string from the terminal's internal calendar.

```
20 LET D$=DATE$[6,11]        !DATE$=TUE, JAN 12, 1982
30 PRINT D$                  !Print selected portion of date string
        JAN 12
```

LEN(S$)   Returns the number of ASCII characters in the string S$.

```
10 DIM A$[20]
20 LET A$="1234"
30 LET B = LEN(A$)           !B=4
```

NUM(S$)   Returns a numeric value between 0 and 255 corresponding to the first character of the string S$. S$ cannot be the null string "".

```
10 LET T$="Payroll"
20 LET B = NUM(T$[2;1])      !B=97
```

POS(S1$,S2$)   Searches the string S1$ for the first occurrence of the string S2$. Returns
                   the starting index if found, otherwise returns 0.

```
10 LET T$="Television"
20 LET V$="vision"
30 LET C=POS(T$,V$)              !C=5
```

PROC$(X)   Returns the string representation of the key with a code value of X. If the
                   key is a function key, the appropriate escape sequence is returned. PROC$
                   cannot process all of the terminal keycodes.

RPT$(S$,X)   Repeats the string in S$ X times.

```
10 LET T$"Ring!"
20 LET E$=RPT$(T$,3)
E$="Ring!Ring!Ring!"
```

TIME$ Returns the current time from the terminal's internal clock as an ASCII text
                   string.

```
10 LET K$="TIME SENT"
20 LET H$=TIME$
30 PRINT K$;H$
           TIME SENT 3:45:34 AM
```

TRIM$(S$)   Returns a string which is equal to S$ with all leading and trailing blanks
                   stripped off.

```
10 LET S$="  Pay roll   "
20 LET F$=TRIM$(S$)
F$="Pay roll"
```

UPC$(S$)   Returns a string made up of the uppercase equivalents of each of the
                   characters in S$.

VAL$(S$)   Returns the numeric equivalent of the string S$. *--The data type returned
                   may be integer, short, or long as required.

```
200 N=VAL(N$)+I
```

VAL$(X)   Returns the value of the numeric expression X expressed as a string of
                   ASCII digits.

```
10 LET T$=VAL$(128+34)           !T$="162"
```

VERS$ Returns the BASIC version identification string printed in the BASIC
                   welcome message.

```
20 LET P$=VERS$
30 PRINT P$
           A-2142-42
```

Examples

RPT$, TRIM$, LEN, and POS are used to manipulate and create strings:

```
10 LET S$ = "REPEAT"                    Output is: REPEATREPEATREPEAT
20 PRINT RPT$(S$,3)
```

```
10 LET A$ = "  AB C  "                  Prints AB C (no leading or trailing blanks)
20 LET B$ = TRIM$(A$)
30 PRINT B$
```

```
10 DIM A$[20]                           Prints the number 4
20 LET A$ = "ABCD"
30 PRINT LEN(A$)
```

```
10 LET T$ = "Television"
20 LET V$ = "vision"
30 LET C = POS(T$,V$)                    C = 5
40 PRINT C
```

NUM and CHR$ are used to go back and forth between an ASCII character and its ASCII code value:

```
10 LET A$ = CHR$(13)                     PRINT statement will execute a carriage return.
20 PRINT A$
```

```
10 LET A$ = CHR$(69)                     Prints the letter E.
20 PRINT A$
```

```
10 PRINT "QUOTE";CHR$(34);"MARK"         Prints QUOTE " MARK
```

```
10 PRINT NUM("Egg")                      Prints the number 69 (ASCII numeric equivalent
                                         of capital E)
```

```
10 B$="Payroll"                          Prints the number 97.
20 PRINT NUM(B$[2])
```

VAL$ and VAL are used to go back and forth between strings of ASCII digits ("1234") and numeric values:

```
10 LET B$ = VAL$(COS(PI))                Prints 1
20 PRINT B$
```

```
5  LET Pay$ = "127.99"
10 LET D=VAL(Pay$)                       D = 127.99
20 PRINT D
```

```
10 LET T$ = VAL$(128)                    T$="128"
20 PRINT T$
```

Statements _____4

This section introduces program statements and describes some of the fundamental statement types. All programs are made up of numbered statements. These statements are executed by the BASIC interpreter in numeric sequence to perform the program task.

The statements presented in this section are:

REM	RESTORE	FOR and NEXT
LET	PRINT	GOSUB
INPUT	GOTO	RETURN
LINPUT	ON...GOTO	ON...GOSUB
READ	IF...THEN	STOP and END
DATA		

The REM statement and comments are useful for program documentation. All programs should be carefully documented to make them easier to debug, use, and maintain. The LET, INPUT, LINPUT, READ, DATA, and RESTORE statements supply data for a program. The PRINT statement prints program results. The rest of the statements described here are used to control the flow of a program by branching, looping, and calling subroutines. Statements that apply to more advanced programming techniques such as formatted output, string operations, file operations, and subprogramming are covered in separate sections.

The discussion of each statement begins with a description of what the statement is used for and a few examples to demonstrate how it can (or cannot) be used. In most cases, this will be sufficient explanation to begin using the statement in your programs. The remainder of each discussion describes the statement in more detail, often illustrating special uses with more examples and sample programs.

REM Statement

The REM statement allows you to insert helpful notes and messages in your program. These statements do not affect the execution of the program (but they do take up storage space). The name and purpose of the program, how to use it, how certain parts of the program work, and expected results are useful information to include in a program for documentation. If control is passed to the REM statement, execution continues with the statement following the REM statement.

Examples:

```
10 REM  YOU CAN SAY
20 REM --- ANYTHING YOU WANT
30 REM:    TNAW UOY YAW YNA
40 REM.... IN A $&#*(@ REM STATEMENT
```

The message itself can contain any printable characters.

The exclamation mark (!) can be used in place of the REM statement. The exclamation mark enables you to put comments on the same line as a program statement.

Examples:

```
10 LET A= B^2    ! SET A EQUAL TO B SQUARED
20 PRINT A       ! PRINT THE VALUE OF A
```

Any printable character may follow the letters REM in a REM statement or the exclamation mark in other statements. Note that you cannot execute statements or functions that follow an exclamation mark ("!") in a line.

If quotes are used in the REM statement, they must be used in pairs. You cannot use unmatched quotes in REM statements.

REM statements and comments following exclamation marks are part of a BASIC program and are output when the program is listed, displayed, saved, or recorded but they have no effect on program execution.

LET Statement

The LET statement assigns a value to one or more variables. The value can be an expression, constant, function, or another variable.

Examples:

```
           variable
              /  value
             /  /
10 LET A = 0             Sets A to 0

10 LET B = A             Sets B to the value of A.

10 LET M$ = T$           Sets M$ to the value of T$.

10 LET R = SIN(T)^2      Computes the value of the expression SIN(T)^2 and assigns that
                         value to R.

10 LET X = X$            INVALID. Cannot assign a string value to a numeric variable.

10 LET X$ = X            INVALID. Cannot assign a numeric value to a string variable.
```

Notice that the equals sign does not indicate equality, but is a signal that the value on the right is to be assigned to the variable on the left.

If the numeric type of the value on the right of the equal sign is different than the numeric type of the variable on the left, then when the value is assigned to the variable it is converted to the same type as the variable. SHORT and LONG values are rounded to the nearest whole number when being converted to INTEGER.

Example:

```
10 INTEGER I
20 LONG X
30 LET X = 1.998.4236   X has the LONG value 1.998.4236
```

```
40 LET I = X            The value of X is still 1.988.4236 but the value assigned to I is
                        the INTEGER 2, that is, 1.998.4236 converted to an INTEGER.
```

```
50 PRINT "I=";I
```

RUN [RETURN]

```
I=2
```

For convenience LET may be omitted from the statement. This is the only statement in the BASIC language in which the statement name is optional.

Examples:

```
10 A = 0        !Same as: 10 LET A = 0

10 B = A        !Same as: 10 LET B = A

10 M$ = T$      !Same as: 10 LET M$ = T$

10 R=SIN(T)^2   !Same as: 10 LET R=SIN(T)^2
```

Multiple Assignment

Multiple assignment allows you to assign a value to several variables in a single LET statement. For example, the statement

```
30 LET A,B,C = 12
```

assigns the value 12 to the variables A, B, and C. The following two sets of statements are equivalent:

```
 5 Pi=3.14          5 Pi=3.14
10 INTEGER I,J     10 INTEGER I,J
20 LET I = 0       20 LET I,J = 0
30 LET J = 0       30 X,J,Y = Pi
40 X = Pi
50 J = Pi
60 Y = Pi
```

Variables used in multiple assignment statements are assigned values from left to right. This is very important when using subscripted variables, particularly if the subscript contains a variable that is assigned a new value in the same statement.

Examples:

```
10 N=5
20 A(N),A(N+1)=7
```
Array element A(5) is assigned the value 7 and then element A(6) is assigned the value 7.

```
30 N=3
40 A(N)=6
50 A(A(N)),A(N)=7
```
Array element A(6) is set to 7 and then element A(3) is set to 7.

```
60 PRINT A(5),A(6),A(6),A(3)
70 END
```

>RUN **RETURN**

```
7       7       7       7
```

Example:

```
10 N=2
20 A(N),N,A(N)=3
30 PRINT A(2),N,A(3)
```

>RUN **RETURN**

```
3       3       3
```

Relational Tests For Equality

The statement LET A = B = C is a relational test for equality that sets the value of A to either 1 or 0 depending on whether B = C or B <> C. (See 'Relational Operators', Section 3.) A conflict between multiple assignment and a relational test for equality cannot occur since the multiple assignment statement uses commas to separate variables. The examples which follow illustrate multiple assignment statements and relational tests for equality.

Examples:

`10 A,B, = C`	Multiple assignment: B= C and A= C.

`10 A = B$ = C$` Relational test for equality:
Does B$ = C$?
　If YES then A=1;
　if No then A=0.

This is not multiple assignment because a string value cannot be assigned to a numeric variable.

`10 B$,C$ = D$` Multiple assignment: C$= D$ and B$= D$.

`10 A,B = C = D` Combination: Does C = D?
　If YES then B=1 and A=1;
　if No then B=0 and A=0.

`10 B$ = A = B` INVALID. This cannot be multiple assignment because the numeric value of B cannot be assigned to the string variable B$; nor can this be a relational test for equality:
Does A = B?
　If Yes then B$=1;
　　Invalid
　if No then B$=0.

INPUT Statement

The INPUT statement allows you to enter data in a program while the program is running. The variables to be input are listed in the INPUT statement, separated by commas. When an INPUT statement is executed, a question mark (?) appears as a prompt on the display screen and the program waits for you to type your input data. If more than one data item is requested by a single INPUT statement, the items must be input separated by commas. When you have finished typing the input data, press the ████ key. If you enter too many items, the message EXTRA IGNORED will be displayed in the message window. If you do not enter enough items, a double question mark (??) will be displayed as a prompt. You can then simply continue entering data.

If you enter string data when numeric data was requested, or numeric data when string data was requested in the INPUT list, the message "REDO FROM START" will be displayed. You must then re-enter all of the data items in the INPUT list.

Examples:

```
　　　　　variable list
100 INPUT A
110 INPUT C,D,S$

RUN ████
```

`?45 ████` A now equals 45.

`?1,2,California ████` C equals 1, D equals 2, and S$ equals California.

4-5

When data is read in, the input is taken from the line containing the cursor. All characters, including non-displaying control characters, to the right of the "?" prompt are input. This means that if additional characters are present on the input line, perhaps from some previous operation, they will also be input. These extra characters will probably result in an error.

Examples:

```
10 INPUT "Enter two numbers", A,B
20 PRINT "Their sum is";A+B

RUN RETURN

Enter two numbers 2.5, 5 RETURN
Their sum is 7.5

5   INTEGER X,L
10 INPUT "NO. OF COPIES?",X
15 INPUT "NO. OF LINES?",L
20 PRINT X; "COPIES AT ";L;" LINES EACH WILL BE";X*L;
              " LINES."

RUN RETURN

NO. OF COPIES?30        RETURN
NO. OF LINES?700        RETURN
30 COPIES AT 700 LINES EACH WILL BE 21000 LINES.
```

An INPUT statement causes the variables in the variable list to be assigned, in order, to the values supplied from the terminal during execution of the program.

Numeric values must be supplied for numeric variables; any values are acceptable for string variables; numbers are read as ASCII characters. Numeric data is automatically converted to match the type of the variable to which it is assigned.

Example:

```
5 LONG S
10 INPUT I,N$,S,L$
20 PRINT "I=";I,"N$=";N$,"S=";S,"L$=";L$

RUN RETURN

? 2236,JIM SOBEL,44,NEBRASKA RETURN

I=2236    N$=JIM SOBEL    S=44    L$=NEBRASKA
```

Strings may either be quoted or unquoted when supplied for an INPUT statement. A quoted string may contain any printable ASCII characters except the quote mark itself. An unquoted string may contain any printable ASCII characters except quote mark, or a comma. Leading blanks are ignored in an unquoted string. For example, using the same 3-line program listed above, respond to the input prompt as follows:

```
7, 6^6, 44, 00001 RETURN

I=7    N$=6^6    S=44    L$=00001
```

In this case, 6^6 and 00001 are both unquoted strings.

You can have a message printed instead of the "?" prompt by adding a quoted text string as the first item in the INPUT list. This can be used to tell the user what kind of data is required. Also, if you wish to display no prompt at all, you can do so by entering a null string ("") as the quoted message.

Rather than respond to a program requesting input, you can enter the break character (normally Control-A) to interrupt (or terminate) the program.

LINPUT Statement

The LINPUT statement reads an entire line of input from the screen. When the LINPUT statement is executed, a question mark (?) is displayed on the screen and the program waits for you to enter a line of characters. The line is then assigned to a specified string variable. Any ASCII character may be used including quotes and blanks.

Example:

```
        string variable
          ┌──┴──┐
40 LINPUT Name$
```
Reads a line of input and assigns it to the string variable Name$.

```
50 LINPUT Code$(12,2)
```
Reads a line of input and assigns it to the subscripted string variable Code$(12,2)

You may have a special prompt printed on the input device by inserting the prompt in quotes after the keyword LINPUT. The prompt and the string variable must be separated by a comma.

Example:

```
            prompt
          ┌──┴──┐
70 LINPUT "Address: " , A$
```
Prints the prompt Address: on the terminal screen. When a line of characters is entered, those characters are assigned to A$.

The string variable should be dimensioned large enough to accept the expected line of input.

If extra characters are entered, they are ignored (truncated).

Sample Program

```
10 DIM A$[20], B$[5]
```
A$ has a maximum length of 20 characters; B$ has a maximum length of 5 characters.

```
20 LINPUT "Type 20 characters:",A$
30 LINPUT "Type  5 characters:",B$
40 PRINT A$
50 PRINT B$

RUN RETURN

Type 20 characters: 12345678901234567890 RETURN
Type 5 characters: 1234567890 RETURN
12345678901234567890
12345
```
The last five characters input for B$ are ignored.

READ and DATA Statements

The READ statement reads data from DATA statements and assigns the data to variables in a variable list. The values in the variable list must be separated by commas.

Examples:

variable list

10 READ A   Reads a new value for A.

10 READ N$ Reads a new value for N$.

10 READ A,N$,T   Reads values for A,N$,and T.

DATA statements contain both string and numeric data for the READ statements. The values in the data list must be separated by commas.

Examples:

data list

10 DATA 10   Holds one numeric value, 10, or one string value, "10".

10 DATA "CALIFORNIA"   Holds one string value, "CALIFORNIA".

10 DATA 405,OREGON   Holds two values, one numeric (405) and one string ("OREGON") or two string values, "405" and "OREGON".

Strings in DATA statements may be quoted or unquoted except when the exclamation mark (!), comma (,), ASCII control character, or non-ASCII character is used. Leading spaces in an unquoted string are ignored. Because BASIC treats anything following an exclamation mark as a comment, to use an exclamation mark as part of a string in a DATA statement, the entire data item must be enclosed in quotes. The comma, ASCII control characters, and non-ASCII characters can also be used by surrounding them with quotes.

Example:

```
10 DATA 53, HELLO, GOODBYE !THIS IS A COMMENT
10 DATA 53, HELLO, GOODBYE,"!THIS IS NOT A COMMENT"
```

Statement 10 contains three data items: 53, "HELLO", and "GOODBYE"; statement 20 contains a fourth item, the string "!THIS IS NOT A COMMENT".

Interaction Between READ and DATA Statements

DATA statements may appear anywhere in a program. They need not come before or after the READ statement that references them. All of the data from every DATA statement in a program is linked together to form a single data list which acts effectively as an extended DATA statement. READ statements read data starting at the beginning of this extended DATA statement.

At the beginning of program execution, the data pointer is set to the beginning of the program. The first READ statement reads one data item for each variable in the read list. As data items are assigned, the pointer is advanced through the data list. The next READ statement begins reading data where the previous READ statement left off. It is important to

have enough data in DATA statements to supply all the variables in READ statements, otherwise an OUT OF DATA message will be printed and the program will halt. Extra data is ignored.

When DATA statements are used in subprograms, each subprogram maintains its own data pointer. This means that a READ statement in the main program or subprogram will only affect the data pointer in its own program unit. The data pointer for a subprogram is reset to the first data item each time the subprogram is entered. Additional information on subprograms is given in Section 8.

The following examples illustrate how READ and DATA statements work.

Examples:

```
10 DATA 3,5,7
20 READ A,B,C
30 PRINT A;B;C
```

RUN RETURN

```
3    5    7
```

```
5   INTEGER A,C
10 DATA 100
20 DATA FLORIDA
30 DATA 300
40 READ A,B$,C
50 PRINT A,B$,C
60 READ D
```

RUN RETURN

```
100         FLORIDA         300
END OF DATA IN LINE 60
```
There is no data left for the variable D.

It is also important that each variable in the READ statement be the same type as the corresponding value in the data list. Any data is valid for a string variable; only numeric data is valid for numeric variables.

```
5   SHORT A
10 DATA "1.0E20"
20 READ A
30 PRINT A
```

RUN RETURN   A is a numeric variable, but the only data item is a string.

```
DATA ERROR IN LINE 20
```

```
5   LONG A
10 DATA 1.0E20,1.0E20    The two data items appear identical, however, the first is read
20 READ A,A$             as a number and the second as a string of six characters.
30 PRINT A,A$
```

RUN RETURN

```
1E+20       1.0E20
```

RESTORE Statement

The RESTORE statement is used to select the DATA statement that is to be read by the next READ statement. In its simpler form, the RESTORE statement causes the next data read to be the first value of the first DATA statement in the program. The RESTORE statement restores or resets a pointer to the next data item to be read by BASIC. It affects only the data input using the READ statement. The pointer position is unchanged by INPUT, LINPUT, LINPUT #, or READ # statements.

The RESTORE statement allows you to reread the same set of data over and over again. It can also be used to change the order in which DATA statements are accessed. You can select the DATA statement to be used depending on a test made by your program.

Example:

```
5  INTEGER A,B,C,D,X,Y,Z
10 DATA 1,2
20 DATA 3,4
30 READ A,B,C,D          Statement 30 starts reading at statement 10.

40 RESTORE               Statement 50 starts reading at statement 10. If statement 40
50 READ X,Y,Z            were omitted, statement 50 would cause an END OF DATA
                         error.

60 PRINT A;B;C;D;X;Y;Z

RUN RETURN

1 2 3 4 1 2 3
```

Optionally you can specify a DATA statement line number so that the next READ statement starts reading at the first data item of the specified DATA statement. If the line number referenced in the RESTORE statement is not a DATA statement, the data pointer is advanced to the next DATA statement in the program.

Example:

```
5  INTEGER A,B,C,D,E,F,G
10 DATA 1,2
20 DATA 3,4
30 DATA 5,6
40 READ A,B,C            Statement 40 starts reading at statement 10.

50 RESTORE 20
60 READ D                Statement 60 starts reading at statement 20.

70 RESTORE 10
80 READ E,F,G            Statement 80 starts reading at statement 10.

90 PRINT A;B;C;D;E;F;G

RUN RETURN

1 2 3 3 1 2 3
```

Notice that in this example the statement

```
70 RESTORE 10
```

is equivalent to

```
70 RESTORE
```

because statement 10 is the first DATA statement in the program.

A general program can be used and new data can be supplied each time the program is run. This can be done by simply typing in new data statements. In the following program, statements 100 and 110 contain the data.

Example:

```
10   REM Average 6 numbers
20   FOR I=1 TO 6
30   READ N
40   S=S+N
50   NEXT I
60   PRINT "Average is ";S/6
70   RESTORE
80   READ A,B,C
90   PRINT "Average of the first 3 numbers is ";(A+B+C)/3
100 DATA 5.5,3.46,52
110 DATA 77.3,.89,50

>RUN RETURN

Average is 46.21
Average of the first 3 numbers is 20.32
```

New data can be added as follows:

```
>100 DATA 17.6,.009,305.67
>110 DATA 59.75,175.6,33.59
```

When executed the following averages will be calculated:

```
>RUN RETURN

Average is 98.7032
Average of the first 3 numbers is 107.76
>
```

PRINT Statement

The PRINT statement is used to display the results of a program. Normally, results are displayed on the screen. If you want to have data printed on another output device, you must use the ASSIGN and PRINT # statements described later in this manual.

Examples:

```
10 PRINT              ! Prints a blank line.
10 PRINT A,B          ! Prints the value of A, then the value of B.
10 PRINT A*B/6        ! Prints the value of the expression A*B/6.
10 PRINT "HELLO";N$   ! Prints HELLO followed by the value of N$.
```

The results to be displayed are specified in a print list. The print list consists of expressions separated by commas or semicolons. The items in the print list are printed in order from left to right.

When a comma separates two expressions, their values are printed spaced across the screen (or page) in fields of 15 spaces each. When an item is longer than 15 characters the extra characters are printed in the next field. Each line of output has five 15 character fields.

When a semicolon separates two expressions in the print list, the values are printed immediately adjacent to each other on the output device. Print list items can be formatted to output data in almost any configuration. Refer to the PRINT USING and PRINT # USING statements for additional information (Section 5).

Numbers are always printed with one trailing blank. Positive numbers are printed with a leading blank.

Sample Program

```
10 PRINT "THIS","LINE","USES","COMMAS"
20 PRINT "THIS";"LINE";"USES";"SEMI-COLONS"
30 PRINT "THIS ";"LINE ";"ALSO ";"USES ";"SEMI-COLONS"
40 PRINT "SEMICOLONS";1;2;3
50 PRINT "COMMAS",1,2,3

RUN RETURN

THIS           LINE           USES           COMMAS
THISLINEUSESSEMI-COLONS
THIS LINE ALSO USES SEMI-COLONS
SEMICOLONS 1   2   3
COMMAS         1              2              3
```

If there is not enough space to print all the items on one line, remaining items are printed on the following line. For example, change statements 10 and 20 to the following:

```
10 PRINT "USING ";"SEMIS ";"THIS ";"LINE ";"FITS ";"ON ";"ONE ";"LINE"
20 PRINT "USING ","COMMAS ", "THIS ","LINE ","WILL ","NOT ","FIT"

RUN RETURN

USING SEMIS THIS LINE FITS ON ONE LINE
USING          COMMAS         THIS           LINE           WILL
NOT            FIT
THIS LINE ALSO USES SEMI-COLONS
SEMICOLONS 1   2   3
COMMAS         1              2              3
```

If the last character in a PRINT statement is a comma or a semicolon, the next PRINT statement continues printing on that same line. Otherwise, the next PRINT statement begins a new line.

Sample Program

```
10 PRINT 1,2,3,
20 PRINT 4,5,6
30 PRINT 7,8,9;
40 PRINT 10,11,12
```

RUN **RETURN**

1	2	3	4	5
6				
7	8	9 10	11	12

Print Functions

Any of the following print functions can be included in a print list to control the format of a program's output:

- TAB
- SPA
- LIN

A print function is used in the print list just like a print list item. Print functions can be used only in the PRINT statement. They cannot be used in IMAGE or PRINT USING statements or in format specifications. (Refer to Section 5 for additional information on formatting output.)

In addition to the TAB, SPA, and LIN functions there are several special terminal functions that allow you to position the cursor. These MOVC functions are not present in most BASIC interpreters and can only be used with the terminal's display. The MOVC functions are described in Section 10.

TAB(X) — The TAB function prints spaces up to column X. X must be a positive number or a numeric expression that can be evaluated and then rounded to a positive integer between 0 and 80. Values between 81 and 255 will be reduced to 80 if the TAB function is used in a PRINT statement. If used in a PRINT # statement the limit for X is 255. Values greater than 255 or less than 0 will result in an error.

Example:

```
10 REM   THIS PROGRAM GENERATES A SINE WAVE PATTERN FOR THE VALUES
20 REM   OF 0 TO 360 DEGREES IN INCREMENTS OF 10 DEGREES (IN A BOX)
30 REM
40 Deg2rad=3.1415926/180            !   VAR TO CONVERT FROM DEG TO RAD
50 Scenter=30                       !   SCREEN-CENTER OF SINE WAVE
60 Slimit=24                        !   SINE WAVE DISTANCE LIMIT
70 PRINT RPT$("█",60)               !   60 DEL CHARACTERS (TOP OF BOX)
80 FOR Deg=0 TO 360 STEP 10         !   START OF LOOP:  0 TO 360 BY 10s
90 Rad=Deg*Deg2rad                  !   CONVERT DEGREES TO RADIANS
100 Spos=Scenter+Slimit*SIN(Rad)    !   COMPUTE SINE WAVE POSITION
110 PRINT "█"                       !   PRINT LEFT SIDE OF BOX
120 PRINT TAB(Spos);"*";            !   PLACE ASTERISK IN SINE WAVE POSITION
130 PRINT TAB(60);"█"               !   PRINT RIGHT SIDE OF BOX
140 NEXT Deg                        !   END-OF-LOOP
150 PRINT RPT$("█",60)              !   PRINT BOTTOM OF BOX
160 END

>RUN
```
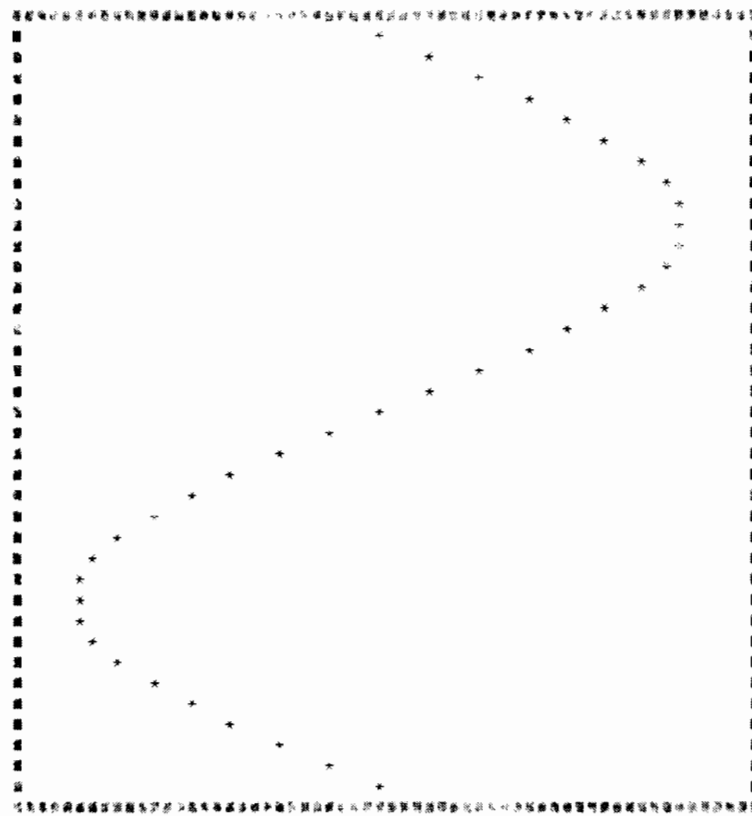

If the value of X is less than the current print position, the TAB function is ignored.

SPA(X) — Blanks are printed for the number of spaces indicated by the numeric expression X. If the number of spaces will not fit on the current line, the remaining spaces are sent to the beginning of the next line. For example, 10 PRINT A;SPA(10);B prints the value of A, prints 10 additional blanks and then prints the value of B.

Example:

```
5   REM: This program prints a 20 by 20 asterisk box
10  PRINT "********************"
20  FOR I = 1 TO 18
30  PRINT "*";SPA(18);"*"
40  NEXT I
50  PRINT "********************"
```

RUN `RETURN`

```
********************
*                  *
*                  *
*                  *
*                  *
*                  *
*                  *
*                  *
*                  *
*                  *
*                  *
*                  *
*                  *
*                  *
*                  *
*                  *
*                  *
*                  *
*                  *
********************
>
```

LIN(X) — The number of lines specified by the numeric expression X are advanced and the print position returns to the left margin. If X is negative, then ABS(X) lines are advanced but the horizontal print position remains the same (the column remains the same instead of returning to the left margin). Note that $+0$ and -0 both result in a carriage return with no line feed.

Examples:

```
10 LET S$ = "SAN FRANCISCO"
20 PRINT S$,LIN(1),S$,LIN(-1),S$,LIN(-10),S$
```

RUN ◄

```
SAN FRANCISCO
SAN FRANCISCO
              SAN FRANCISCO
```

```
                                          SAN FRANCISCO
```

A typical use of the three print functions is to provide header information for a report:

```
10 PRINT TAB(10),"SUMMARY REPORT",SPA(15),"PAGE 1"
20 PRINT LIN(3), "DETAIL LINES"
```

You can specify more or fewer digits, or delete the trailing spaces after numbers, or otherwise overcome print conventions with the PRINT USING statement and the IMAGE statement. Refer to Section 5 for the discussions of these statements and for more information on formatted output.

GOTO and ON...GOTO Statements

The GOTO and ON...GOTO statements are used to override the normal sequential order of processing statements by transferring control to a specified line number. The GOTO statement can transfer control (branch) to a single line number.

The ON<expression>GOTO<linelist> form of the GOTO statement allows you to branch to one of several line numbers. When the statement is executed, the expression (which must be numeric) is evaluated and rounded to an integer. This integer is used to select one of the line numbers from the line list. A "1" selects the first line number in the line list, a "2" selects the second line number, and so on. If the expression evaluates to less than one or to greater than the number of lines in the line list, control passes to the statement following the ON GOTO statement.

Examples:

```
        label list
            \
100 GOTO 10
```
Transfers control to line number 10.

```
110 GOTO 20
```
Transfers control to line number 20.

```
      expression
       ┌─┴─┐
120 ON A*B GOTO 60,50,75
```
This is called a "computed GOTO". The expression A*B is evaluated and rounded to the nearest integer. If the resulting value is 1, control is transferred to line number 60; if the value of A*B is 2, control is transferred to line number 50; if the value is 3, control is transferred to line number 75. If the value of the expression is less than 1 or greater than the number of items in the line list, then processing continues with the following line.

This example shows a simple GOTO in line 200 and a multibranch, or computed GOTO in line 600.

```
100 LET I  = 0
200 GOTO 600
300 PRINT I
400 REMARK THE VALUE OF I IS ZERO
500 LET I = I+1
600 ON I+1 GOTO 300,500,800
700 REM THE FINAL VALUE OF I IS 2
800 PRINT I
```

This program prints the initial value of I (0) and the final value of I (2).

If the GOTO statement references a non-existent line number, an error will be displayed, and the program will halt. If the GOTO statement references a non-executable statement such as REM, execution will continue with the statement following the non-executable statement.

Sample Program

```
10 FOR I=0 TO 5 STEP .4
20 PRINT "I=";I;
30 ON I GOTO 60,70,80
35 PRINT "I is too large or too small"
40 NEXT I
50 END
60 PRINT " Went to 1"\GOTO 40
70 PRINT " Went to 2"\GOTO 40
80 PRINT " Went to 3"\GOTO 40
```

>RUN `RETURN`

```
I= 0    I is too large or too small
I=  .4  I is too large or too small
I=  .8  Went to 1
I= 1.2  Went to 1
I= 1.6  Went to 2
I= 2    Went to 2
I= 2.4  Went to 2
I= 2.8  Went to 3
I= 3.2  Went to 3
I= 3.6  I is too large or too small
I= 4    I is too large or too small
I= 4.4  I is too large or too small
I= 4.8  I is too large or too small
>
```

IF...THEN...ELSE Statement

The IF...THEN statement allows you to branch on a condition. When an IF...THEN statement is executed, a logical expression is tested. If the expression is TRUE (non-zero), the program transfers control to a statement specified by the line number or executes the specified statement.

Examples:

```
          expression          label
          ┌──┴──┐             ╱
    20 IF A = B THEN 10 ╱      If the expression A=B is TRUE, then control
                               passes to statement 10.
```

If the expression A=B is TRUE, then control passes to statement 10.

```
    30 IF A-B+C THEN A=B
                    └─┬─┘
    statement ╱
              ┌──┴──┐
    40 IF A THEN PRINT B
```

If the expression A−B+C is TRUE, then the value of B is assigned to A. (Note: A−B+C is a logical expression. See Section 3.)

If A is TRUE, then the value of B is printed.

If the expression following IF is FALSE, control passes to the next sequential statement and the statement following THEN is ignored.

The IF...THEN...ELSE form of the statement allows you to select alternate statements based on a condition. If the condition is true, the statement following THEN is executed. If the condition is false, the statement following ELSE is executed.

All statements following the IF...THEN statement in the line are assumed to be a part of the IF...THEN statement or its conditions. A multiple statement following the THEN will be executed only if the condition is true. A multiple statement following the ELSE will be executed only if the condition is false.

Example:

```
    10 IF A+1>3 THEN PRINT 10\Y=2   ELSE PRINT 20\X=3
    40 PRINT "DONE"
```

will execute as if it were written as follows:

```
    10 IF A+1>3 THEN 30
    20 PRINT 20 \ X=3 \ GOTO 40
    30 PRINT 10 \ Y=2
    40 PRINT "DONE"
```

The following BASIC statements should not be used in the THEN or ELSE part of an IF statement:

DATA	REM	SUBEND
IMAGE	SUB	type declaration statements

The SUB statement will cause a program error and the others may produce unanticipated results elsewhere in the program.

Example:

```
10 REM...The following program computes
20 REM...the amount of overtime money
30 REM...earned by an employee
40 INTEGER Hours,Over
50 LONG Pay
60 PRINT "hours worked?"
70 INPUT Hours
80 PRINT "pay rate?"
90 INPUT Pay
100 IF Hours>40 THEN 140
110 IF Hours=0 THEN 170
120 PRINT "no overtime returned"
130 GOTO 60
140 Over=Hours-40
150 PRINT "overtime pay=";Over*Pay*1.5
160 GOTO 60
170 SUB POINT
180 REM...The point of this subroutine
190 REM...is to show that the sub
200 REM...statement cannot be used as
210 REM...part of an IF,THEN statement.
220 PRINT "this program got to line 170"
230 SUBEND
>run
hours worked?
? 40
pay rate?
? 8.00
no overtime returned
hours worked?
? 65
pay rate?
? 6.50
overtime pay= 243.75
hours worked?
? C
pay rate?
? 0
>

UNDEFINED LINE NUMBER IN LINE 110
```

FOR and NEXT Statements

The FOR and NEXT statements allow you to cause a section of your program to execute more than once. The FOR statement indicates the beginning of the group of statements that is to be repeated and the number of times to repeat that group. The NEXT statement indicates the end of the group. A group of statements bounded by a FOR statement and its corresponding NEXT statement is called a FOR loop. Although the IF and GOTO statements can usually be used to perform the same type of looping as the FOR and NEXT statements, the FOR and NEXT statements are normally easier to use and understand.

The start, end, and step size values can be expressions.

```
        loop variable
             \   initial     final
              \  value       value      step value
               \     \         /        /
    10 FOR I = 1 TO N STEP 10
       .       .
       .       .
       .       .
FOR loop  .    .
       .       .
       .       .
       .       .
       .            loop variable
       .          /
    100 NEXT I
```

The FOR statement and the corresponding NEXT statement are linked together by the loop variable. The loop variable must be a SHORT or INTEGER simple variable. When the FOR loop is first executed, the loop variable is set to an initial value. The loop is then executed. At the end of the FOR loop, the loop variable is modified by a step value and the value of the loop variable is tested against the final value. The FOR loop repeats until the value of the loop variable exceeds the final value. When the step value is negative, the loop terminates when the loop variable is less than the final value.

Note that the FOR loop is always executed at least once.

In the example above, the loop variable is I, and the initial value of the loop variable is 1. After each execution of the FOR loop, the value of the loop variable I is incremented by the step value of 10. When the value of I becomes greater than the final value, N, execution of the FOR loop ends. If N is less than the initial value, 1, the FOR loop executes only once.

Example:

```
10 FOR J=N TO 1 STEP -1
   .
   .
   .
   .
   .
100 NEXT J
```

In this example, the loop variable is being decremented, so execution of the FOR loop ceases when the loop variable J becomes less than 1.

The initial value, final value, and step value can be any valid numeric expression. When the FOR loop is first encountered during program execution, the initial value, final value, and step value are calculated and those values are used throughout the execution of the FOR loop. For example, assume that Y=10. The following FOR loop will execute 10 times, not 20 times.

```
100 LET Y=10
110 FOR I=1 TO Y STEP 1
120 PRINT "OUTPUT REPEATED TEN TIMES"
130 Y=20
140 NEXT I
```

The value of the loop variable, however, can be changed inside the loop. The following FOR loop will execute 1 time, not 10.

```
100 FOR I=1 TO 10 STEP 1
110 PRINT "REPEAT OUTPUT"
120 I=10
130 NEXT I
```

The step value may be omitted in which case it is assumed to be 1. For instance, the statement

```
100 FOR I=1 TO STEP 1
```

is equivalent to

```
100 FOR I=1 TO 10
```

The following program uses a FOR...NEXT loop to calculate and print cumulative savings over a 10 year period on an initial deposit of $1000 at 4-1/2% interest compounded annually.

Sample Program

```
20 LONG T,D
30 I=.045
40 T,D=1000
50 FOR Y=1 TO 10
60   T=I*T+T
70   PRINT "AFTER";Y;"YEAR(S), TOTAL = ";T
80 NEXT Y

RUN RETURN

AFTER 1 YEAR(S), TOTAL = 1044.999998062849
AFTER 2 YEAR(S), TOTAL = 1092.024995951355
AFTER 3 YEAR(S), TOTAL = 1141.166118653748
AFTER 4 YEAR(S), TOTAL = 1192.518591782556
AFTER 5 YEAR(S), TOTAL = 1246.181926102682
AFTER 6 YEAR(S), TOTAL = 1302.260110363261
AFTER 7 YEAR(S), TOTAL = 1360.861812806933
AFTER 8 YEAR(S), TOTAL = 1422.10059174705
AFTER 9 YEAR(S), TOTAL = 1486.095115620844
AFTER 10 YEAR(S), TOTAL = 1552.969392944991
>
```

Multiple Loops

It is possible to have more than one FOR loop in a program. If this is the case, the FOR loops must either be disjoint (completely separated) or one must be nested within the other. Following is an example of two disjoint FOR loops.

Sample Program

```
10 FOR I=1 TO 5
20 PRINT I;
30 NEXT I
40 FOR I=10 TO 100 STEP 20

50 PRINT I;
60 NEXT I

RUN  RETURN

 1  2  3  4  5  10  30  50  70  90
```

Notice that the variable I is the loop variable in both of the above FOR loops. When FOR loops are disjoint, identical loop variables may be used.

This next example shows a nested FOR loop. When FOR loops are nested one within the other, different loop variables must be used. The J FOR loop here is nested within the I FOR loop.

Sample Program

```
10 FOR I=1 TO 5
20 FOR J=10 TO 100 STEP 20
30 PRINT I;J;
40 NEXT J
45 PRINT
50 NEXT I

RUN  RETURN

 1  10  1  30  1  50  1  70  1  90
 2  10  2  30  2  50  2  70  2  90
 3  10  3  30  3  50  3  70  3  90
 4  10  4  30  4  50  4  70  4  90
 5  10  5  30  5  50  5  70  5  90
```

FOR loops that overlap are not permitted. The following is an example of an overlapping FOR loop.

```
            ┌10 FOR I=1 TO 10
            ┼20 FOR J=10 TO 100
INVALID     ││30 PRINT I
            └40 NEXT I
             50 PRINT J
            └60 NEXT J
```

The J FOR loop begins inside the I FOR loop but ends outside the I FOR loop, so the J FOR loop overlaps the I FOR loop.

The following program uses multiple loops to calculate and print the cumulative savings over a 10 year period on initial deposits of $100 to $1000 at 4-1/2% interest per year.

Sample Program

```
10 LET I=.045
20 FOR D=100 TO 1000 STEP 100
30 PRINT "FOR AN INITIAL DEPOSIT OF $";D
40 LET T=D
50 FOR Y=1 TO 10
60 LET T=I*T+T
70 PRINT "AFTER";Y;"YEAR(S), TOTAL =";T
80 NEXT Y
90 NEXT D
```

RUN `RETURN`

```
FOR AN INITIAL DEPOSIT OF $ 100
AFTER 1 YEAR(S), TOTAL = 104.5
AFTER 2 YEAR(S), TOTAL = 109.203
AFTER 3 YEAR(S), TOTAL = 114.117
AFTER 4 YEAR(S), TOTAL = 119.252
AFTER 5 YEAR(S), TOTAL = 124.618
AFTER 6 YEAR(S), TOTAL = 130.226
AFTER 7 YEAR(S), TOTAL = 136.086
AFTER 8 YEAR(S), TOTAL = 142.21
AFTER 9 YEAR(S), TOTAL = 148.61
AFTER 10 YEAR(S), TOTAL = 155.297
      .
      .
      .
FOR AN INITIAL DEPOSIT OF $ 1000
AFTER 1 YEAR(S), TOTAL = 1045
AFTER 2 YEAR(S), TOTAL = 1092.03
AFTER 3 YEAR(S), TOTAL = 1141.17
AFTER 4 YEAR(S), TOTAL = 1192.52
AFTER 5 YEAR(S), TOTAL = 1246.18
AFTER 6 YEAR(S), TOTAL = 1302.26
AFTER 7 YEAR(S), TOTAL = 1360.86
AFTER 8 YEAR(S), TOTAL = 1422.1
AFTER 9 YEAR(S), TOTAL = 1486.1
AFTER 10 YEAR(S), TOTAL = 1552.97
>
```

FOR Loop Cautions

FOR loops must always be entered from the top of the loop, that is, starting with the FOR statement. Never write a program that contains a statement which branches into the middle of a FOR loop. You may begin a FOR loop, branch out of the loop, and come back. However, if you do you must be sure that the loop variable has not been modified outside of the loop. If it has, it will affect the operation of the loop. A clever programmer may wish to take advantage of this feature, but it is extremely difficult to debug clever programs.

For instance, in the following example, the loop variable is I and there is also a variable I outside the loop at statement 200. Statement 200 will cause the program to loop indefinitely.

Example:

```
        .
        .
        .
40  LET K=3
50  FOR I=1 TO 10
60  IF I=K THEN 200
```

When I becomes 3, branch to statement 200. Statement 200 changes the value of I to 2 then the program branches back to statement 70. The value of I becomes 3, statement 60 branches to statement 200 and the cycle repeats . . . and repeats . . . and repeats . . .

```
70  NEXT I
80  STOP
        .
        .
        .
200 LET I=I-1
205 PRINT "LOOP"
210 GOTO 70
        .
        .
        .
```

The best approach is to reserve loop variables for use within the loops only, and not reference these variables outside of the loop.

You can terminate a FOR loop by branching out of the loop with a transfer statement such as GOTO or IF...THEN. For example, the following program is terminated as soon as the value of T is greater than or equal to 1000. This program finds the smallest amount, in multiples of $100, that must be deposited if a person wants to have $1000 within 10 years.

Sample Program

```
10  LET I=.045
20  FOR D=100 TO 1000 STEP 100
30  LET T=D
40  FOR Y=1 TO 10
50  LET T=I*T+T
60  IF T>=1000 THEN 85
70  NEXT Y
80  NEXT D
85  PRINT "FOR AN INITIAL DEPOSIT OF";D
90  PRINT "AFTER";Y;"YEAR(S), TOTAL=";T

RUN  RETURN

FOR AN INITIAL DEPOSIT OF .....
        .
        .
        .
AFTER 9 YEAR(S), TOTAL= 1040.26
```

When you are using FOR loops, be careful to avoid endless loops. An endless loop occurs when a statement within the FOR loop causes the value of the loop variable to always be smaller than the final value. The following is an example of a FOR loop that will never end.

```
10 FOR X=1 TO 10
20 X=1
30 NEXT X
```

X is always reset to 1 in statement 20, so the loop will never end.

GOSUB and RETURN Statements

You may want to perform the same sequence of instructions in many different places within the same program. Instead of typing this block of statements every time you want to use it, you can write it once as a subroutine and use GOSUB statements to call the subroutine whenever you need it. (For a long sequence of instructions, you should write a subprogram. See Section 8.) Figure 4-1 illustrates the general structure of a program with a subroutine.

Since BASIC subroutines are simply a collection of statements, it is a good idea to place them where they will not be executed accidentally. If you place a subroutine in the middle of your program for example, you will need to use a GOTO statement or some other technique to route the execution of your program around the subroutine. It is also helpful to clearly describe the purpose of the subroutine together with any parameters in REM statements at the beginning of the subroutine. If the RETURN statement is not the last statement in the subroutine it is also helpful to indicate the last statement of the subroutine with a comment or remark.

```
10 .
20 .
30 .
40 .
50 GOSUB 200       line number corresponding to first statement of
                   subroutine.
  .           .
90 GOSUB 200
  .
  .
  .
140 GOSUB 200
  .
  .
  .
190 GOTO 310
  ┌200
  │ .
  │ .
  │ .
subroutine ┤ .
  │ .
  │ .
  │ .
  └300 RETURN
   310
```

Figure 4-1. Structure of a Program with a Subroutine

The GOSUB statement specifies the line number of the first statement of the subroutine. The RETURN statement indicates the end of the subroutine. When a subroutine is finished processing, control returns to the first statement following the GOSUB statement. Note that the subroutine in Figure 4-1 is preceded by a GOTO statement; a subroutine should only be entered with GOSUB statements.

Example:

```
  .
  .
  .
  .
40 GOSUB 500                Transfers to subroutine at statement 500.

50 ....
  .
  .
  .
  .
  .
  .
  .
500 REM:BEGIN SUBROUTINE
  .
  .
  .

590 RETURN                  Transfers to statement 50 (the first statement)
                            following the GOSUB statement).
```

The following program generates a digital clock on the display. It uses nested subroutines to input the data values and to check if they are valid. Note that the values used to check for valid input are different for hours, minutes, and seconds and are set before entering the subroutine.

```
10 REM   THIS PROGRAM SHOWS A DIGITAL CLOCK ON THE SCREEN
20 REM
30 REM   VARIABLES LIST:
40 REM   QUEST$:   CONTAINS QUESTION TO BE ASKED
50 REM   X     :   TO CONTAIN USER'S ANSWER
60 REM   H,M,S :   TO CONTAIN HOURS, MINUTES, AND SECONDS
70 REM   TI$   :   CONTAINS EITHER 'AM' OR 'PM'
80 REM   LL    :   LOWER-LIMIT FOR ANSWER INPUT
90 REM   UL    :   UPPER-LIMIT FOR ANSWER INPUT
100 REM
110 DIM Quest$[100]
120 REM   GET HOUR FROM USER
130 Quest$="WHAT IS THE CURRENT HOUR"
140 L1=0\U1=23\ GOSUB 1000
150 H=X
160 IF H=0 OR H>12 THEN Am=(H<12 OR H=0)\H=H MOD 12\ GOTO 220
170 INPUT "IS THAT 'AM' ? ",A$
180 A$=UPC$(A$)
190 IF POS("YES",A$)=1 THEN Ti$="AM"\ GOTO 220
200 IF POS("NO",A$)=1 THEN Ti$="PM"\ GOTO 220
210 PRINT "ERROR: PLEASE ANSWER YES OR NO ."\ GOTO 170
220 REM   GET MINUTES FROM USER
230 Quest$="WHAT IS THE CURRENT MINUTE"
```

```
240 L1=0\U1=59\ GOSUB 1000
250 M=X
260 REM   GET SECONDS FROM USER
270 Quest$="WHAT IS THE CURRENT SECOND"
280 L1=0\U1=59\ GOSUB 1000
290 S=X
300 REM   GOSUB TO SET TERMINAL TIME
310 GOSUB 2000
320 REM   WARN USER ABOUT HITTING CONTROL-A BREAK
330 PRINT LIN(1);"NOTE:  YOU MUST HIT CONTROL-A TO STOP THIS PROGRAM"
340 REM   START OF LOOP
350 GOSUB 3000   !  DISPLAY CURRENT TIME
360 GOTO 340
1000 REM   INPUT RESPONSE FROM USER AND CHECK
1010 REM   NUMBER INPUT MUST BE BETWEEN LL AND UL
1020 REM   RETURNS ANSWER IN THE VARIABLE 'X'
1030 PRINT Quest$;
1040 INPUT X
1050 IF X=INT(X) AND X>=L1 AND X<=U1 THEN RETURN
1060 PRINT "ERROR:  THAT IS NOT A LEGAL NUMBER.  TRY AGAIN."
1070 GOTO 1030
2000 REM SET TIME GIVEN H,M,S AND AM VARIABLE
2010 COMMAND "SET TIME "&VAL$(H)&":"&VAL$(M)&":"&VAL$(S)&" "&Ti$
2020 RETURN
3000 REM   DISPLAY TIME
3010 COMMAND "SHOW TIME"
3020 PRINT MOVCR(-2,0); ! MOVE UP TWO LINES TO OVER-WRITE PREVIOUS LINE
3030 RETURN
9999 END

>RUN
WHAT IS THE CURRENT HOUR? 10
IS THAT 'AM' ? Y
WHAT IS THE CURRENT MINUTE? 25
WHAT IS THE CURRENT SECOND? 30

NOTE:  YOU MUST HIT CONTROL-A TO STOP THIS PROGRAM

 10:25:34 AM
```

ON...GOSUB Statement

You can select one of several subroutines in the same manner as the ON...GOTO statement described earlier. When the ON <expression> GOSUB <line list> form of the statement is used, the numeric expression is evaluated and rounded to an integer to select a line number from the line list. If the expression evaluates to less than one or to a value greater than the number of lines in the line list, the next statement following the ON...GOSUB statement will be executed.

Example:

```
        expression        statement labels
          ┌───┴───┐      ┌──────────┴──────────┐
10 ON X + Y/3 GOSUB 100,500,300,40,100
```

The expression is evaluated and rounded to the nearest integer. If the resulting value is 1, control is transferred to statement 100; if the value is 2, control is transferred to statement 500; etc.

If the expression evaluates to less than 1, or to greater than the number of line numbers specified, control passes to the statement following the ON...GOSUB. When a RETURN statement is executed, control returns to the statement following the ON...GOSUB statement.

Sample Program

```
10 FOR X=0 TO 2
20 ON X+1 GOSUB 200,300,400
30 NEXT X
100 REM...CONTROL WILL REACH HERE ONLY WHEN THE
105 REM...FOR LOOP IS FINISHED
110 PRINT "FOR LOOP IS FINISHED"
120 STOP
200 PRINT X;SIN(X)
210 RETURN
300 PRINT X;2*X;COS(X)
310 RETURN
400 PRINT X;3*X;TAN(X)
410 RETURN
500 END

RUN  RETURN

0   0
1   2   .540302
2   6  -2.18504
FOR LOOP IS FINISHED
```

NOTE: If the above output looks unfamiliar, remember that in BASIC/47 the sin, cos, and tan functions operate on variables expressed in radians rather than degrees.

In-Line Subroutine Nesting

The following program and its output illustrate subroutine nesting. Note that control will not return to the main routine until all intervening RETURN statements have been executed.

```
10 REM    THIS PROGRAM PRINTS A CONTINUOUS TABLE OF RANDOM NUMBERS
20 REM    (IN GROUPS OF 10) UNTIL THE USER REQUESTS IT TO STOP
30 REM
40 REM    THE NESTED GOSUB ROUTINES BELOW WORK AS FOLLOWS:
50 REM
60 REM    1)  MAIN ROUTINE CALLS SUBROUTINE #1 (LINE 1000)
70 REM    2)  SUBROUTINE #1 CALLS SUBROUTINE #2 (LINE 2000)
80 REM    3)  WHEN SUBROUTINE #2 FINISHES, IT RETURNS TO SUBROUTINE #1
90 REM        (THE ROUTINE WHICH CALLED IT)
100 REM   4)  WHEN SUBROUTINE #1 FINISHES, IT RETURNS TO THE MAIN ROUTINE
110 REM        (THE ROUTINE WHICH CALLED IT)
120 REM
130 REM   PRINT LIST OF 10 RANDOM NUMBERS
140 PRINT "LIST OF TEN 3-DIGIT RANDOM NUMBERS:"! HEADING
150 FOR N=1 TO 10                           ! START OF LOOP
160 PRINT INT(1000*RND),                    ! PRINT RANDOM NUMBER
170 NEXT N                                  ! END OF LOOP
180 PRINT                                   ! BLANK LINE
```

```
190 REM  ASK USER IF HE WANTS TO CONTINUE
200 Question$="CONTINUE"                        ! LOAD QUESTION STRING
210 GOSUB 1000                                  ! ASK AND GET ANSWER
220 IF True THEN 130                             ! IF YES, DO IT AGAIN
230 PRINT "END-OF-PROGRAM"                       ! END OF MAIN ROUTINE
240 END
1000 REM ████████████ SUBROUTINE:  GET YES/NO FROM USER ████████████
1010 PRINT Question$;                           ! DISPLAY QUESTION
1020 INPUT Answer$                              ! GET ANSWER
1030 GOSUB 2000                                 ! GOSUB: CLEAN UP ANSWER
1040 IF POS("YES",Answer$)=1 THEN 1080          ! Y, YE OR YES?
1050 IF POS("NO",Answer$)=1 THEN 1090           ! N OR NO?
1060 PRINT "PLEASE ANSWER  YES  OR  NO ."       ! ILLEGAL RESPONSE
1070 GOTO 1000                                  ! GOTO ASK AGAIN
1080 True=1\ GOTO 1100                          ! SET ANSWER-FLAG TO YES
1090 True=0                                     ! SET ANSWER-FLAG TO NO
1100 RETURN                                     ! RETURN TO MAIN ROUTINE
2000 REM ████████████ SUBROUTINE:  CLEAN UP ANSWER$ ████████████████
2010 REM  REMOVE BLANKS FROM ANSWER$
2020 Blank=POS(Answer$," ")                     ! FIND POSSIBLE BLANK
2030 IF Blank=0 THEN 2060                        ! NO BLANK FOUND???
2040 Answer $[Blank]=Answer$[Blank+1]           ! REMOVE BLANK
2050 GOTO 2010                                  ! LOOK FOR ANOTHER
2060 Answer$=UPC$(Answer$)                      ! LOWER-CASE TO UPPER-CASE
2070 RETURN                                     ! RETURN TO ROUTINE WHICH
9999 END                                        !      CALLED THIS ONE

RUN
LIST OF TEN 3-DIGIT RANDOM NUMBERS:
 894          101          865          800          989
 364          211          261          692          937

CONTINUE? y
LIST OF TEN 3-DIGIT RANDOM NUMBERS:
 391          673          707          449          848
 190          672          117          936          349

CONTINUE? 5q
PLEASE ANSWER  YES  OR  NO .
CONTINUE? Ye
LIST OF TEN 3-DIGIT RANDOM NUMBERS:
 801          469          270          390          185
 550          970          544          517          474

CONTINUE?  n o
END-OF-PROGRAM
>
```

STOP and END

The STOP and END statements are used to terminate a program. If a program contains a subprogram (see Section 8), the END statement must precede the first subprogram. STOP statements may appear anywhere in a program. When a program halts as a result of a STOP statement, the effect is the same as that of a program BREAK. The program can be resumed by entering the GO command. Execution continues with the statement following the STOP statement that was used to halt the program.

Examples:

```
10 INPUT A
20 IF A=0 THEN STOP
```
Statement 20 causes this program to stop when a 0 is input for A.

```
30 PRINT A,.065*A
40 GOTO 10
```

```
10 INPUT A
20 IF A=0 THEN GOTO 50
```
When a 0 is input for A in this program, statement 20 passes control to statement 50 and the program terminates.

```
30 PRINT A,.065*A
40 GOTO 10
50 END
```

Sample Program

```
10 REM: This program asks for a name and address,
20 REM: then prints it out in mailing format.
30 DIM Name$[30],Street$[30],City$[20],Zip$[5]
40 PRINT "Please type the name. Type an X to stop."
50 INPUT Name$
60 IF (Name$="X" OR Name$="x") THEN 140
70 PRINT "Now type the street address, city, and zip code."
80 PRINT "Separate them with commas.";LIN(1)
90 INPUT Street$,City$,Zip$
100 PRINT SPA(20),Name$
110 PRINT SPA(20),Street$
120 PRINT SPA(20),City$;",California ";Zip$;LIN(1)
130 GOTO 40
140 END
```

```
>RUN RETURN
Please type the name. Type an X to stop.
? Mary Ann Oullette
Now type the street address, city, and zip code.
Separate them with commas.

? 1 Grove Place,Cupertino,99999
                    Mary Ann Oullette
                    1 Grove Place
                    Cupertino, California 99999

Please type the name. Type an X to stop.
? x
>
```

Formatted Output_____5

Some programs require output to be printed in a complex format. The PRINT USING and IMAGE statements allow you to specify, with more flexibility than the simple PRINT statement, the format in which program results will be displayed or printed.

Recall from Section 4 that in the PRINT statement:

- When commas are used as item separators, results are printed in evenly spaced fields across the screen or page.

- When semicolons are used as item separators, results are printed close together.

- Each numeric value is printed with a leading blank if positive, and a trailing blank.

- The TAB function moves the print position to a specified column.

- The LIN function generates a specified number of blank lines.

- The SPA function moves the print position a specified number of columns to the right.

PRINT USING and IMAGE Statements

The PRINT USING statement extends your control of the output format to include the following capabilities:

- You can eliminate leading and trailing blanks in numeric output.

- You can insert periods and commas in numeric output.

- You can specify placement of plus and minus signs.

- You can designate precisely where on the page (screen) you want each data item printed.

Sample Program

```
  5 INTEGER Day, Year
 10 INPUT Weekday$, Month$, Day, Year
 20 PRINT Weekday$, Month$, Day, Year
 30 PRINT Weekday$, Month$; Day, Year
 40 PRINT SPA(19); Weekday$, LIN(1); TAB(50);Month$;Day,SPA(3);Year
```

RUN [RETURN]

?Monday,March,13,1978 [RETURN]

Monday	March	13	1978
Monday	March 13	1978	
	Monday	March 13	1978

The PRINT USING statement consists of a list of items to be printed (the print list) and a format string that describes how these items are to be printed. The format string is separated from the print list by a semicolon. The format string can be specified in one of four ways as illustrated in the following examples and explained below:

Examples:

A. The format string is included in the PRINT USING statement:

```
                    format string        print list
              ┌─────────────────────┐  ┌─────────┐
20 PRINT USING "3A,3X,2D,2X,4D";   A$,B,C
```

B. The format string is represented by a string variable in the PRINT USING statement:

```
           format string
      ┌─────────────────────┐
10 P$="3A,3X,2D,2X,4D"
```

```
      string variable
                   \       print list
                    \    ┌───────┐
20 PRINT USING P$;A$,B,C
```

C. The format string is specified in an IMAGE statement. The PRINT USING statement references the IMAGE statement line number:

```
    IMAGE statement
         label
             \       print list
              \    ┌───────┐
20 PRINT USING 30;A$,B,C
```

```
         format string
         ┌─────────────────┐
30 IMAGE 3A,3X,2D,2X,4D
```

Note that the IMAGE statement does not use quote marks around the format string. Also the IMAGE statement must be the only statement used in the line.

D. The format string is represented by a string expression in the PRINT USING statement:

```
10 DIM A$[80]
20 A$="THIS IS HOW YOU GET A"
30 PRINT USING "22A"&CHR$(34)&"QUOTED LITERAL"&CHR$(34);A$
```

run `RETURN`

```
THIS IS HOW YOU GET A "QUOTED LITERAL"
```

The print list can include quoted strings, variables, and expressions. In the first three examples, the print list includes three variables: A$, B, and C. The items in the print list are separated by commas or semicolons. The commas and semicolons in PRINT USING statements are interpreted only as item separators and do not affect the format of the output.

Format Symbols

A format consists of a series of format symbols. The format symbols specify how each item in the print list is to be formatted and how the items are to be arranged on the page. Commas separate individual format specifications in the format string. The five specifications in the format string "3A,3X,2D,2X,4D" indicate that the values of A$, B, and C are to be printed as three ASCII characters (3A) followed by three blank spaces (3X), two decimal digits (2D), two more blank spaces (2X), and four more decimal digits (4D). There must be a format specification for every variable in the print list. Additional specifications may be used to insert spaces and control carriage returns and linefeeds.

Sample program:

```
10 INTEGER B,C
20 LET A$="JUNE"
30 LET B=14
40 C=1976
50 PRINT USING 60;A$,B,C
60 IMAGE 4X,4A,2X,2D,2X,4D
```

RUN [RETURN]

```
     JUNE  14  1976
```

The first step in preparing formatted output is usually to make a sketch of how you want the results arranged on the page or screen. Then build the format string using the symbols presented in Table 5-1. The following pages describe how these symbols are used to format string and numeric output.

Separators

Three symbols are used to separate specifications:

,   A comma is used only to separate two specifications.

/   A slash separates two specifications and begins a new line if writing to a lineprinter or a new record if writing to a file.

@   The @ sign separates two specifications and, on lineprinter output, begins a new page.

/ and @ can also be used as specifications by themselves; that is, they can be separated from other specifications by a comma. Only the / can be directly replicated, however, as explained later.

Table 5-1. Format Symbols

Symbol	Description	Example
Strings		
A	ASCII Character	AAA
K	Compacted format	K
Blanks		
X	Blank space	XXX
Separators		
,	Separator only	AA,DD
/	Separates specifications and begins a new line or a new record if writing to a file.	AA/DD
@	Separates specifications and, on lineprinter output, begins a new page.	AA@DD
Carriage Control	Characters (used at the end of the print list)	
−	Suppress carriage return	
+	Suppress linefeed	
#	Suppress both linefeed and carriage return	
Replicators		
n	Single replicator	3A
n( )	Group replicator	3(3A,2X)
Numeric Specifications		
S	Sign character (+ or −)	SDD
M	Minus sign	MDD
D	Numeric digit, blank fill	DDD
.	Decimal point (.)	DDD.DD
R	European Decimal point (.)	DDDRDD
C	Comma	DDCDDD
P	Period (European comma)	DDPDDD
K	Compacted format	"ONLY",XKX, "ITEMS"

Literal Specifications

Literal specifications can be included in the format specifications in two ways:

- nX specifies n blank spaces (x specifies a single blank).
- " " can be used to enclose any ASCII data.

Literal specifications can be used with format specifications for any print list item. You should not use commas to separate literal specifications from other specifications for a given print list item. This is because output for the PRINT USING statement is terminated by the end of the print list or a separator in the format specification.

Examples:

```
10 A=5
20 PRINT USING 40;A
30 PRINT USING 50;A
40 IMAGE "LIST OF",DD," ITEMS"
50 IMAGE "LIST OF"DD"ITEMS"

run RETURN

LIST OF 5
LIST OF 5 ITEMS
```

Formatting Strings

Strings can be specified in two ways:

" " A literal specification is a string enclosed in quotes. Literals may be included in any specification.

A The character A is used to specify a single string character. nA specifies n characters.

Examples:

Each of these sequences causes the same output:

```
10 Res$= "RESTART"
20 IMAGE "***"4X7A4X"***"
30 PRINT USING 20; Res$

40 PRINT USING "3A4X7A4X3A";"***RESTART***"

RUN RETURN
***     RESTART     ***
***     RESTART     ***
```

If the string item in the print list is longer than the number of characters specified, the string is truncated. For example, the statement

```
50  PRINT USING "4A";"RESTART"
```

will only print the first four characters of "RESTART":

```
          REST
```

If the item is shorter, the rest of the field is filled with blanks on the right.

Formatting Numbers

Numeric specifications can be made up of digit symbols, sign symbols, radix symbols, separator symbols and an exponent symbol.

Digit Symbols

D specifies a digit position. nD specifies n digit positions. Leading zeros are replaced with blank spaces as fill characters.

Example:

```
10 PRINT USING "DDDDD,2X,DD";250,45

RUN RETURN

     250   45
```

Digit Separators

Digit separators are used to break large numbers into groups of digits (generally three digits per group) for easier reading. In the United States, the comma is customarily used; in Europe, the period is commonly used. Note that C and P cannot both be used in the specification for the same print list item.

C   Specifies a comma as a separator in the specified position.

P   Specifies a period as a separator in the specified position.

The digit separator symbol is output only if a digit in that item has already been output; the separator must appear between two digits.

Examples:

```
10 N=12345.67
20 PRINT USING "DDDDD.DD";N
30 PRINT USING "DDCDDD.DD";N
40 PRINT USING "2DC3D.2D";N
50 PACK A$ USING "2DP3D";N

RUN RETURN

12345.67
12,345.67
12,345.67
12.346
```

Radix Symbols

A radix symbol separates the integer part of a number from the fractional part. In the United States, this is customarily the decimal point, as in 34.7. In Europe, this is frequently the comma, as in 34,7. No more than one radix symbol can appear in a numeric specification.

.   Specifies a decimal point in that position.

R   Specifies a comma in that position.

Examples:

```
10 PRINT USING
   "DDD.DD,2X,DDD.DDD,2X,DDDRDD";123.4,56.789,98,7
20 IMAGE DDD.DDD,4X,DDD.DD
30 PRINT USING 20;.111,22.33
```

RUN **RETURN**

```
123.40   56.789   98,00   7.00
  0.111     22.33
```

Sign Symbols

Two sign symbols control the output of the sign characters + and −. Only one sign symbol can appear in each numeric specification.

S   Specifies output of a + sign if the number is positive, − if the number is negative.

M   Specifies output of a − sign if it is negative, a blank if it is positive.

If the sign symbol appears before the first significant digit of output it floats to the left of the leftmost significant digit.

When no sign symbol is specified and the only value to be output is negative, one of the digit positions will be used for the sign.

Example:

```
10 PRINT USING
   "MDD.DD,2X,DDSD.DD,DDDD";-34.5,-67,-10
```

```
-34.50   6-7.00  -10
```

Floating Specifiers

The sign specifications S and M in a numeric specification will "float" past blanks to the left most digit of a number or to the radix indicator. Sign symbols that are imbedded between significant digits do not float. The following examples show floating and non-floating specifications for two input values: -17 and $+17$.

Specification Output for -17 Output for $+17$

M4D	-17	17
S4D	-17	$+17$
DMDD	-17	17
DDMD	$1-7$	$1\ 7$
DDDDS	$17-$	$17+$

Format Replication

Many of the symbols used to make up format specifications can be replicated (repeated) by placing an integer (from 1 thru 255) in front of the symbol. For instance, the following IMAGE statements specify the same format:

```
30 IMAGE DDDDDD.DD
40 IMAGE 2DD3D.2D
50 IMAGE 6D.2D
```

Placing an integer before a symbol works exactly like having multiple adjacent characters. The X, D, A, and / symbols can be replicated directly.

Example:

```
20 PRINT USING "5DX";1,2,3,4,5

RUN  [RETURN]

    1     2     3     4     5
```

In addition to symbol replication, an entire specification or group of specifications can be replicated by enclosing it in parentheses and placing an integer before the parentheses.

Examples:

```
10 IMAGE 3(K,)

10 IMAGE DD.D,6(DDD.DD,)D3(CDDD).DD

10 IMAGE D.D,2(DDD.DD,)3(D.DDD,)

10 IMAGE D,4(4X,DD.DD,"LABEL2",2X,DD)
```

The symbols K and @ can also be repeated:

```
10 IMAGE 4(K,)2(@)
```

Note the use of the "," character in the above examples. If a format is repeated for several items in a print list, a comma must be included in the parentheses to indicate that additional print list items are to follow. For example, if there are three print list items, a simple numeric format might be 3(DD,). This is equivalent to DD,DD,DD,. If the comma is not included the result would be DDDDDD and could be used only with the first print list item.

Examples:

```
10 PRINT USING 20;"ABCDEFGH","abcdefgh","xxxx","YYYY"
20 IMAGE 3(AA)
run  RETURN
ABCDEFabcdefxxxx  YYYY

>
10 PRINT USING 20;"ABCDEFGH","abcdefgh","xxxx","YYYY"
20 IMAGE 3(AA,)
run  RETURN
ABabxxYY

>
10 PRINT USING 20;"ABCDEFGH","abcdefgh","xxxx","YYYY"
15 PRINT "END"
20 IMAGE #3(AAAAAA,)
run  RETURN
ABCDEFabcdefxxxx  YYYY  END
```

Compacted Formatting

A single symbol, K, is used to define an entire specification for either numeric or string output. If the corresponding print item is a string, the entire string is output. If it is a number, it is output in standard form. K outputs no leading or trailing blanks in numeric fields.

Example:

```
90 IMAGE K,2X,K,K,K
100 PRINT USING 90;"ABC",123,"DEF",456

RUN  RETURN

ABC   123DEF456
```

Carriage Control

The carriage return and linefeed normally output at the end of the list can be altered by using a carriage control symbol as the first item in a format string.

\# Suppresses both the carriage return and linefeed.

\+ Suppresses the linefeed.

\- Suppresses the carriage return.

Example:

```
30 IMAGE #,4(A2X,)
40 IMAGE K
50 PRINT USING 30;"A","B","C","D"
60 PRINT USING 40;"***"

RUN RETURN

A  B  C  D  ***
```

Notice that PRINT USING "+" is equivalent to PRINT LIN(0); and PRINT USING "−" is equivalent to PRINT LIN(−1).

Reusing the Format String

A format string is reused from the beginning if it is exhausted before the print list. The carriage return and linefeed are not normally output until the print list is exhausted.

Example:

```
70 PRINT USING "DDD.DD";25.11,99,9

RUN RETURN

25.11 99.00  9.00
```

Field Overflow

If a numeric item requires more digits than the field specification provides, an overflow condition occurs. When this happens, the item which causes the overflow is replaced with a field of $s. Then the rest of the print list is output.

Example:

```
105 IMAGE 3(DD.D,)
110 PRINT USING 105;111,2,33.3
120 PRINT USING 105;12.3,123.4,1234.56
130 PRINT USING 105;12.3,-1.2,-12.3

RUN RETURN

$$$$ 2.033.3
12.3$$$$$$$$
12.3-1.2$$$$
```

Programming Considerations

One factor that must be taken into account when creating formatted output with PRINT USING is the number of columns in the output device. When printing numeric output, you should specify a format that will not cause a line of output characters to exceed the number of characters per line of the output device (display or printer).

This section describes the statements available to organize and use data in arrays. The statements presented are those used to reserve storage space for the array data:

- DIM
- INTEGER
- LONG
- SHORT

Techniques used to enter and print array data are also described.

The discussion is initially limited to one and two-dimensional arrays. However, many of the statements can also be used with arrays of up to 32 dimensions. The general case of multi-dimensional arrays is covered in the final pages of this section.

Dimensioning Arrays

DIM Statement

The DIM statement is used to reserve storage space for the data in an array of SHORT or string data. (Type SHORT is the normal default data type. This can be changed using the SET command.) In order to use arrays of LONG or INTEGER data, use a type declaration statement (refer to the description of the DIM statement). A DIM statement specifies the array name and the size (number of elements) of the array. Size is indicated by specifying an upper bound for each dimension. For example, a one-dimensional array, Weight, that contains 25 LONG values is dimensioned:

```
            array
            name
             /
     LONG Weight(24)
```

Individual elements are then referenced with subscripted variables as Weight(0), Weight(1), etc., through Weight(24).

The statement

```
     10 LONG Z(25)
```

dimensions a LONG one-dimensional array of 26 elements: Z(0), Z(1), Z(2), etc., through Z(25).

The statement

```
                 maximum
                  length
                    /
     10 DIM Job$(50)[20]
```

dimensions a one-dimensional array of 51 string values. In addition to the keyword, the array name, and the upper bound, you may also specify the maximum length (maximum number of

characters per string). You must specify a maximum length for all string arrays that will contain strings longer than 18 characters.

It is recommended that you specify the maximum length even when all strings will be shorter than 18 characters. This makes the program easier for yourself and others to read.

upper bounds

```
10 DIM Inv(5,3)
```

dimensions a two-dimensional array of SHORT values with 6 rows and 4 columns. Individual elements are then referenced as:

Inv(0,0)	Inv(0,1)	Inv(0,2)	Inv(0,3)
Inv(1,0)	Inv(1,1)	Inv(1,2)	Inv(1,3)
Inv(2,0)	Inv(2,1)	Inv(2,2)	Inv(2,3)
Inv(3,0)	Inv(3,1)	Inv(3,2)	Inv(3,3)
Inv(4,0)	Inv(4,1)	Inv(4,2)	Inv(4,3)
Inv(5,0)	Inv(5,1)	Inv(5,2)	Inv(5,3)

For example, Inv(1,3) is a subscripted variable that refers to the value in the first row and third column of Inv. Note that Inv(0,3) refers to the value in the zero row and third column.

The statement

```
20 DIM Prod$(2,39)[25]
```

dimensions a two-dimensional array of string values with 3 rows and 40 columns. Each string in the array has a maximum length of 25 characters.

Type Declaration Statements

Type declaration statements are used to dimension arrays that contain numeric data. An array of SHORT values can also be dimensioned with a DIM statement. (Numeric data types are explained in Section 2.) A type declaration statement for numeric arrays consists of a numeric type, an array name, and the size of the array. Multiple arrays can be included in a single type declaration statement. The numeric type applies to all arrays named in the statement.

Examples:

```
           numeric    array
             type     names
              |       /
        10 INTEGER A(10),B(4,4)
```

10 INTEGER A(10),B(4,4)   Declares a one-dimensional array A with 11 INTEGERS, and a two-dimensional size array B with 25 INTEGERS.

10 SHORT B1(20)   Declares a one-dimensional array B1 with 21 SHORT values. The statement 20 DIM B1(20) has the identical effect.

10 LONG C(5),D(39)   Declares a one-dimensional array C with six LONG values, a one-dimensional array D with 40 LONG values.

10 DIM A(15),B(4)   The array A contains 16 SHORT values and the array B contains 5 SHORT values.

10 DIM C$(5,3)[20]   The two-dimensional array C$ has 6 rows and 4 columns of string values.

10 DIM G(19,4)   The two-dimensional array G contains 20 rows and 5 columns of SHORT values.

An array which appears in a type declaration statement may not appear in a DIM statement. If it does, the program will halt and an error message will be displayed. You cannot redimension arrays within a program. Attempting to execute the same dimension statement again or to execute another dimension statement with the same variables and parameters will cause a REDECLARED VARIABLE error.

It is possible (but not recommended) to use an array in a program without first using a DIM statement or a type declaration statement to set aside storage space for the data. In this case, the number of dimensions is assumed from the first program statement executed which references the array. If the dimensions of a one-dimensional array are not specified, the array is assumed to have an upper bound of 10 (i.e., 11 elements). If the dimensions of a two-dimensional array are not specified, both dimensions are assumed to have an upperbound of 10 (i.e., 11 rows and 11 columns). This provides a total of 121 elements.

If the actual array is larger, the program will halt and the error message "SUBSCRIPT OUT OF RANGE IN LINE xxx" will be displayed.

The DIM statement and the type declaration statements specify a maximum number of elements for each array.

You may not exceed or change this maximum within the program. If you attempt to do so, an error occurs and program execution stops.

Using Arrays

Entering Data into an Array

Data may be entered into an array element by element as shown in the examples below.

Example:

(One-dimensional array)

```
10 DIM A(4)
20 FOR I=0 TO 4
30 INPUT A(I)
35 PRINT "A(";I;")=";A(I)
40 NEXT I

>RUN RETURN

? 1 RETURN
A( 0 )= 1

? 22 RETURN
A( 1 )= 22

? 36 RETURN
A( 2 )= 36

? 48 RETURN
A( 3 )= 48

? 64 RETURN
A( 4 )= 64
>
```

You need not reference the 0 row or column in an array. All elements in arrays are initialized to 0 when the program is run. String array elements are initialized to null strings.

Example:

(Two-dimensional array)

```
10 FOR I=1 TO 3
20 FOR J=1 TO 5
30 INPUT M(I,J)
40 NEXT J
50 NEXT I
 .
 .
 .

RUN [RETURN]
```

```
?2.5 [RETURN]          M = 2.5 1.7 4.2 0.2 1.0
?1.7 [RETURN]              4.9 32. 4.7 5.6 7.2
?4.2 [RETURN]              8.9 9.8 1.1 2.2 3.3
 .
 .
 .
etc
```

Sample Program

```
10 DIM Price(5,3)
20 DATA 1,0,2.95,2,0,4.95,3,6,5.50,4,2,12.00,5,1,2.00
30 LET Sum=0
40 FOR I=1 to 5
50 READ Price(I,1),Price(I,2),Price(I,3)
60 LET Sum=Sum+Price(I,3)
70 NEXT I
80 PRINT ''Sum='';Sum

>RUN [RETURN]
Sum= 27.4
```

Printing Array Data

```
100 PRINT A(1), LIN(2)
101 PRINT A(2), LIN(2)
102 PRINT A(3), LIN(2)
103 PRINT
104 PRINT B(1,1),B(1,2),LIN(2),B(2,1),B(2,2),LIN(2)
105 PRINT
```

Examples:

```
10 PRINT A(1),LIN(2),A(2),LIN(2),A(3),LIN(2),LIN(2),SPA(10),
11 PRINT B(1,1),B(1,2),LIN(2)
20 PRINT B(2,1),B(2,2),LIN(2),LIN(1)

200 PRINT A(1),LIN(2),
201 PRINT A(2),LIN(2),
202 PRINT A(3),LIN(2),
204 PRINT B(1,1);B(1,2);LIN(2);B(2,1);B(2,2);LIN(2),LIN(1)
```

Substrings as Array Elements

It is possible to reference strings or substrings as array elements. (Refer to Section 7 for a discussion of strings and substrings.) The substring designator immediately follows the array element name and is enclosed in parentheses.

Example

```
Assume

10 DIM A$(3)[8],B$(2,2)[7]
```

and

```
20 A$(1)="ABC"
30 A$(2)="DEF"
40 A$(3)="GHI"
50 B$(1,1)="123"
60 B$(1,2)="456"
70 B$(2,1)="789"
80 B$(2,2)=""
```

then

```
90 PRINT A$(1)[2,2]
100 PRINT A$(1)[2;1]
110 PRINT B$(2,1)[2,2]
120 PRINT B$(2,1)[2;1]

>RUN RETURN

B
B
8
8
>
```

Example: It is often convenient to store character data in an array in order to reference groups of items.

Assume that you must store the names, addresses, and zip codes for 10 companies. Each of the items can be up to 20 characters long. They can be stored in a 2-dimensional string array. The declaration would be as follows:

```
10 DIM A$(10,3)[20]
```

You can now address any of the elements for any of the companies directly. The row number gives the data entries for a specific company while the column number selects the name, address, or zip code.

```
Data = "AJAX1","100 River Drive","99990"
       "AJAX2","110 River Drive","99991"
                      .
                      .
       "AJAX10","190 River Drive","99999"
```

The address of the 7th company would be accessed as A$(7,2).

Array Functions

There are no built-in matrix or array functions in BASIC/47. Most array functions can be easily provided with subroutines or subprograms. Examples are given for the following functions:

- Zero
- Constant
- Identity
- Transpose

Setting an Array to Zero or Other Constant

One of the most common array functions is the initializing of arrays or array elements. All array elements are set to zero when the array is declared. This eliminates the need to assign the value of zero to each array element at the beginning of a program. If the array must be set to zero at some point later in the program or if the initial value is other than zero, you can use the following technique:

Example: Set the array Balance(5,10) to a value of 10.

```
10 FOR I = 1 TO 5
20 FOR J = 1 TO 10
30 Balance(I,J) = 10
40 NEXT J
50 NEXT I
```

Setting an Array to an Identity Array

An array can be set to the identity array using the following technique:

Example: Set the array Ident(8,8) to an identity. Array elements on the major diagonal (I=J) having a value of 1, all others having a value of zero.

```
10 FOR I = 1 TO 8
20 FOR J = 1 TO 8
30 Ident(I,J) = I AND J
40 NEXT J
50 NEXT I
```

Transposing Arrays

You can transpose an array using the following technique:

Example: Assign the array Trans(3,4) to the transposition values of array Normal(4,3).

```
10 FOR I = 1 TO 4
20 FOR J = 1 TO 3
30 Trans(J,I) = Normal(I,J)
40 NEXT J
50 NEXT I
```

Strings are groups of ASCII characters. They allow you to manipulate textual data and to provide printed output. In addition, strings allow you to perform interactive dialog with the program user. Allowing you to use strings and string functions makes it relatively easy to develop application programs to control text. (Refer to Section 2 for a discussion of string data.)

BASIC provides a variety of special string functions. These functions are described in Section 12. Most string functions are used to act on only a portion of a string at a time. For example, the text string that makes up a sentence might be scanned a few characters at a time to detect words. The remainder of this section describes how to access the data contained in strings.

Substrings

A substring is a portion of a string rather than the entire string. Normally, a reference to a string variable refers to the entire string. For instance, if A$ = "ABC123" then after the statement

```
50 LET B$ = A$
```

```
B$ = "ABC123".
```

However, sometimes it is necessary to reference only a portion of a string. Suppose B$ is to be set equal only to the last three characters of A$. This can be done using the substring designator. Again assume that A$ = "ABC123". The statement

```
50 LET B$ = A$[4,6]
```

sets B$ = "123", not "ABC123".

There are three ways to designate a substring. The first method is to indicate the starting index (position) of the substring, followed by a comma (","), followed by the ending index of the substring.

Examples:

```
If A$ = "ABC123+-*/", then

starting   ending
 index      index
      \    /
    A$[1,3] = "ABC"
    A$[4,6] = "123"
    A$[7,10] = "+-*/"
    A$[1,10] = "ABC123+-*/"
```

Notice that A$[1,10] is effectively the same as just plain A$. The second method of designating a substring is to give the position of the first character, followed by a semicolon (";"), followed by an expression indicating the length of the substring. Continuing the above example:

```
  starting
   index      length
       \      /
      A$[1;3] = "ABC"
      A$[4;3] = "123"
      A$[7;4] = "+-*/"
      A$[1;10] = "ABC123+-*/"
```

Notice that A$[1;10] is effectively the same as just plain A$.

The third method of designating a substring is to give the starting index of the substring only. This is really a special case of method 1, in which the ending index is assumed to be the length of the string. Continuing the example above:

```
  starting
   index
       \
      A$[1] = "ABC123+-*/"
      A$[5] = "23+-*/"
      A$[10] = "/"
```

Notice that A$[1] is effectively the same as just plain A$.

Regardless of which method is used, the first position indicator, the second position indicator, and the length indicator may be any numeric expression with a value of 1 to 255. Any expression used as a substring designator which has a SHORT or LONG value is rounded to the nearest integer.

Examples:

```
      A$[N,M]
      A$[F;L]
      A$[N + SQR(Z);L - 1]
      A$[A(I),A(J)]
      A$[C^D^E;C^D^E + Q]
```

In the special case of the null string, a null substring is always extracted as long as the substring designator is valid.

Substrings are frequently used to insert or change characters in a string without affecting the rest of that string.

If the item being assigned is too long for the substring, the item is truncated from the right until it fits.

If the starting index is greater than the length of the string, the null string is assumed.

If the specified starting index is less than 1, the ending index is less than 1, or the starting index is greater than the ending index, an error message will be displayed and the program will be terminated.

Example:

```
10 DIM Heading$[25]
```
The string Heading$ has maximum length of 25 characters.

```
20 LET Heading$ = "School of "
```
Initially, Heading$ is the 10 character string "School of"

```
30 LET B$ = "Business"
40 LET Ed$ = "Education"
50 LET En$ = "Engineering"
60 LET L$ = "Law"
70 LET M$ = "Medicine"
   .
   .
   .
   .
1050 LET Heading$[11] = B$
```
The string B$ is inserted into the string Heading$ beginning at the 11th character of Heading$.

```
1055 PRINT Heading$
   .
   .
   .
   .
```
Prints: School of Business

```
3000 LET Heading$[11] = En$
```
The string En$ is inserted into the string Heading$ beginning at the 11th character of Heading$.

```
3010 PRINT Heading$
```
Prints: School of Engineering

```
5500 LET Heading$[11] = L$
```
The string L$ is inserted into the string Heading$ beginning at the 11th character of Heading$.

```
5510 PRINT Heading$
```
Prints: School of Law

Subprograms_____8

Programs developed for business or engineering applications can easily contain hundreds of statements. A large program is easier to develop, debug, and document if it is divided into several program units. Each of the program units can then perform a single task. In BASIC, the term program unit can refer to a main program or a subprogram.

Study the diagram in figure 8-1. It illustrates how a large program might be broken up and re-organized using subprograms. A main program links the program units together.

```
Main Program          1st Subprogram       2nd Subprogram

10  REM Main          200 SUB Pro1         300 SUB Pro2
20                     .                    .
30                     .                    .
 .                     .                    .
 .                     .                    .
 .                    260 CALL Pro2         .
60  CALL Pro1         270                   .
70                     .                    .
 .                    290 SUBEND           390 SUBEND
 .
 .
 .                     3rd Subprogram
 .
 .
100 CALL Pro3         400 SUB Pro3
110                    .
 .                     .
 .                    450 SUBEND
```
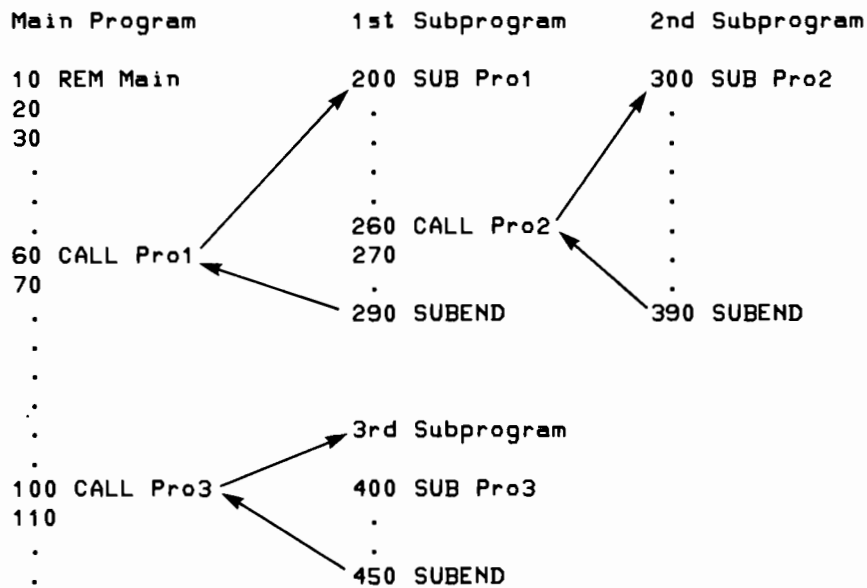
Figure 8-1. Subprogram Example

The arrows indicate where control is transferred from one program unit to another. For instance, at main program statement 60, control passes to the first subprogram. When the first subprogram is finished processing, the main program resumes at statement 70. Notice how, before the first subprogram ends, control passes to the second subprogram, and from there back to the first subprogram.

The main program can call subprograms. Subprograms can call other subprograms and they can call themselves.

Listed following is a BASIC program that calculates a person's weekly salary including overtime pay (statements 10 to 110), and then calls on a subprogram, Dduct, first (at statement 190) to calculate a tax deduction, then (at statement 210) to calculate a deduction for an automatic savings deposit, and a third time (at statement 230) to calculate a deduction for an insurance payment.

SUB and SUBEND Statements

A subprogram always begins with a SUB statement and ends with a SUBEND statement. The occurrence of the first SUB statement terminates the main program. In the example in figure 8-2 the SUB statement is:

```
             subprogram
                name
                                 formal parameters
                    ┌──────────────────┴──────────────────┐
300 SUB Dduct (Category$,Salary,Percent,Takehome,Tded)
```

A subprogram name is composed of an upper case letter of the alphabet, A through Z, which may be followed by any combination of digits, lower case letters, or the underscore symbol "_".

The SUB statement must be the first statement in a line and the SUBEND statement must be the only entry in a line.

A formal parameter can be a simple string or numeric variable, or an entire numeric or string array or a logical file number. When the subprogram is executed, values are passed to the formal parameters from the actual parameters in the CALL statement. For instance, in figure 8-2, the first CALL statement is:

```
             subprogram
                name
                             actual parameters
                    ┌─────────────┴─────────────┐
190 CALL Dduct (D$,Salary,Percent,Ttal,Tded)
```

At this point in the program, the values of the actual parameters are:

```
D$        = "Tax"
Salary    = 139.65
Percent   = 10
Ttal      = 139.65
Tded      = 13.97
```

The values of the parameters in the CALL statement are passed to the parameters in the SUB statement according to their position in the parameter list; that is, the value of the first actual parameter is passed to the first formal parameter; the value of the second actual parameter is passed to the second formal parameter; and so on. Thus, when the subprogram Dduct is called by statement 190, the values assigned to the formal parameters are:

```
Category$ = "Tax"
Salary    = 139.65
Percent   = 10
Takehome  = 139.65
Tded      = 13.97
```

```
10 PRINT SPA(12);"Weekly Pay Calculator";LIN(1)
20 READ Hours,Wage
30 DATA 40,2.85,6
40 Pay=Hours*Wage
45 IMAGE 4X,3D.2D,16A,D.2D,4A,3D.2D
50 PRINT USING 45;Hours;" standard hrs @";Wage;" = $";Pay
60 REM ** This section computes overtime pay **
80 READ Ovtime
90 Ovpay=Ovtime*Wage*1.5
100 Ttal=Pay+Ovpay
110 PRINT USING 45;Ovtime;" overtime hrs @ ",Wage*1.5," = $";Ovpay
112 PRINT SPA(34);"_____"
115 PRINT USING "12X,18A,4A,3D.2D2/";"Total pay";" = $";Ttal
120 REM ** This section CALLs Dduct to compute deductions **
130 Salary=Ttal
140 D$="Tax"
150 Temp=10
160 IF Ttal>160 THEN Temp=15
170 IF Ttal>195 THEN Temp=20
180 IF Ttal=240 THEN Temp=25
190 CALL Dduct(D$,Salary,Temp,Ttal,Tded)
200 D$="Savings"
210 CALL Dduct(D$,Salary,Temp/2,Ttal,Tded)
220 D$="Insurance"
230 CALL Dduct(D$,Salary,3,Ttal,Tded)
235 PRINT SPA(34);"_____"
240 PRINT USING 260;"Total deductions   = $";Tded
250 PRINT USING 260;"NET (takehome) PAY = $";Ttal
260 IMAGE /12X,22A,3D.2D/
270 END
300 SUB Dduct(Category$,Salary,Percent,Takehome,Tded)
310 REM ** This subprogram makes payroll deductions **
320 Temp=Salary*Percent*.01
330 PRINT USING "10X,21A,3A,3D.2D";Category$&" deduction ";"= $";Temp
340 Takehome=Takehome-Temp
350 Tded=Tded+Temp
360 SUBEND
```

RUN [RETURN]

```
              Weekly Pay Calculator

    40.00 standard hrs @ 2.85 = $114.00
     6.00 overtime hrs @ 4.28 = $ 25.65

                              ------

    Total pay             = $139.65



        Tax deduction       = $ 13.97
        Savings deduction   = $  6.98
        Insurance deduction = $  4.19

                              ------

        Total deductions    = $ 25.14

        NET (takehome) PAY = $114.51
```

Figure 8-2. Payroll Deduction Subprogram

Note that the variable names of the corresponding parameters are not the same. The values are assigned according to their position in the parameter list. An actual parameter can be a numeric or string variable, numeric or string expression, an array, or a logical file number.

The null subscript notation is used to specify an array in subprogram formal and actual parameter lists. There are two forms of the null subscript notation as illustrated below. When commas are used to indicate the number of dimensions in the array, BASIC checks to insure that the number of subscripts in the actual and formal parameters match.

Examples:

Salary( )   Specifies a one-dimensional numeric array named Salary.

Salary(,)   Specifies a two-dimensional numeric array named Salary. The number of
                  dimensions is indicated by the number of commas. n commas indicate an
                  n + 1 dimensional array.

Names$(, ,)   Specifies a 3-dimensional string array named Names$. The formal and
                  actual parameters must correspond in type (i.e. string or numeric, single
                  value or array or logical file number) and number (i.e. if there are 3 actual
                  parameters in the CALL statement, there must be exactly 3 formal
                  parameters in the corresponding SUB statement).

Examples:

A. 10 CALL Route(D$,L$,M)   Valid.
   .
   .
   99 SUB Route(Driver$,License$,
       Mileage)

B. 10 CALL Order(X + 1, N * Y,Z)   Valid.
   .
   .
   99 SUB Order(Number,Quantity,
       Price)

C. 10 CALL Reg(Number,Name$)   INVALID. The number of actual parameters
                                   must equal the number of formal
                                   parameters.

   50 SUB Reg(Number,Name$,Date)

D. 10 CALL Report(N$,D$,A12$)   INVALID. Third actual parameter in the
                                   SUB statement is a string but third formal
                                   parameter is a numeric variable. Code
                                   should be Code$.

   50 SUB Report(Q$,R$,Code)

Numeric variables in parameter lists are assumed to be type SHORT unless specified otherwise.

Examples:

100 SUB Sample (A(,),B)   The subprogram Sample has two SHORT parameters: the array A, and B.

200 SUB Sample_2(S(,),INTEGER T)   The subprogram Sample—2 has the SHORT array parameter S(,) and the INTEGER parameter T.

300 SUB Sample_3 (LONG X(,),Y)   The subprogram Sample—3 has the LONG array parameter X(,) and the SHORT parameter Y.

Notice that when a numeric type is specified in a parameter list, the type applies only to the immediately following variable.

Pass-By-Reference/Pass-By-Value

Subprogram parameters can be passed-by-reference or passed-by-value. Passed-by-reference means that the actual and formal parameters have the same location in memory. Thus, when the value of the formal parameter is changed in a subprogram, the value of the actual parameter is also changed.

Example:

```
10 X = 0
20 Y = 99
```

30 CALL Formula(X,Y)   Actual parameters X and Y are passed-by-reference to formal parameters A and B. Any changes to A and B in the subprogram will also affect X and Y.

```
120 SUB Formula (A,B)   ! A = 0 ; B = 99
130 A = A + 3           ! A = 3 ; X = 3
140 B = B + 1           ! B = 100 ; Y = 100
150 SUBEND
 .
 .
 .
```

Actual parameters that are variables are always passed-by-reference unless the variable is enclosed in parentheses. Arrays must always be passed-by-reference. When numeric parameters are passed by reference, the formal parameter is changed to the same type as the actual (passed) parameter. If the formal parameter is declared to be of a specified type then the passed parameter must be of the same type.

When the actual parameter is a constant, an expression, or a variable in parentheses, it is passed-by-value. When a parameter is passed-by-value, the actual and formal parameters have different locations in memory. Initially the two locations have the same value, but if the value of one is changed, the value of the other is not affected.

Example:

```
10 M = 0
20 N = 21
30 CALL Calc(3,(M),N+4)    These actual parameters are all passed-by-value to the formal
                           parameters A1, A2, and A3. Any changes to A1, A2, and A3
                           in the subprogram have no effect on the actual parameters.

120 SUB Calc(A1,A2,A3)  ! A1 = 3;  A2 = 0;  A3 = 25
130 A1 = A1^A2          ! A1 = 3^0    = 1
140 A2 = A2 + 2         ! A2 = 0 + 2 = 2
150 A3 = A3^A2          ! A3 = 25^2  = 625
```

The examples in figure 8-3 illustrate the distinction between pass-by-reference and pass-by-value. The four examples are all the same except that the fourth parameter is passed-by-reference in example A and passed-by-value in examples B, C, and D.

A. The fourth actual parameter is a variable, hence when the value of the fourth formal parameter is changed by the subprogram (statement 340), the value of the actual parameter also changes.

B. The fourth actual parameter is a variable enclosed in parentheses, hence when the value of the fourth formal parameter is changed in the subprogram (statement 340), the value of the actual parameter is unaffected.

C. The fourth actual parameter is an expression, hence when the value of the fourth formal parameter is changed in the subprogram (340), the value of the actual parameter is unaffected.

D. The fourth actual parameter is a constant, hence when the value of the fourth formal parameter is changed in the subprogram (340), the value of the corresponding actual parameter is unaffected.

Local Variables

Variables within a subprogram that are not contained in the list of formal parameters in the SUB statement are local to that subprogram. This means that the variables in a program which calls a subprogram are distinct from variables with the same name within the called subprogram.

Examples:

A.
```
10 INTEGER A
20 LET A = 100
   .      .
   .      .
   .      .
70 CALL Calc (X,Y,Z)
```
Values of X,Y, and Z are passed (by reference) to the formal parameters Height, Weight, and Age. Subprogram Calc begins.

```
80 PRINT A
```
Prints the value of the INTEGER A (100, not 8.95).

```
400 SUB Calc(Height,Weight,Age)
```
Height, Weight, and Age are assigned the values of X,Y, and Z.

```
410 LONG A
```
Declares a LONG variable A that is distinct from the INTEGER A in the calling program unit.

```
490 LET A = 8.950496
```
Assigns the value 8.950496 to the local LONG variable A. This does not affect the value of the INTEGER A in the calling program unit.

```
500 PRINT A
```
Prints 8.950496

```
510 SUBEND
```
Returns to the calling program unit at statement 80.

B.
```
10 LONG B
20 LET B = 47
   .     .
   .     .
   .     .
70 CALL Result (B)
```
The value of B is passed (by reference) to the variable X in statement 300. Any changes to X in the subprogram will also change the value of B in this calling program unit.

```
80 PRINT B
   .     .
   .     .
   .     .
300 SUB Result (X)
```
Prints 49.

X is assigned the value of B.

```
320 LET X = X + 2
```
Changes the value of the formal parameter X. Also changes the value of B in the calling program unit (passed-by-reference).

```
340 PRINT X
```
Prints 49.

```
350 SUBEND
```
Returns to calling program unit at statement 70.

```
            fourth actual parameter (variable)
                                              \
    190 CALL Dduct(D$,Salary,Percent,Ttal,Tded)
      .
      .     fourth formal parameter
      .
    300 SUB Dduct(Category$,Salary,Percent,Takehome,Tded)
    340 LET Takehome = Takehome - ( Salary * Percent * .01 )
    350 SUBEND
```

A. Variable Parameter

```
        variable in parentheses
                                \
    190 CALL Dduct(D$,Salary,Percent,(Ttal),Tded)
      .
      .
    300 SUB Dduct(Category$,Salary,Percent,Takehome,Tded)
    340 LET Takehome = Takehome - ( Salary * Percent * .01 )
    350 SUBEND
```

B. Parameter Enclosed in Parentheses

```
    (expression)
                \
    190 CALL Dduct(D$,Salary,Percent,4*Ttal,Tded)
      .
      .
    300 SUB Dduct(Category$,Salary,Percent,Takehome,Tded)
    340 LET Takehome = Takehome - ( Salary * Percent * .01 )
    350 SUBEND
```

C. Parameter is an Expression

```
    (constant)
              \
    190 CALL Dduct(D$,Salary,Percent,240,Tded)
      .
      .
    300 SUB Dduct(Category$,Salary,Percent,Takehome,Tded)
    340 LET Takehome = Takehome - ( Salary * Percent * .01 )
    350 SUBEND
```

D. Parameter is a Constant

Figure 8-3. Parameter Passing

All DATA, IMAGE, GOTO, ON END, ON ERROR, ON KEY, OFF KEY, GOSUB, ON...GOTO, ON...GOSUB, RESTORE, and RESUME statements in a program unit are local to that program unit. They can only reference line numbers and interrupt keys for that program unit.

Referencing Files

A file in a calling program may be referenced in a subprogram by passing its file number as a parameter.

Example:

```
20 ASSIGN "AFILE" TO #2
```
AFILE is opened as #2.

```
80 CALL Sprog1(N,N$,#2)
```
When the subprogram is called, #2 is passed as the third parameter.

```
200 SUB Sprog1(Number,Name$,#10)
   .
   .
250 READ #10;A$
```
This statement reads a value for A$ from the file "AFILE" which is file #2 in the main program and file #10 in the subprogram Sprog1.

In the CALL statement, the file number parameter must be a number or a numeric expression. The numeric expression is evaluated and rounded to the nearest integer at the time of the call. In the SUB statement, the file number must be an INTEGER value. File numbers that are not passed as parameters are local to the subprogram in which they appear. Any ON END # statement is local to the subprogram.

Example:

```
20 X = 1
30 CALL Closef(#X)
   .               .
   .
200 SUB Closef(#3)
210 ASSIGN * TO #3
220 SUBEND
   .
   .
```

When a file is closed in a subprogram with the ASSIGN * statement, all references to the originally passed file are deleted and any subsequent assignment to the formal file parameter in the subprogram will be local to that subprogram. If you want to reassign a file in a subprogram, use the ASSIGN filename statement to close the passed file. This will establish a new reference to the file which can be used locally as well as in other program units.

Files_____9

Files are used to store collections of data that 1) are too large to be contained conveniently in DATA statements in your program; or 2) must be stored independently because the data is going to be used by more than one program.

The use of files involves the following operations which are explained in detail in this section:

- Open files. — This gives your program access to the file and assigns a file number to that file. The file number is then used throughout the rest of your program to refer to that particular file.

- Write to files. — You can write on the file (put data into the file from your program).

- Read from files. — Read data on the file and copy data from the file to variables in your program.

- Close files. — Closing a file indicates that your program is finished using the file. The data remains on the file for future use.

ASSIGN Statement

Before a program can read from or write to a file, that file must be opened by using an ASSIGN statement. Opening a file causes that file to be associated with a file number. The file pointer is set to the current position of the selected file. The terminal file system maintains the position of the file pointer from this point on. Once a file is open, all references to it within a program are through its file number.

Examples:

                    filename   file number
                       /   /
10 ASSIGN "AFILE" TO #1
20 ASSIGN "PAYFILE" TO #2
30 ASSIGN "CFILE" TO #5.8   Associates CFILE with file #6 (the closest integer to 5.8).

40 ASSIGN "DFILE" TO #1   Closes AFILE and associates file #1 with DFILE.

50 ASSIGN "F4" TO #(5*R)   Evaluates the expression in parentheses to the nearest integer, then associates that value (file number) with F4.

60 ASSIGN "PAYFILE" TO #8   PAYFILE has already been associated with file #2 in statement 20. PAYFILE can now be referenced as either file #2 or file #8.

The filename can be a standard name that the terminal file system uses. Standard filenames are RTAPE, DISPLAY, HP-IB#4, etc. You can also use the Terminal ASSIGN command (COMMAND Channel) to establish special filenames by equating your filename to a standard terminal file name. For example, PRNTR can be equated to an HP-IB device.

`COMMAND`, ASSIGN [NAME] PRNTR TO HP-IB#5, `RETURN`

When an ASSIGN statement is executed, the program is given access to read from and write to the file. If the file cannot be opened, an error message is displayed and the program will halt. If another file is already associated with the file number specified in the ASSIGN statement, that file is closed and the specified one is opened.

Assigning a file to the display equates the file to the contents of the terminal display memory, not the terminal keyboard.

Serial PRINT # Statement

The serial PRINT # statement writes data items on a file starting at the current position of the pointer. The items may be numeric or string expressions.

Examples:

file number

print list

`100 PRINT #2;X,Y,Z`   Prints the values of the three numeric variables X,Y, and Z on file #2.

`300 PRINT #1`   A serial PRINT # statement with no print list will generate a record with only a carriage return and a line feed character.

The items in the print list are separated by commas or semicolons. Each item in the print list is written on the file in the order it appears on the print list. The items are written starting at the current position of the file pointer, overwriting whatever data may be in that position on the file. When a tape or device file is opened, the pointer is set to the current file position (not the beginning of the file). When a disc file is opened, the pointer is set to the beginning of the file.

Files opened by the tape emulator are treated as tapes.

Example:

`ASSIGN "LEFT TAPE" to #1`

Emulator files assigned as filenames bypass the emulator and are treated as disc files.

Example:

`ASSIGN "CTUL001" TO #1`

If you want to suppress the carriage return and linefeed at the end of a file record, terminate the file print statement with a comma or semicolon following the print list. A record will be sent to the file each time the end of the print list is reached and each time a linefeed character is generated. This allows a single print statement to generate multiple records if you include linefeed characters (CHR$(10)) as print list items.

`10 PRINT #2; A$,CHR$(10),B$,CHR$(10),C$`

READ # Statement

The READ # statement reads data items from a specified file into numeric or string variables.

Examples:

file   variable
number   list

100 READ #3;A,B,C   Reads three values from #3 into the numeric variables A,B, and C.

200 READ #2;N$,P   Reads from file #2, a string value for N$ and a numeric value for P.

The first item read is the item following the current position of the pointer; i.e., the item immediately following the last item accessed. Record boundaries are ignored. Numeric values can be assigned only to numeric variables, and string values to string variables. Otherwise an error message is generated. If a numeric value is not the same data type as the variable, conversion with rounding is performed. If an attempt is made to read beyond the end of data in the file, an error message is displayed and the program stops. Data items in the file must be separated by commas.

LINPUT # Statement

The LINPUT # statement reads the entire contents of the next available record from the file and assigns the contents to a specified string variable. If the record contains a carriage return and line feed at the end, these characters will not be removed and will be assigned to the string variable along with the other characters. If the record contains numeric values, the numbers are read as ASCII characters.

Example:

file number

100 LINPUT #3;A$ Reads the next available record from file #3 and assigns the contents to A$.

The string variable should be dimensioned large enough to accept the entire record. If the record is too long to fit in the string variable, it is truncated.

RESTORE # Statement

The RESTORE # statement repositions the file pointer to the beginning of the file specified. This statement is effective only for tapes and some HP-IB devices. It does not affect the display.

Example:

file number

30 RESTORE #4   Repositions the file pointer #4 to the beginning of file #4.

ON END # Statement

The ON END # statement sets a flag for a specified file so that if and when an end-of-file is encountered in reading or writing on that file, control is transferred to another statement in that program. There are three forms of the ON END # statement:

```
ON END...GOTO
ON END...GOSUB
ON END...CALL
```

Examples:

line number

10 ON END #2 GOTO 120   When an end-of-file is encountered in file #2, the program branches to line 120.

file number

line number

20 ON END #3 GOSUB 200   When an end-of-file is encountered in file #3, the program branches to the subroutine at line 200.

file number

30 ON END #4 CALL Sb1   When an end-of-file is encountered in file #4, the program calls subprogram SB1. Subprograms and the CALL statement are discussed in Section 8.

subprogram
name

ON END ... GOSUB and ON END ... CALL cause the specified subroutine or subprogram to be executed. When the subroutine or subprogram returns control, the file input statement that generated the ON END condition will be re-executed. The entire item list will be input (not just the items following the item that caused the end-of-file condition). This means that the subroutine or subprogram must do something to eliminate the end-of-file condition (e.g. rewind the tape or reassign the file) before returning. If this is not done you will simply loop until the program is manually aborted.

If no ON END # statement is used and an end-of-file is encountered, an error message is displayed and the program stops.

Closing a File

When a program ends, all files that were opened in that program are automatically closed.

There are two ways to close a file before the end of the program. You may use an ASSIGN statement in which the filename is replaced with an asterisk(*):

Examples:

```
10 .ASSIGN * TO #2       Closes file #2.
20 ASSIGN * TO #4       Closes file #4.
```

If the ASSIGN * statement is used in a subprogram, references to the file number are deleted in all program units. Additional information on subprograms is given in Section 8.

You can also close a file by assigning that file's file number to another file:

Examples:

```
10 ASSIGN "FILE1" TO #3    Opens FILE1.
20 ASSIGN "FILE2" TO #3    Closes FILE1 and opens FILE2.
```

This method of closing files affects only the current program unit. It does not close the file in other program units. If you attempt to close a file that is not open, nothing will happen.

File Error Variables

File errors can be acted on within your program by using the ON ERROR statement. You can test the error number to determine the type of the file error and take the appropriate action. A list of file errors is given in Appendix E.

Output to Printers and Terminals

You can list data to a lineprinter or a terminal by assigning a file number to the device with any ASSIGN statement.

Examples:

device name

```
50 ASSIGN "LP1" TO #4      Assigns file number 4 to the lineprinter named LP1.

60 ASSIGN "TE#4" TO #5     Assigns file number to the terminal named TE#4.

70 ASSIGN "T" TO #3        Assigns file number to the device named T.
```

Each lineprinter and terminal that is connected to your terminal has a device name which was assigned when the device was attached to the terminal. If you do not know a particular device name, use the terminal SHOW ASSIGNMENTS command. You can use the terminal command channel to equate any convenient name to a logical device. For additional information refer to the manual titled, *Getting Started with Your HP 2647F*.

Once a file number has been assigned, you can use that file number in a PRINT statement as shown in the following examples.

Examples:

```
80 PRINT #4;X,Y,Z         Prints the values of X,Y, and Z to file #4 which was
                          assigned to the lineprinter LP1.

90 PRINT #3;A$;N          Prints the values of A$ and N to file #3 which was
                          assigned to the device T.
```

The spacing rules that apply to commas, semicolons, and the TAB, SPA, and LIN functions in a simple PRINT statement (Section 4) also apply when printing to files.

Formatted Output to Devices

Output to a device file such as a line printer can be formatted with the PRINT # USING statement.

Examples:

```
120 PRINT #4 USING 121;A,B,C
121 IMAGE 3X,2D,3X,2D,3X,2D        Prints the values of A,B, and C to file #4 which was
                                   assigned to LP1. The output is formatted using the
                                   format specified in the IMAGE statement at line 121.
```

The rules for specifying the format are the same as those described in Section 5. The file number in a PRINT # USING statement must be assigned to a file name with the ASSIGN statement.

Cartridge Tape Emulator Files

This feature of the terminal allows the creation of sequential files (as found on cartridge tapes) on a flexible disc. The default Source and Destination assignments of the terminal (left tape and right tape) automatically create such files. The terminal will translate any commands directed to left tape to CTUL??? and to right tape to CTUR???. This operation allows the terminal to respond to tape commands contained in existing programs so that those programs can be run without modification.

The emulator can access any disc file through the assign command:

Press COMMAND , f4 ASSIGN, f4 more, f4 more, f2 CTUR, to PSEUDO RETURN .

This assignment directs the emulator to use file PSEUDO for all references to the right tape. The assignments may contain either volume or device specifications:

```
ASSIGN CTUL to STUFF;MYVOL
ASSIGN CTUR to PSEUDO:DISC#2
```

The assignments given so far direct the emulator to a single file only. For multiple file emulation, the assignment must contain a string of question marks to indicate where a file number should be placed. A single question mark (?) signals emulation of files 1 through 9, a double question mark (??) refers to the 99 files from 1 through 99, and a triple question mark (???) refers to files 1 through 255, the largest permissible file number. Attempts to read or write to file numbers beyond those assigned will trigger the error messages END OF DATA (read) or END OF TAPE (write).

The emulator allows writing to the left and right "tapes" at the same time. It is up to the user to insure that CTUL and CTUR refer to files on different disc drives. After a write, edit, or read-to-end-of-file operation, emulator files remain open until a REWIND, MARK FILE, or FIND FILE command is executed.

Terminal Operations__ 10

This section describes some of the special features available in BASIC/47 that allow your program to interact with the terminal's display, keyboard, and data communications capabilities. These features are not normally found in a BASIC interpreter. You should review these capabilities before writing your application program.

> **NOTE:** Operation of the terminal's flexible disc, cartridge tape, tape emulation, file, and graphics functions are described in the Files, and AGL sections of this manual.

The terminal functions are used in the same manner as other BASIC functions and statements. The operations use string and numeric arguments and may return string, numeric, or logical values or perform some terminal operation. Terminal operations are divided into the following categories:

- Display
- Keyboard
- Data Communications
- Program Control

Table 10-1 contains a list of terminal operations that can be accessed using BASIC/47. (Note that all terminal features can be accessed by using the appropriate escape in a PRINT

Table 10-1. Terminal Operations

OPERATION	DESCRIPTION	OPERATION	DESCRIPTION
Display		Data Communications	
CSENA(R,C)	Sense the absolute cursor position	GETDCM OFF	Disable the GETDCM function
CSENS(R,C)	Sense the screen relative cursor position	GETDCM ON	Enable the GETDCM function
MOVCA(R,C)	Move the cursor to an absolute position	GETDCM(S$)	Get datacomm input directly
MOVCR(R,C)	Move the cursor to a relative position	PUTDCM(S$)	Send datacomm output directly
MOVCS(R,C)	Move the cursor to a screen relative position	Program Control	
DSPIN$(L,X)	Input data directly from the display	COMMAND	Output terminal commands
Keyboard		ERRL	Line number of error
		ERRN	Error number
GETKBD OFF	Disable the GETKBD function	ERROR	Define program error condition
GETKBD ON	Enable the GETKBD function	FRE	Report program and variable storage
OFF KEY #	Disable the selected key	ON ERROR	Branch on error conditions
ON KEY #	Enable the selected key	RESUME	Continue program after error recovery
GETKBD(X)	Get keyboard input directly		
PROC$(X)	Process keycodes	SLEEP	Set BASIC to a dormant state
KEYCDE(X,N,C)	Redefine keys to new keycodes	WAKEUP	Restore BASIC to active state

10-1

statement). Refer to HP 2647F *Reference Manual* (part number 02647-90037) for detailed information concerning terminal escape sequences. The following paragraphs describe each of the terminal operation types.

Display Operations

The display operations allow you to sense the position of the alphanumeric cursor, position the alphanumeric cursor, and input data directly from the alphanumeric display memory.

Cursor Sensing

The cursor sensing functions return the position of the alphanumeric cursor and the current display workspace number. The alphanumeric memory is made up of rows of 80 characters. The display can contain from 1 to 255 rows of characters. The display window number can range from 1 to 4. The cursor positions can be requested in absolute or screen relative coordinates. The coordinates refer to the current display workspace. Additional information on cursor addressing is given in the terminal *Reference Manual,* Section 2.

Absolute Sensing

The CSENA(R,C) function returns the absolute cursor position. The function parameters must be numeric variables. The first variable specified is set to the row number (1 to 255) and the second to the column number (1 to 80). The value of the function is set to the number of the current display workspace (1 to 4).

Examples:

```
10 A=CSENA(P(I),Q(J))
20 Window=CSENA(R,C)
```

Screen Relative Sensing

The CSENS(R,C) function returns the screen relative cursor position. The function parameters must be numeric variables. The first variable specified is set to the row number and the second variable is set to the column number. The value of the function is set to the number of the current display workspace.

Examples:

```
10 B=CSENS(A(1),B(I))
20 W=CSENS(R,C)
```

Cursor Positioning

The alphanumeric cursor can be positioned in the current display workspace using absolute, screen relative, or cursor relative coordinates. The cursor positioning functions are used only in PRINT statements and do not return a value.

Absolute Cursor Positioning

The MOVCA(R,C) function can be used to position the alphanumeric cursor to any character position in the current display workspace. R defines the destination row and C defines the destination column. Both R and C must be numeric expressions and are rounded to integers before being used. The range of valid values for R will vary with the size of the workspace but will normally be between 1 and 255. Values for C should always be between 1 and 80.

Example: Position the cursor to row 5 and a column position given by C=4*Y.

```
10 PRINT MOVCA(5,4*Y);
```

Relative Cursor Positioning

The MOVCR(R,C) function can be used to position the alphanumeric cursor to any character position relative to the current cursor position. R defines the relative row position and C defines the relative cursor position. Both R and C must be numeric expressions and are rounded to integers before being used. The destination cursor position can be anywhere in the current display workspace. The values for R can range from −255 to +255. The values for C can range from −79 to +79.

Example: Position the cursor 3 rows down and 2*Z columns to the left.

```
10 PRINT MOVCR(3,-2*Z);
```

Screen Relative Cursor Positioning

The MOVCS(R,C) function can be used to position the alphanumeric cursor to any character position on the currently displayed screen of 24 lines. R gives the screen row position and C gives the screen column position. Both R and C must be numeric expressions. The values can be between 1 and 24 for R and 1 and 80 for C.

Example: Position the cursor to row 5 and column P^2.

```
10 PRINT MOVCS(5,P^2);
```

Direct Display Input

You can input string data directly from the terminal screen. You do not have to enter data from the keyboard in order for your program to access it. The DSPIN$(L,X) function allows you to read data stored anywhere in the current display workspace.

The DSPIN$(L,X) function reads string data beginning at the current alphanumeric cursor position. L is the number of characters to be read (length of the string). If L is positive, the display codes used by the terminal to indicate alternate character sets, display enhancements, and data fields are expanded into their equivalent escape sequences. A brief table of expanded codes is given in table 10-2. Detailed information on display memory functions is given in the terminal *Reference Manual* (part number 02647-90037). If L is negative, the display codes will be ignored.

Examples:
```
10 A$=DSPIN$(-10,X)
30 B$=DSPIN$(10,X)
```

Table 10-2. Terminal Display Codes

Escape Sequence	Description	Escape Sequence	Description
ᴱc & d @	End of enhancement	Nᶜ	Turn on alternate character set
ᴱc & d A	Blinking	Oᶜ	Turn off alternate character set
ᴱc & d B	Inverse video		
ᴱc & d D	Underline	ᴱc [	Begin unprotected field
ᴱc & d H	Halfbright	ᴱc]	End unprotected or transmit-only field
ᴱc) @	Define standard character set		
ᴱc) A	Define character set A	ᴱc {	Begin transmit-only field
ᴱc) B	Define character set B		
ᴱc) C	Define character set C	ᴱc 6	Begin alphabetic only field
		ᴱc 7	Begin numeric only field
		ᴱc 8	Begin alphanumeric field

Example: Run the sample program below. It will print a test string and then input the string using the DSPIN$ function. It will then input the string again, this time checking for display codes. Note the added display control characters in the second string.

```
10 DIM A$[36],B$[36]
20 PRINT MOVCS(20,1);
30 PRINT "This is a test for ᴱc6Alphaᴱc8 fields!";MOVCS(20,1);

40 A$=DSPIN$(-36,X)
50 PRINT
60 FOR I=1 TO LEN(A$)
70 PRINT USING 150;NUM(A$[I,I])
80 NEXT I
90 PRINT MOVCS(20,1);
100 B$=DSPIN$(36,X)
110 PRINT MOVCS(23,1)
120 FOR I=1 TO LEN(B$)
130 PRINT USING 150;NUM(B$[I,I])
140 NEXT I
150 IMAGE #(3DX,)

This is a test for Alpha fields!

 84 104 105 115  32 105 115  32  97  32 116 101 115 116  32 102
111 114  32
 65
108 112 104  97  32 102 105 101 108 100 115  33

 84 104 105 115  32 105 115  32  97  32 116 101 115 116  32 102
111 114  32  27
 54  65 108 112 104  97  27  56  32 102 105 101
108 100 115  33
>
```

Note that only the specified number of characters are returned and that control characters (ᵗᶜ etc.) count as input characters. This means that you could input the same display and get two different sets of characters depending on whether the length parameter is positive or negative. For example, if the string length were 21 in the previous example, the results would have been as follows:

```
L=-21
 84 104 105 115  32 105 115  32  97  32 116 101 115 116  32 102
111 114  32
 64

L=21
 84 104 105 115  32 105 115  32  97  32 116 101 115 116  32 102
111 114  32  27
```

The X parameter is a return variable and is used to let you know if the end of the display line or end of display memory was reached before the required number of characters could be input. The X parameter will be set to "0" if the required characters were input without reaching a display boundary (end of line or display memory). X will be set to "1" if the end of a data field or the end of the current line is encountered before the required number of characters are input. X will be set to "−1" if the end of display memory is reached.

If the terminal is in format mode and the field is larger than the input string specified in the DSPIN$ function, only the specified number of characters will be input and X will be set to 0.

In the previous example, the value of X returned with A$ would be "0" since no field boundary was encountered. The value of X returned with B$ would be "1" since the end of the display line was reached.

Figure 10-1 contains an example illustrating use of the return variable in the DSPIN$ function.

Example illustrating the values of the return variable.

```
10 D1$=CHR$(217) ! Display Functions On
20 D2$=CHR$(154) ! Display Functions Off
22 F1$=CHR$(215) ! Format Mode On
24 F2$=CHR$(216) ! Format Mode Off
30 PRINT MOVCA(10,1);"ᵗᶜ[1234567ᵗᶜ6ABCᵗᶜ]DEᵗᶜ81234567"
40 PRINT "ᵗᶜ[This is a test."
50 PRINT "ᵗᶜ[This is the end."
60 FOR I=1 TO 3
70 PRINT MOVCA(9+I,6);
80 A$=DSPIN$(12,X)
90 PRINT F1$;MOVCA(9+I,6);
100 B$=DSPIN$(-12,Y)
110 PRINT F2$;MOVCA(I*2,-10);
120 PRINT D1$;A$;"*";TAB(30);"X=";X;D2$
130 PRINT D1$;B$;"*";TAB(30);"X=";Y,D2$
140 NEXT I

run ⬛RETURN
67ᵗᶜ6ABCᵗᶜ]DEᵗᶜ[*            X= 0
67ABC*                        X=-1
is a test.*                   X=-1
is a test. *                  X= 0
is the end!*                  X= 1
is the end! *                 X= 0

[1234567 6ABC ]DE 81234567
[This is a test.
[This is the end!
```
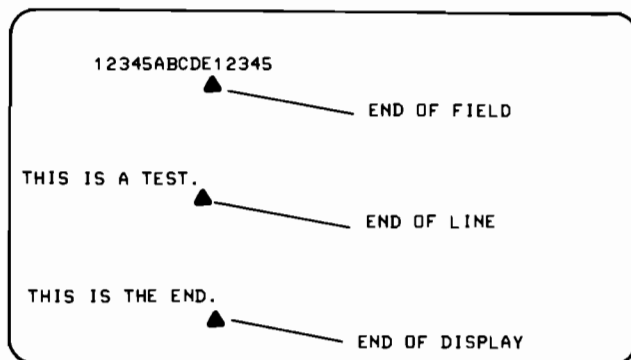

Figure 10-1. Display Input Conditions

Keyboard Input And Control

The operation of the terminal keyboard can be controlled from your BASIC program. The ways that you can control the keyboard are as follows:

- Disable all key interrupts — (GETKBD ON/OFF)
- Disable specific key interrupts — (OFF KEY #)
- Interrupt on specific keys — (ON KEY #)
- Direct process keyboard input — (GETKBD)
- Process key functions — (CHR$ and PROC$)
- Redefine the keyboard — (KEYCDE)

Disabling All Key Interrupts

The GETKBD ON and GETKBD OFF statements allow you to enable and disable the GETKBD function. They also disable and enable all keyboard interrupts from the program break character (normally CONTROL—A) or from keys specified in ON KEY # statements. (Interrupt characters received over the data communications line including characters being echoed from the keyboard, are not affected.)

The GETKBD ON and GETKBD OFF statements can be used to temporarily disable special keyboard input processing or re-enable it with a single statement.

Example:

```
10 REM Set up interrupts
20 FOR N=0 TO 9
30 ON KEY #N GOSUB 200
40 NEXT N
50 REM Disable interrupts
60 GETKBD ON
    .
    .
    .
100 REM Enable interrupts
110 GETKBD OFF
    .
(resume)
```

Disabling Interrupt Keys

You can selectively disable interrupt keys using the OFF KEY # statement. Executing OFF KEY #N (where N is the code of the desired key) will disable any program interrupt that has been set up with an ON KEY #N statement. A list of key codes is given in tables 10-3 and 12-1.

Example: Enable and then disable interrupts from the [RETURN] key.

```
10 ON KEY #13 GOTO 100
20 OFF KEY #13
```
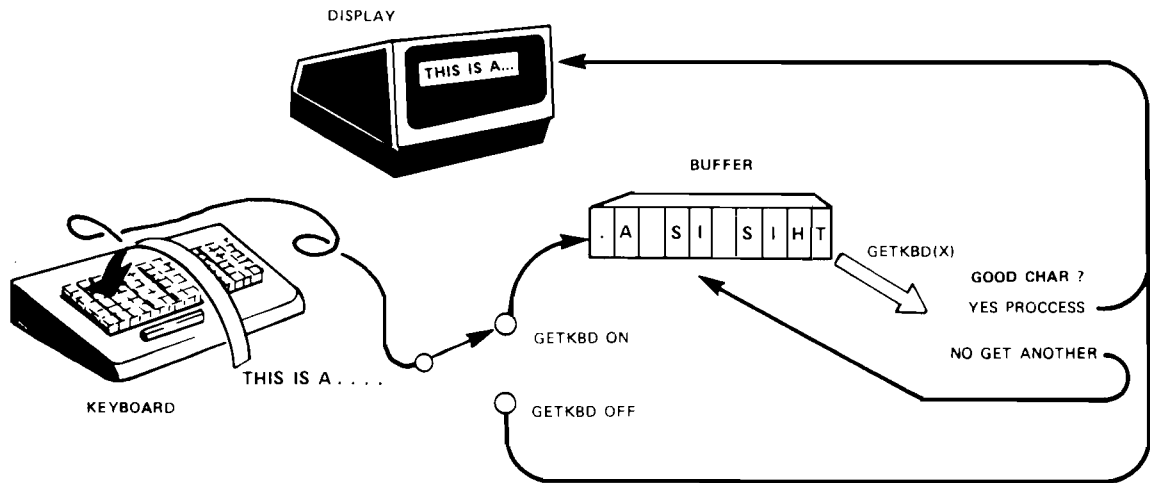
Interrupting on Specific Keys

You can cause your program to interrupt and branch when a selected key on the keyboard is pressed. If your program is dormant due to a SLEEP statement, it will resume execution of your program when the selected key is pressed. You can then perform any special processing and return to the sleep state or use the WAKEUP statement and continue executing your program.

Example: Set up an interrupt for the characters A,B, and C. Branch to addresses 100,200, and 300 respectively.

```
10 ON KEY #65 GOTO 100
20 ON KEY #66 GOTO 200
30 ON KEY #67 GOTO 300
```

Directly Acting on Keyboard Input

You can intercept input from the terminal keyboard and process the characters yourself. This is done with the GETKBD(X) function. When this technique is used, the characters are not automatically displayed on the terminal screen when the keys are pressed. This means that they are not entered into the terminal's display memory. Control keys such as BACKSPACE, CURSOR LEFT, etc. are not executed. You can use the CHR$ function to directly force their execution.

Before the GETKBD(X) function can be used, it must be enabled with the GETKBD ON statement. This causes keyboard input to be held in a buffer instead of being processed normally. Each time that you execute the GETKBD(X) function X will be set to the code of the next sequential character in the buffer. The function itself will be set to a value of "1" if a character was present in the buffer or "0" if the buffer was empty.

Example: Perform direct keyboard input and convert all digits to their word equivalents.
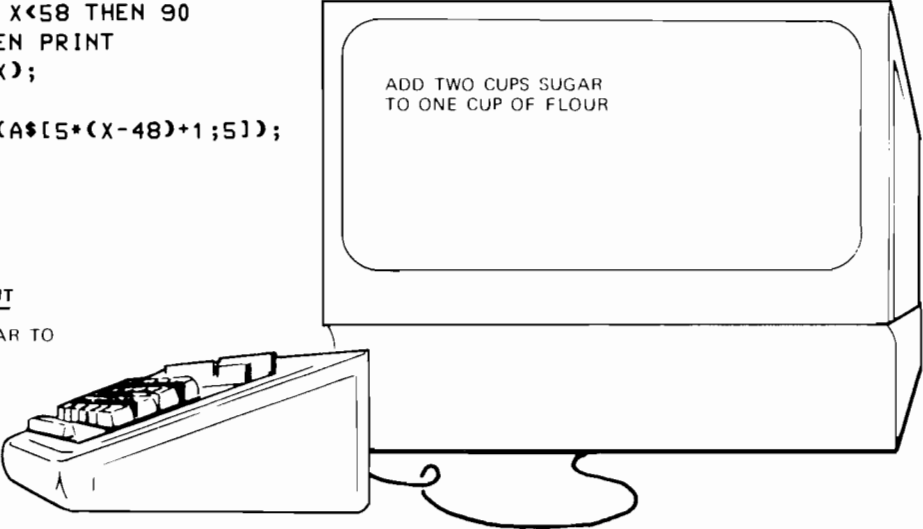
```
10 DIM A$[80]
20 A$="zero one  two  threefour five six  seveneightnine "
30 GETKBD ON
40 IF GETKBD(X)=0 THEN 40
50 IF 47<X AND X<58 THEN 90
60 IF X=239 THEN PRINT
70 PRINT CHR$(X);
80 GOTO 40
90 PRINT TRIM$(A$[5*(X-48)+1;5]);
100 GOTO 40
```

DISPLAY

ADD TWO CUPS SUGAR
TO ONE CUP OF FLOUR

KEYBOARD INPUT

ADD 2 CUPS SUGAR TO
1 CUP OF FLOUR

Processing Key Functions

You can process key codes that have been intercepted with the GETKBD(X) function or generated by your program. This is done with the CHR$(X) function in a PRINT statement. This will have the same effect as if the keycode specified by X had been input from the keyboard normally. For example, PRINT CHR$(8); would have the same effect as pressing the BACKSPACE key.

Example: Ensure that the first letter of a sentence is a capital. Assume that the beginning of a sentence is defined as "." followed by a letter.
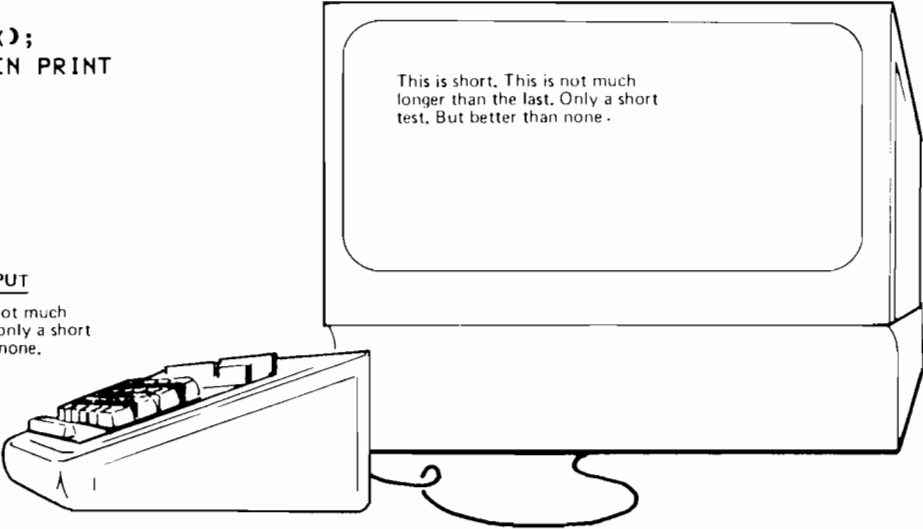
```
10 GETKBD ON
20 IF GETKBD(X)=0 THEN 20
30 IF X=46 THEN Sent=1\GOTO 70
40 IF X=32 AND Sent=1 THEN Sent=2\GOTO 70
50 IF 96<X AND X<123 AND Sent=2 THEN X=X-32
60 Sent=0
70 PRINT CHR$(X);
80 IF X=239 THEN PRINT
90 GOTO 20
```

DISPLAY

This is short. This is not much
longer than the last. Only a short
test. But better than none .

KEYBOARD INPUT

This is short. this is not much
longer than the last. only a short
test. but better than none.

The PROC$(X) function is used to expand special key functions into their equivalent escape or control sequence. The cursor right key (#46) has a keycode of 195. When this keycode is used in the PROC$ function it is translated into a two character sequence ᖶ C. This allows you to easily write control sequences to a file for example.

The statements PRINT CHR$(205) and PRINT PROC$(205) will both have the same result since the destination is the terminal. The CHR$(205) will pass a byte with a value of 205 to the terminal. The terminal uses 205 to indicate a delete line function. The PROC$(205) will be translated into ᖶ M which is the terminal escape sequence for DELETE LINE. However, if the PRINT was a PRINT #1 where file #1 was assigned to a disc file, CHR$(205) would have caused a byte with the value 205 to be recorded. The PROC$(205) would have caused two bytes ᖶ and M to be recorded.

Redefining the Keyboard

You may want to redefine keys so that they generate different codes or you may want to move existing keys to different locations on the keyboard. The KEYCDE(S,N,C) statement allows you to set the code generated by each physical key location on the keyboard. You can select up to three different codes that can be generated by each key, depending on whether or not a shift key is being pressed.

Each physical key location on the keyboard is assigned a number. A figure showing the key number assignments is given in section 12 (figure 12-1). A list of the keys given in key number order is given in table 12-1. The terminal maintains a table linking key number to a character code. This code is usually the ASCII code for the particular character. For instance, the SPACE key is physical key number 52 (see figure 12-1) and generates the decimal code 32 (see table 12-1).

Keys are "changed" by assigning a new character code to the physical key location. You can then remove the keycaps and reinstall them in a different location on the keyboard and the result would be a keyboard customized for your application. You can even assign codes outside of the 0-127 range. This may be of use if you are using an extended ASCII character set. You can use character codes with values between 0 and 255. A key assigned a code number of 0 will not generate a null; it is effectively turned off. Table 10-3 contains a list of the character codes and keys ordered by code value.

The left and right shift keys, CNTL, AUTO LF, BLOCK MODE, CAPS LOCK, REMOTE, and RESET keys cannot be changed.

The S parameter is used to select one of four possible code tables to be used when a key is pressed. The values are as follows:

S = 0 Unshifted keys
    1 Left shift key down
    2 Right shift key down
    3 Either shift key down

If you use S=3 to define a code table, the right and left shift keys will generate the same character code. This means that you have access to one unshifted character code and one shifted character code. This is the normal mode of operation. If you want you can use values of S=1 and S=2 to define two possible character codes for each shifted character. If the left shift key is used with a character key one code will be generated, and if the right shift key is used, a different character code will be sent.

Table 10-3. Character Codes and Key Numbers

Code	Key #	Char	Code	Key #	Char	Code	Key #	Char	Code	Key #	Char
0	91	Null	64	91	@	128	*		192	X	One Second Delay
1	109	SOH	65	109	A	129	*		193	39	Cursor Up
2	44	STX	66	44	B	130	*		194	62	Cursor Down
3	28	ETX	67	28	C	131	*		195	46	Cursor Right
4	93	EOT	68	93	D	132	*		196	30	Cursor Left
5	27	ENQ	69	27	E	133	*		197		Hard Reset
6	85	ACK	70	85	F	134	*		198	38	Home Down
7	77	BEL	71	77	G	135	15	Stop	199	X	Cursor Return
8	69	BS	72	69	H	136	79	G Cursor	200	X	Home To Xmit—Only
9	67	HT	73	67	I	137	87	Rb Ln	201	3	Cursor Tab
10	61	LF	74	61	J	138	78	Zoom	202	55	Clear Display
11	53	VT	75	53	K	139	6	Zoom In	203	55	Clear To End—Of—Line
12	45	FF	76	45	L	140	22	Zoom Out	204	80	Insert Line
13	13	CR	77	68	M	141	06	Clear	205	72	Delete Line
14	60	SO	78	60	N	142	79	G Dsp	206	56	Control—Insert Char (Wrap)
15	75	SI	79	75	O	143	15	A DSP	207	64	Control—Delete Char (Wrap)
16	83	DLE	80	83	P	144	7	Draw	208	64	Delete Char
17	11	DC1	81	11	Q	145	23	Move	209	56	Insert Char On
18	35	DC2	82	35	R	146	86	Autoplot Menu	210	56	Insert Char Off
19	101	DC3	83	101	S	147	95	Autoplot	211	31	Roll Up
20	43	DC4	84	43	T	148	95	Axes	212	70	Roll Down
21	59	NAK	85	59	U	149	78	Text	213	47	Next Page
22	36	SYN	86	36	V	150	14	T Ang	214	54	Prev Page
23	19	ETB	87	19	W	151	22	T sze	215	X	Format Mode On
24	20	CAN	88	20	X	152	74	Enter	216	X	Format Mode Off
25	51	EM	89	51	Y	153	105	Break	217	57	Display Functions On
26	12	SUB	90	12	Z	154	57	Display Functions Off	218	*	
27	2	ESC	91	99	[	155	24	Command	219	X	Start Unprotected Field
28	99	FS	92	92	\	156	32	Read	220	*	
29	16	GS	93	21	]	157	40	Record	221	X	End Unprotected Field
30	21	RS	94	106	^	158	48	Softkeys	222	X	Send Terminal Status
31	106	US	95	107		159	41	Control—Test	223	X	Non—Displaying Terminator
32	52	SP	96	91	'	160	32	Control—Read	224	X	Relative Cursor Sense
33	10	!	97	109	a	161	54	Control—Prev Page	225	X	Absolute Cursor Sense
34	18	"	98	44	b	162	87	G. Up Arrow	226	X	Enable Keyboard
35	26	#	99	28	c	163	23	G. Right Arrow	227	X	Disable Keyboard
36	34	$	100	93	d	164	14	G. Down Arrow	228	X	Send Display
37	42	%	101	27	e	165	7	G. Left Arrow	229	X	Fast Binary Read
38	50	&	102	85	f	166	86	Cursor Fast	230	X	Disconnect Modem
39	58	'	103	77	g	167	*		231	X	Soft Reset
40	66	(	104	69	h	168	*		232	38	Home To Protected Field
41	82	)	105	67	i	169	*		233	3	Back Tab
42	29	*	106	61	j	170	*		234	47	Display Softkey Menu
43	37	+	107	53	k	171	*		235	47	Return Normal Display
44	76	,	108	45	l	172	*		236	25	Memory Lock On
45	98	—	109	68	m	173	*		237	25	Memory Lock Off
46	84	.	110	60	n	174	*		238	*	
47	92	/	111	75	o	175	*		239	X	Soft Carriage Return
48	90	0	112	83	p	176	*		240	73	F1
49	10	1	113	11	q	177	63	Set Tab	241	81	F2
50	18	2	114	35	r	178	71	Clear Tab	242	89	F3
51	26	3	115	101	s	179	71	Control—Clear Tab	243	97	F4
52	34	4	116	43	t	180	30	Control—Left Cursor	244	112	F5
53	42	5	117	59	u	181	46	Control—Right Cursor	245	104	F6
54	50	6	118	36	v	182	X	Alpha Only Field	246	96	F7
55	58	7	119	19	w	183	X	Numeric Only Field	247	88	F8
56	66	8	120	20	x	184	X	Alphanumeric Field	248	X	Data Comm Self—Test
57	82	9	121	51	y	185	*		249	X	Monitor Mode On
58	29	:	122	12	z	186	*		250	41	Terminal Self—Test
59	37	;	123	99	{	187	*		251	X	Start Xmit—Only Field
60	76	<	124	16	:	188	*		252	*	
61	98	=	125	21	}	189	*		253	*	
62	84	>	126	106	~	190	*		254	*	
63	92	?	127	107	■	191	*		255	X	Display Enhancement Indicator

Notes: X = Not accessible from the keyboard   * = Not used.

Note that shifted and unshifted character codes need not be related to each other. For example, an unshifted "A" could generate the code for a lowercase "a" and the shifted "A" could generate an "@".

A simple application of this feature would be to allow you to access an extended character set without using any special coding. For example, some text processing systems use as many as four types of "-". The single ASCII character may have to be translated into a hyphen, a minus, or a dash. On the computer system only one graphic character is normally used. By moving the "=" sign to the "+" key you can have three possible minus characters on the "—" key.

Example:

```
10 REM Let the "=" sign be a left shift ";" key
15 KEYCDE(1,37,61)
20 REM Let the "+" sign be a right shift ";" key
25 KEYCDE(2,37,43)
30 REM Let a minus sign be the unshifted "-" key
35 KEYCDE(0,98,45)
40 REM Let a hyphen be a left shift "-" key
45 KEYCDE(1,98,167)
50 REM Let a dash be a right shift "-" key
55 KEYCDE(2,98,168)
```

The character codes 167 and 168 were chosen for the two new characters since they are the first two unused character codes in table 10-3. The computer can be programmed to accept the new character codes and convert them to whatever the required code is. If you perform direct keyboard input so that your program is handling all keyboard input, you can simply change the terminal key codes to match the computer's character codes.

Data Communications

Enabling the Data Communications Functions

The GETDCM ON statement is used to enable direct datacomm input. This allows your program to monitor the datacomm input buffer directly rather than through the terminal file system.

Note that if a GETDCM ON statement has been executed and you execute a SLEEP statement without executing a GETDCM OFF statement first, datacomm input will not be echoed to BASIC. This means that your program will not interrupt on selected characters (ON KEY statement). You should always execute a GETDCM OFF statement before executing a SLEEP statement. This means that you may have to re-execute a GETDCM ON statement on return to the program from a slept condition.

Also, if you have left datacomm assigned to a file number (ASSIGN "DATACOMM" TO #1) without deassigning the file (ASSIGN * TO #1 or ASSIGN "OTHERFILE" TO #1), the datacomm input will not be echoed to the BASIC interpreter.

Single Byte Transfers

Two built-in functions are used to bypass normal datacomm handshakes, such as DC1 and DC2, to allow you to establish your own handshake protocol. These functions are:

- GETDCM(character)
- PUTDCM(character)

GETDCM is also useful for scanning datacomm input character by character since it returns only one character to the user. Similarly, PUTDCM sends only one character to the datacomm.
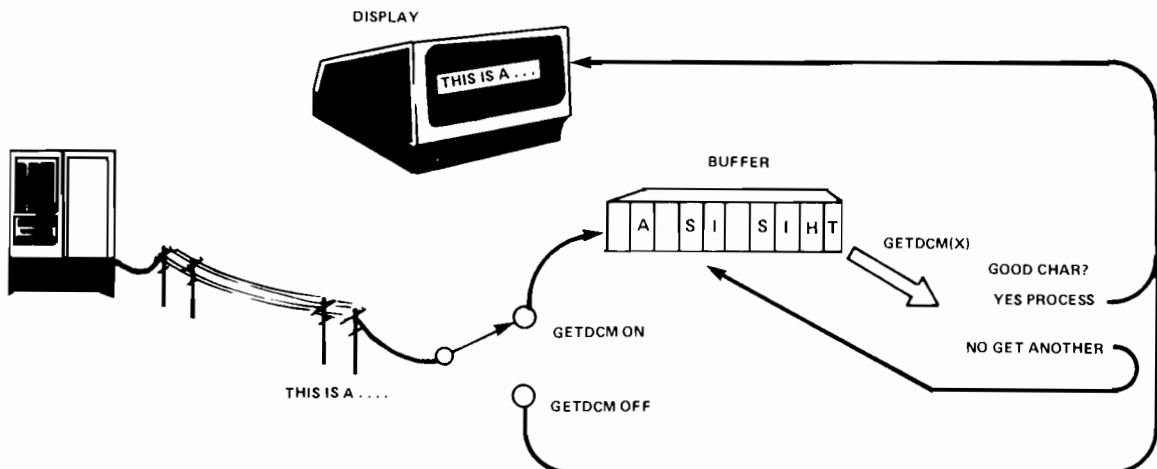
The following terminal program assumes a simple program is available to run on a remote CPU. This example is not intended as a guide to datacomm programming. It is intended to illustrate the mechanics of interacting with a host CPU. The exact techniques used will depend on whether or not you are using full duplex, the protocol used by the host computer, and other factors.

Computer

```
DCTEST
10 INPUT N
20 IF N=0 THEN 50
30 PRINT N**2
40 GOTO 10
50 END
```

Terminal Display

```
>
run
? 3
J*J=9
? 4
J*J=16
? 5
J*J=25
? 6
J*J=36
? 7
J*J=49
? 0
>
```

Dialog

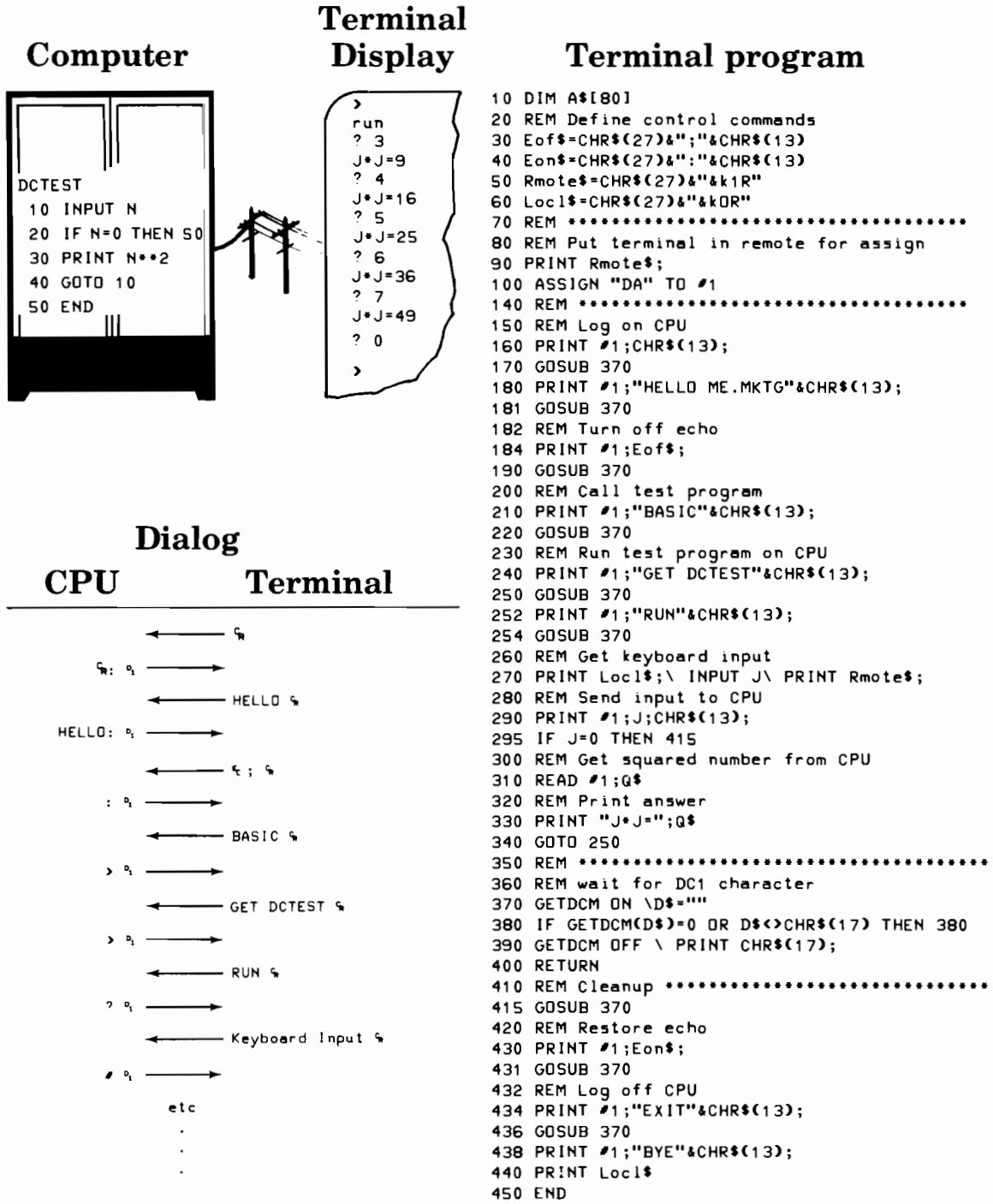| CPU	Terminal
	←——— ⌐ₕ
⌐ₕ: ₒ₁ ——→	
	←——— HELLO ⌐
HELLO: ₒ₁ ——→	
	←——— ⌐ ; ⌐
: ₒ₁ ——→	
	←——— BASIC ⌐
> ₒ₁ ——→	
	←——— GET DCTEST ⌐
> ₒ₁ ——→	
	←——— RUN ⌐
? ₒ₁ ——→	
	←——— Keyboard Input ⌐
⌐ ₒ₁ ——→	
etc	
.	
.	
.	

Terminal program

```
10 DIM A$[80]
20 REM Define control commands
30 Eof$=CHR$(27)&";"&CHR$(13)
40 Eon$=CHR$(27)&":"&CHR$(13)
50 Rmote$=CHR$(27)&"&k1R"
60 Locl$=CHR$(27)&"&kOR"
70 REM ********************************
80 REM Put terminal in remote for assign
90 PRINT Rmote$;
100 ASSIGN "DA" TO #1
140 REM ********************************
150 REM Log on CPU
160 PRINT #1;CHR$(13);
170 GOSUB 370
180 PRINT #1;"HELLO ME.MKTG"&CHR$(13);
181 GOSUB 370
182 REM Turn off echo
184 PRINT #1;Eof$;
190 GOSUB 370
200 REM Call test program
210 PRINT #1;"BASIC"&CHR$(13);
220 GOSUB 370
230 REM Run test program on CPU
240 PRINT #1;"GET DCTEST"&CHR$(13);
250 GOSUB 370
252 PRINT #1;"RUN"&CHR$(13);
254 GOSUB 370
260 REM Get keyboard input
270 PRINT Locl$;\ INPUT J\ PRINT Rmote$;
280 REM Send input to CPU
290 PRINT #1;J;CHR$(13);
295 IF J=0 THEN 415
300 REM Get squared number from CPU
310 READ #1;Q$
320 REM Print answer
330 PRINT "J*J=";Q$
340 GOTO 250
350 REM ********************************
360 REM wait for DC1 character
370 GETDCM ON \D$=""
380 IF GETDCM(D$)=0 OR D$<>CHR$(17) THEN 380
390 GETDCM OFF \ PRINT CHR$(17);
400 RETURN
410 REM Cleanup ********************************
415 GOSUB 370
420 REM Restore echo
430 PRINT #1;Eon$;
431 GOSUB 370
432 REM Log off CPU
434 PRINT #1;"EXIT"&CHR$(13);
436 GOSUB 370
438 PRINT #1;"BYE"&CHR$(13);
440 PRINT Locl$
450 END
```

Program Control

The program control functions available in BASIC are made up of the following functional groups:

- Terminal and resource functions (COMMAND and FRE)
- Error handling functions (ERROR, ON ERROR, ERRL, ERRN, RESUME)
- BASIC suspend (SLEEP and WAKEUP)

Terminal And Resource Functions

The COMMAND statement allows your program to access the standard terminal functions to control flexible discs or cartridge tapes, assign I/O source and destination devices, enable or disable edit mode, make direct data transfers, or to request the time from the terminal's internal clock. A complete list of terminal commands is given in the manual titled *Getting Started with Your HP 2647F* (part number 02647-90036).

The COMMAND statement executes a terminal command just as if it had been entered into the terminal's command channel. The format of the COMMAND statement is as follows:

```
COMMAND <command string> [ ,variable]
```

The <command string> is the terminal command. It may be the command in quotes or a string variable containing the command. If the return variable is used, it will be set to "0" if the statement executed properly, if the statement did not execute properly, the variable will be set to an error code. A list of error codes is given in Appendix E. If a return variable is not specified and an error occurs, the BASIC program will halt.

Example: Copy a file, select the display as the source device and enable Edit Mode.

```
10 COMMAND "C F MEMO DI"   ! Copy file Memo to screen
20 IF X>0 THEN 90          ! Process errors
30 COMMAND "A S DI",X      ! Make source assignment
40 IF X>0 THEN 90          ! Process any errors
50 COMMAND "E E" ,X        ! Enable Edit
60 IF X>0 THEN 90          ! Process any errors
 .
 .
 .
```

An alternate program would be as follows:

```
10 A$="A S DI E E   "      ! Command strings
20 FOR I=1 TO 13 STEP 6
30 COMMAND A$[I;6],X       ! Output Command
40 IF X>0 THEN 90          ! Process Errors
50 NEXT I
 .
 .
 .
```

Error handling routines are described elsewhere in this section.

Memory Space Available

The FRE function can be used to determine the amount of program or variable storage remaining. The function has the following form:

2647A

FRE(expression)

If the expression is numeric (i.e. N, 5, TAN(X)) the value returned is the amount of unused program and numeric variable space (bytes) remaining. If the expression is a string expression (i.e. A$, "MEMORY"), the value returned is the amount of string variable space (bytes) remaining.

2647F

FRE

The expression parameter is not necessary on the 2647F since the FRE function returns the total amount (bytes) of workspace remaining regardless of the type of variable being used.

The FRE function is useful to determine the amount of terminal memory required for a given program application. It can be used together with the REMOVE (or, on the 2647A, the SET SIZE) command to obtain the proper amount of memory space for your program.

Error Handling

BASIC/47 provides a complete error handling capability which allows you to detect program errors and provide for automatic recovery without halting your program. You can even define your own error conditions complete with your own error code and message.

The technique of error processing is to use an ON ERROR statement to cause a branch to your error handling routine whenever BASIC detects an error condition. Once you are in your error handling routine you can check the error variables ERRN and ERRL to determine the type of error and where in your program that it occurred. ERRN contains the code value of the error. ERRL contains the line number of the statement that caused the error.

In many cases it may be possible to correct the error condition programmatically or to indicate to an operator how they can correct the error condition.

At the end of the error handling routine you can use a RESUME statement to cause BASIC to continue executing the program at a specified line number. If you detect an error that you cannot recover from, you can use the ON ERROR GOTO 0 statement. This will cause the program to halt and print out the error information.

Note that to be effective, the ON ERROR GOTO statement that you use to branch to your recovery routine must be executed before an error occurs. If an error occurs before the branch statement has been executed, the program will halt.

Example: Assume that the terminal is placed in Edit Mode from the BASIC program. Entering Edit Mode with the terminal's source and destination devices CTUL??? and CTUR??? will generate an error if the source file was not closed after its last use. The following program recovers from this error and also guides the operator in correcting other frequently encountered errors.

```
10 REM Arm error recovery routine
11 ON ERROR GOTO 16
12 REM Turn on Edit Mode
13 COMMAND "E E"
14 END
15 !****************************************************
16 REM Error Handling Routine
17 ON ERRN=262 GOTO 36        !"SOURCE=DESTINATION"
18 ON ERRN=497 GOTO 27        !NO DISC IN DRIVE
19 ON ERRN=291 GOTO 23        !END OF DATA ON LEFT DRIVE
20 REM Can't recover
21 ON ERROR GOTO
22 !****************************************************
23 REM CTU File Open
24 PRINT "Mark file to close and press RETURN";
25 GOTO 31
26 !****************************************************
27 REM No disc in drive
28 PRINT "Insert unprotected disc in drive and press RETURN";
29 !****************************************************
30 REM Wait for operator and clean up screen
31 LINPUT A$                  !Wait for operator
32 PRINT CHR$(193);CHR$(205); !Clear program message
33 PRINT CHR$(13);            !Clear terminal message
34 RESUME
35 !****************************************************
36 REM Reassign display as source

37 COMMAND "A S DI",X
38 REM If assign error can't recover
39 IF X=0 THEN 33
40 PRINT "Recovery error";
41 END
```

You can define your own special error conditions or change the message and error code produced by an existing terminal error condition. The ERROR statement allows you to set an error code and an optional error message. The statement then forces the error condition. If an ON ERROR statement has not been executed, the error code or optional message will be displayed immediately. If the ON ERROR statement has been executed, the specified error code will be passed to the error handling routine as ERRN. The line number of the ERROR statement will be passed as ERRL.

Example: The following program changes the error message for error code 1027.

```
10 ON ERROR GOTO 100
20 REM Generate Loop Error
30 NEXT I
   .
   .
   .
   .
90 END
100 REM Error code=1027
110 IF ERRN<>1027 GOTO 150
120 REM Change message
130 ERROR 1027 , "Incomplete Loop"
150 ON ERROR GOTO 0
```

After the error handling is complete the RESUME statement should be used to continue program execution. The RESUME statement reenables the ON ERROR statement restoring the error trapping function. If the RESUME statement is not used after processing an error, the next error that occurs will cause the program to halt.

The RESUME statement can use one of the following forms:

```
RESUME
RESUME NEXT
RESUME line number
```

When RESUME alone is used, the program statement that caused the error will be re-executed. RESUME NEXT causes execution to continue with the statement following the statement where the error was detected. If a line number is specified, execution will continue at the new line number. The specified line number must be within the current program unit.

Example: The following program detects errors and then resumes execution at different points in the program depending on the error code. If the error code is 1040, the program will resume at the line following the statement that caused the error. If the error code is 1041, the program will resume at statement 20. If any other error code is detected, the program will attempt to re-execute the statement that caused the error. (Note that this will cause the program to loop forever.)

```
10 ON ERROR GOTO 50
20 INPUT "Enter a number",N
30 PRINT "100\";N;"=";100/N
40 GOTO 20
50 REM ERROR ROUTINE
60 ON ERRN-1039 GOTO 80,90
70 RESUME
80 REM Result too big
82 PRINT "BIG"
84 RESUME NEXT
90 REM Division by zero
92 N=N+.000001
94 RESUME 20
```

Suspending BASIC

You can suspend BASIC causing the terminal to return to normal operation. This is done using the SLEEP statement. The difference between suspending BASIC and using the EXIT command is that the terminal can be programmed to return to BASIC when a particular key is pressed. This feature can be used to perform special processing on data or perform some device control operations and then return to a suspend state. It allows the application program to be activated automatically from the keyboard without the operator even being aware that BASIC has been called. Note that this is not the same as the terminal "SUSPEND" command described in the *2647F Reference Manual,* Section 5.

There are two ways that can be used to return to BASIC:

• Press the BASIC Break key (normally CONTROL-A)

• Press a key whose keycode has been used in an ON KEY statement

If BASIC is awakened using the Break key, the program is interrupted at the last executed SLEEP statement and a Break message is displayed.

If BASIC is reentered by pressing a keycode that has been used in an ON KEY statement, the program will resume execution as specified in the ON KEY statement. If the GOTO form of the ON KEY statement was used, the program will continue executing normally until another SLEEP statement is executed. If the GOSUB or CALL forms of the ON KEY statement were used, BASIC will sleep again automatically when the RETURN or SUBEND statements are executed. The sleep can be prevented if a WAKEUP statement is executed in the specified subroutine or subprogram.

Note that while BASIC is sleeping, interrupt keys are enabled regardless of a previously executed GETKBD ON statement.

The WAKEUP statement cancels the effect of the SLEEP statement and returns the program to normal execution.

The following examples use the programs described earlier under Keyboard Input. This time the ON KEY statement will be used instead of direct keyboard input. The BASIC program will run only when one of the selected keys is pressed.

Example: Perform direct keyboard input and convert all digits to their word equivalents.

```
10 DIM A$[80]
20 A$="zero one  two  threefour five six  seveneightnine "
30 ON KEY #48 GOSUB 150
40 ON KEY #49 GOSUB 160
50 ON KEY #50 GOSUB 170
60 ON KEY #51 GOSUB 180
70 ON KEY #52 GOSUB 190
80 ON KEY #53 GOSUB 200
90 ON KEY #54 GOSUB 210
100 ON KEY #55 GOSUB 220
110 ON KEY #56 GOSUB 230
120 ON KEY #57 GOSUB 240
130 X=57
140 SLEEP
150 X=X-1
160 X=X-1
170 X=X-1
180 X=X-1
190 X=X-1
200 X=X-1
210 X=X-1
220 X=X-1
230 X=X-1
240 PRINT TRIM$(A$[5*(X-48)+1;5]);
250 X=57
260 RETURN
```

Note that it is necessary to arm each key separately. This is to allow you to determine which key has been pressed. When the ON KEY # statement is used you cannot use GETKBD to get the key that caused the interrupt. By using different entry points (lines 150-240) the interrupting key can be determined. The program is longer than when GETKBD was used but the BASIC program is only active when the 0-9 keys are pressed. This allows the terminal to process alpha characters faster and makes it easier to interface to a remote CPU.

Example: Ensure that the first letter of a sentence is a capital. Assume that the beginning of a sentence is defined as a "." followed by a letter.

```
10 REM ***** arm "." and " " keys *****
15 ON KEY #46 GOSUB 40
20 ON KEY #32 GOSUB 45
25 SLEEP
30 REM
35 REM ***** process "." and " " ******
40 PRINT "."; \ Sent=1 \ RETURN
45 PRINT " "; \ If Sent=0 THEN RETURN
50 REM
55 REM ******* monitor keyboard *******
60 GETKBD ON
65 IF GETKBD(X)=0 THEN 65
70 IF 96<X AND X<123 THEN X=X-32
75 IF X=239 THEN PRINT
80 GETKBD OFF\Sent=0
85 PRINT CHR$(X);\RETURN
```

A Graphics Language (AGL) _____ 11

Introduction
What Is "A Graphics Language" (AGL)?

AGL is an extension to BASIC that provides easy to use graphics commands. If you have an HP 2647F Graphics Terminal, you can perform these additional graphics operations. These graphics operations are a subset of the AGL functions available on other Hewlett-Packard graphics systems.

AGL consists of powerful graphics functions that allow you to perform graphics operations with a minimum of programming.

For example, to draw a labeled grid it would take the following individual escape sequences:

```
 10 REM DRAW AND LABEL GRID
 20 PRINT "ENTER XMIN, XMAX, AND XINTERVAL"
 30 INPUT X1, X2, X3
 40 PRINT " ENTER YMIN, YMAX, AND YINTERVAL"
 50 INPUT Y1, Y2, Y3
 60 REM DRAW X/Y LABELS
 70 PRINT CHR$(27)&"*m";X1;Y1;"J"
 80 FOR I = X1 TO X2 STEP X3
 90 I$=VAL$(I)
100 PRINT CHR$(27)&"*1"&I$
110 NEXT I
120 REM SAME FOR Y LABELS
   .
   .
   .
130 REM DRAW GRID
140 REM SELECT DOTTED LINE
150 PRINT CHR$(27)&"*m7B"
160 FOR I=X1 TO X2 STEP X3
170 PRINT CHR$ (27)&"*pa";I;Y1;"b";I+5;Y2;"A"
180 NEXT I
190 REM SAME FOR Y GRID
   .
   .
   .
200 REM RESTORE LINE TYPE
210 PRINT CHR$(27)&"*m1B"
220 REM DRAW AXIS
   .
   .
   .
240 REM DRAW MAJOR AND MINOR TICS
   .
   .
   .
250 END
```

In AGL the same operation could be performed as follows:

```
> 5 PLOTR
>10 LOCATE (20,60,20,60)
>20 SCALE (-30,30,-30,30)
>30 LGRID (5,5,0,0,2,2)
>RUN
```
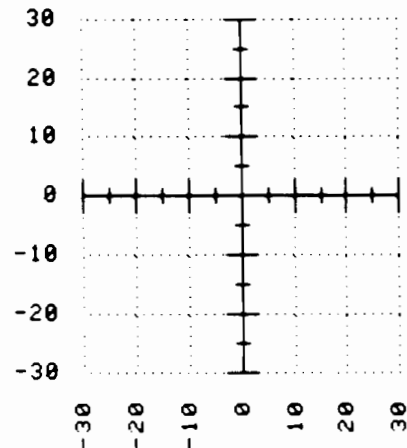
AGL requires fewer statements, no control characters, and English—like BASIC commands. In other words, its easier to use for high level graphics operations.

The rest of this section describes the AGL functions available in the HP 2647F Graphics Terminal and gives examples of their use.

AGL Terminology

Throughout this section some new terms and mnemonics are used to describe a particular operation or syntax. In many cases a term or mnemonic is used before a complete definition can be given. For these cases, the following provides enough information to make each term meaningful without giving complete details of each. The complete meaning of each term becomes clearer as you read the description of each statement. Complete details regarding each statement's syntax is given with that statement. The terms and mnemonics to be defined here are as follows:

Plotting Area
    Hard Clip Limits
    Soft Clip Limits

Unit-of-Measure Modes
    Graphics Display Units (GDUs)
    User Defined Units (UDUs)
    Metric Units
    Current Units

Clipping

Data Values for Plotting Points

Plotting Area

Plotting devices have physically-imposed hard clip limits (for example, the terminal display screen and the HP 9872 Plotter's platen size) that restrict the pen movement.

The physically-imposed hard clip limits and the resulting plotting area depends upon the particular plotting device used. The hard clip limits can be made smaller (for example, to fit a piece of paper) under program control.

The hard clip limits can also be made smaller with the front panel controls on a plotter (refer to the appropriate plotter manual).

Whether the plotting area is defined by the physically-imposed hard clip limits or it is defined by the programmer, it is still referred to as the area within the hard clip limits.

The hard clip limits are set to physically imposed default values by the GPON statement or to the programmer defined values by the LIMIT statement.

Another set of limits that further define the plotting area is the soft clip limits. The soft clip limits are normally defaulted to the hard clip limits, but can be altered under program control. The soft clip limits are discussed further under Soft Clip Limits.

Hard Clip Limits

No marks can be made outside of the area defined by the hard clip limits. Hard clip limits are intended to clip both vectors (lines) and labels. AGL automatically stops plotting at the current hard clip limits.

The hard clip limits can be changed programmatically with the LIMIT and SETAR statements. At no time can the hard clip limits be made larger than those physically imposed by the plotting device. See Figure 11-1 for a possible relationship between the physically-imposed size of the 2647 display screen and the programmer's chosen plotting area.

```
10 PLOTR
20 LIMIT (0,200,0,100)
30 FRAME
>RUN
```
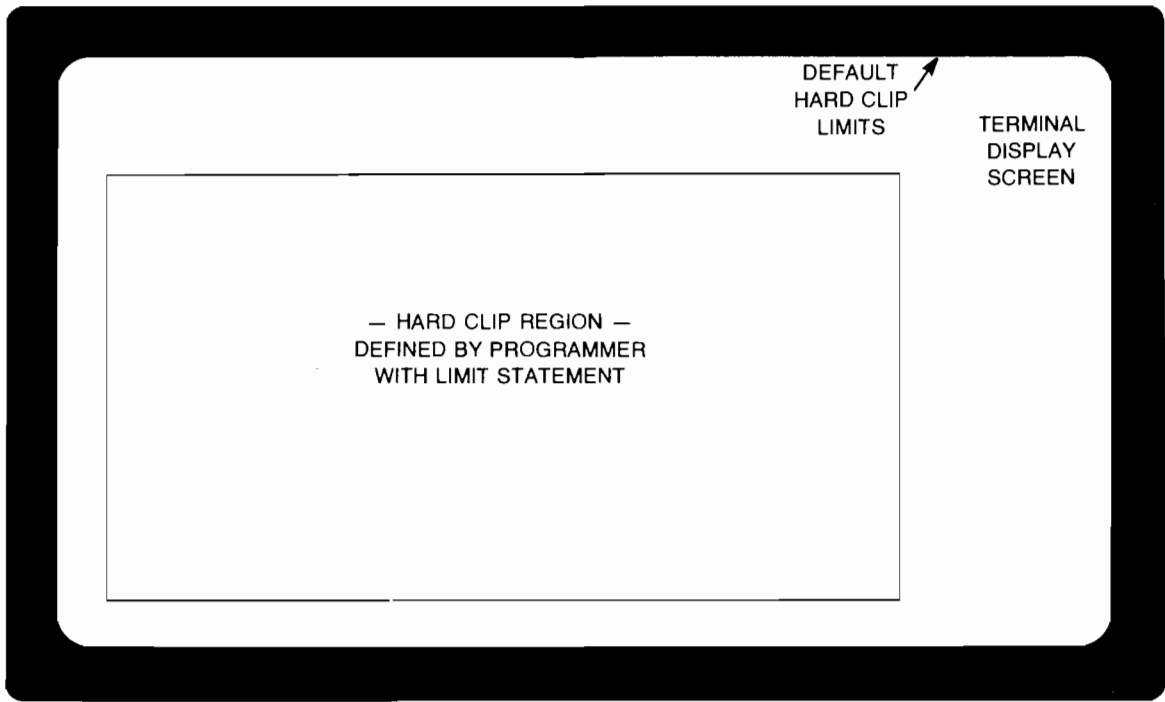
Figure 11-1. Plotting Area

Soft Clip Limits

The soft clip limits allow you to redefine the plotting area within the hard clip region. Only vectors (not labels) are affected by the soft clip limits. This allows you to plot data within the soft clip limits while labels and titles can appear outside of these boundaries. The soft clip limits can be changed and redefined at many points throughout an application program.
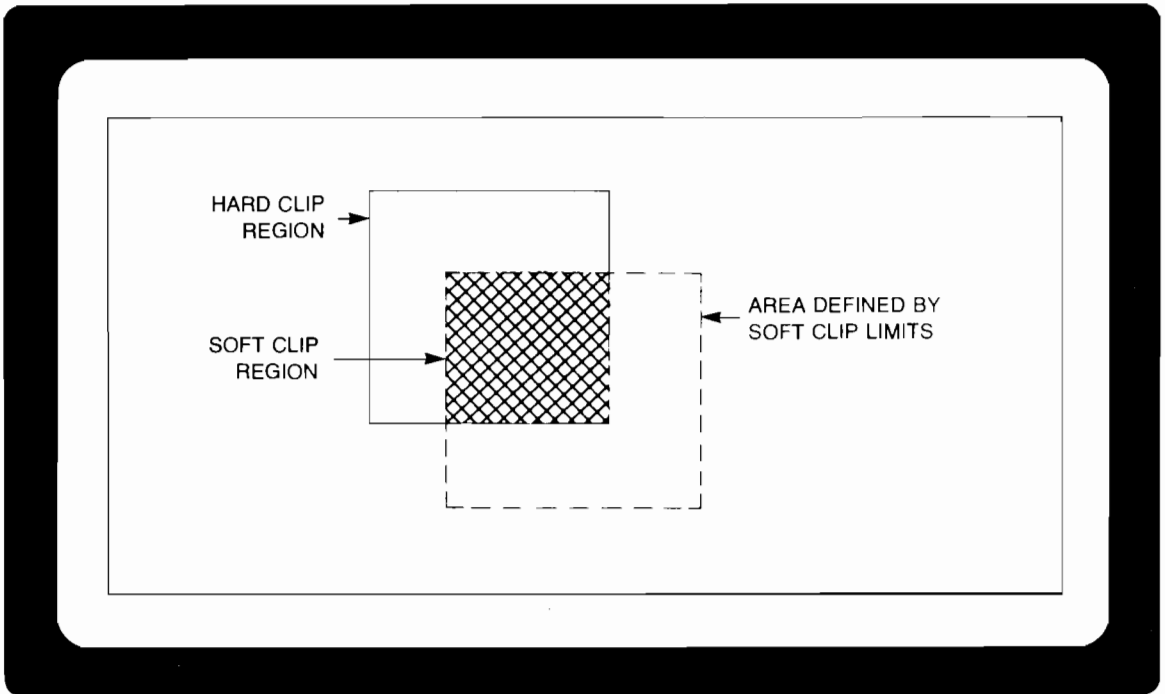

Figure 11-2. Soft Clip Region

If the area defined by the soft clip limits goes outside the hard clip region, the intersection of the two areas is stored by AGL as the soft clip region (see Figure 11-2). If the soft-clip region falls outside of the hard-clip region, and there is no intersection between them, an error will occur.

You can define the soft clip limits with the LOCATE, MARGIN, or CLIP statements.

Once the soft clip limits have been defined, you can map data to the soft clip region with the SCALE or SHOW statements. This region is useful when you wish to plot data in your own units and confine the plot to a specific region in the display. In this case, the LOCATE statement is used in conjunction with SCALE or SHOW to allow plotting in user units. AGL automatically maps user units to plotter units so you need only concern yourself with units meaningful to you.

Unit-Of-Measure Modes

There are three unit-of-measure modes that can be selected under program control: Graphic Display Units (GDUs), User Defined Units (UDUs), and Metric Units.

User Defined Units (UDUs)

User defined units (UDUs) are the units defined by you for a particular application. Several AGL statements are provided to allow you to define UDUs. These units are automatically scaled and translated to machine units by AGL. You can switch between unit modes at different points in your application program. This allows you to plot in units that you understand.

Graphic Display Units (GDUs)

Graphic Display Units (GDUs) are defined as being one percent of the length of the shortest side of the space bounded by the hard clip limits. Thus the short side is 100 GDUs in length, and the long side is
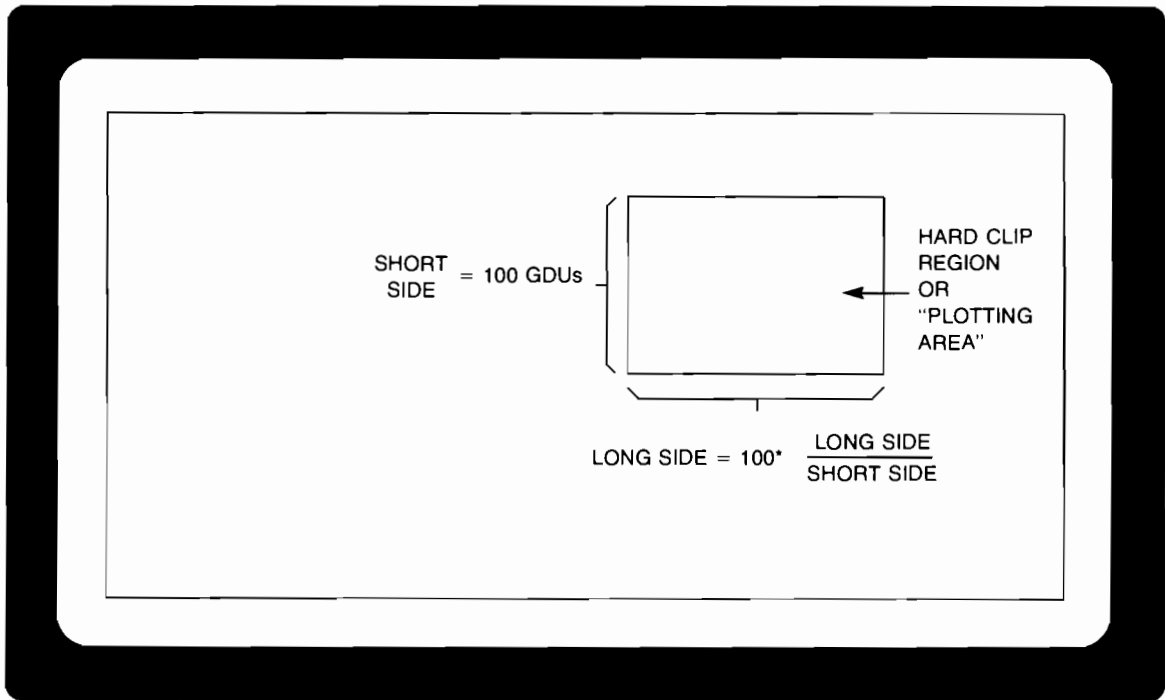
= (100 GDUs) * (LONG SIDE / SHORT SIDE).

Figure 11-3. Graphic Display Units (GDUs)

This unit mode allows the user to reference the plotting area on a percent of full scale basis.

Metric Units

The unit of measure in the Metric Unit mode is the millimeter. This mode is implemented as a special case of UDUs, such that plotting statements executed in this mode will put a line on the plotting area physically scaled to millimeters. This mode is useful in drafting applications to plot physically scaled drawings. Metric units are turned on by the MSCALE command. When plotting to the terminal display, drawings may be distorted slightly due to round off errors.

Current Units

The term current units refers to the last unit of measure mode that was set.

Clipping

The term "clipping" refers to a line or label not drawn on the plotting area due to restrictions imposed by the hard and soft clip limits.

The restriction imposed by the hard clip limits is that nothing will be plotted outside of the area defined by these limits. Any data which is to be plotted outside of these limits will be "clipped" (not drawn).

The soft clip limits affect only data, not labels. Therefore, data which is to be plotted outside of the soft clip boundaries will be "clipped" (not drawn), but labels can be drawn.

If the area defined by the soft clip limits goes outside the area defined by the hard clip limits, the intersection of the two areas is stored by AGL as the soft clip region.

Data Values For Plotting Points

When plotting points to define your graphics drawings, there will be a certain range of data values which can be referenced by the terminal or plotter. The range of data values will be device dependent. For the graphics display screen (CRT) on the HP 2647 the range of data values is: $-16,384 < X,Y < +16,383$.

Your graphics drawings (which are made up of points within the range of data values) will be plotted on the plotting area described earlier. Remember the plotting area is defined by the hard and soft clip limits.

Function Groups

The AGL functions can be separated into four groups:

- Set-Up Functions
- Plotting Functions
- Axis and Labeling Functions
- Interactive Functions

Appendix D contains a summary of the AGL functions. The remaining paragraphs in this section provide detailed descriptions of the functions.

> **NOTE:** The AGL functions are described for the HP 2647F Graphics Terminal. Functions whose operations vary on other graphic devices are also noted.

Function Syntax

The general form for all functions is:

```
keyword (P1,P2,P3, ...Pn)
Where keyword = Function name
            Pn = Numeric expressions or variables
```

Parameters are read left to right. Missing parameters are assigned default values where possible. Refer to the function descriptions for the default values for the various functions. Arrays and strings cannot be used as parameters.

Examples:

```
>10 X = 3
>20 Y = -10
>30 Npen = 2
>40 PLOT (X,Y,Npen)
```

This would read the X coordinate as 3, the Y coordinate as -10, and the pen parameter as 2 (lift after the plot).

```
>10 PLOT (5,10)
```

This would read X coordinate as 5, Y coordinate as 10, and assign the default value for the pen position. The pen default for the PLOT function is 1 (move and then put pen down). Therefore PLOT(5,10)=PLOT(5,10,1).

Set-Up Functions
PLOTR

The PLOTR statement must be used to initialize AGL. In addition, the PLOTR statement selects and initializes the plotting or display device to be used.

Syntax

PLOTR [(LU# [, action [, HP-IB [, LOGLU]]])]

The LU# parameter specifies the logical unit number of the plotting device. Default is 0, which is the terminal's graphic display.

The action parameter specifies one of five possible device operations.

0: Terminates use of device.

1: Default. Selects new device and initializes (level 2A of GPON). A clear operation (GCLR) is not performed. No buffering.

2: Resumes use of device (no initialize).

3: Suspends device.

4: Same as action 1 above. On other plotting devices this may select a device with data buffering.

The HP-IB parameter specifies the HP-IB address of the output device. This parameter is ignored if the LU# is 0. If a LU# has not previously been assigned, the HP-IB parameter will be assigned to the logical unit number.

The LOGLU parameter specifies the logical unit number of the Log (write only) device. Default is 0 which ignores the Log device assignment. This parameter is only used when the LU# parameter is not 0. If the LU# parameter is 0, then the LOGLU parameter is ignored.

The default values for the parameters are:

LU# = 0 (Display)

Action = 1

HP-IB = 0 is LU# = 0, else HP-IB = LU#

LOGLU = 0

Explanation

The default units after the PLOTR statement are GDUs. This can be changed using the SETUU, SCALE, SHOW, or MSCALE statements.

Examples

```
>10 PLOTR (1,0)
>20 PLOTR (A,L)
>30 PLOTR (A+B,C-1)
```

GPON

The GPON statement resets the plotting device. How the plotting device is reset depends on which parameter is supplied with the GPON statement.

Syntax

GPON [(<level>)]

The level parameter is one of the following:

Level 1: Sets the hard clip limits equal to the physical plotting limits (default values) of the particular plotting device. In addition, performs Level 2 and Level 3 operations.

Level 2: Clears display screen (paper moving devices do not advance paper).

Level 2A: Reads the hardware-set hard clip limits from the plotting device. This level is used by the PLOTR and SETAR statements when these statements have no parameters.

Level 2B: Sets the hard clip limits on the plotting device. This level is used by the LIMIT statement.

Level 2C: Resets the hard clip limits on the plotting device.

Calculates the GDUs based on the new hard clip limits so that the hard clip region will have the desired aspect ratio. This level is used by SETAR when parameters are given.

—PEN(#) = LINE(#)   Mapping is enabled. (See PEN.)
Selects Pen 1.
Puts the pen up at the HOME (0,0 GDU) position.
Updating of the relocatable origin is enabled (see PORG).
Sets relocatable origin to (0,0) GDUs.
Performs all Level 3 operations.

Level 3: Gets address space information needed by plotting package.
Sets the soft clip limits equal to the hard clip limits.
Calculates the GDUs based on the hard clip limits.
Sets UDUs equal to the GDUs.
Sets current units equal to GDUs.
Puts the "pen" up, without moving it.
Selects solid lines.
PEN=LINE   Mapping is not affected.
Selects standard character set.
Selects LORG(1) (left-justified labeling).
Sets labeling direction to left-to-right (LDIR (0)).
Clears any error conditions.
CSIZE is set to the device default value.
FXD(0,0) is executed.
Sets PDIR argument to zero degrees.
The "soft clipping" flag is set (turns clipping on).

SETAR

The SETAR (set aspect ratio) statement maintains the height and width ratio of the plotted data from one device to another.

Syntax

SETAR [(<aspect ratio>)]

Explanation

When the aspect ratio parameter is given, the hard clip limits are reset and a call to GPON(2C) is made. The GPON statement resets the GDUs so that the hard clip region will have the desired aspect ratio. The reset hard clip region is made as large as possible within the original hard clip limits.

If no aspect ratio parameter is given, a call to GPON(2A) is made (see GPON statement).

Examples:

```
10 SETAR
20 SETAR(2)
30 SETAR (Y/X)
```
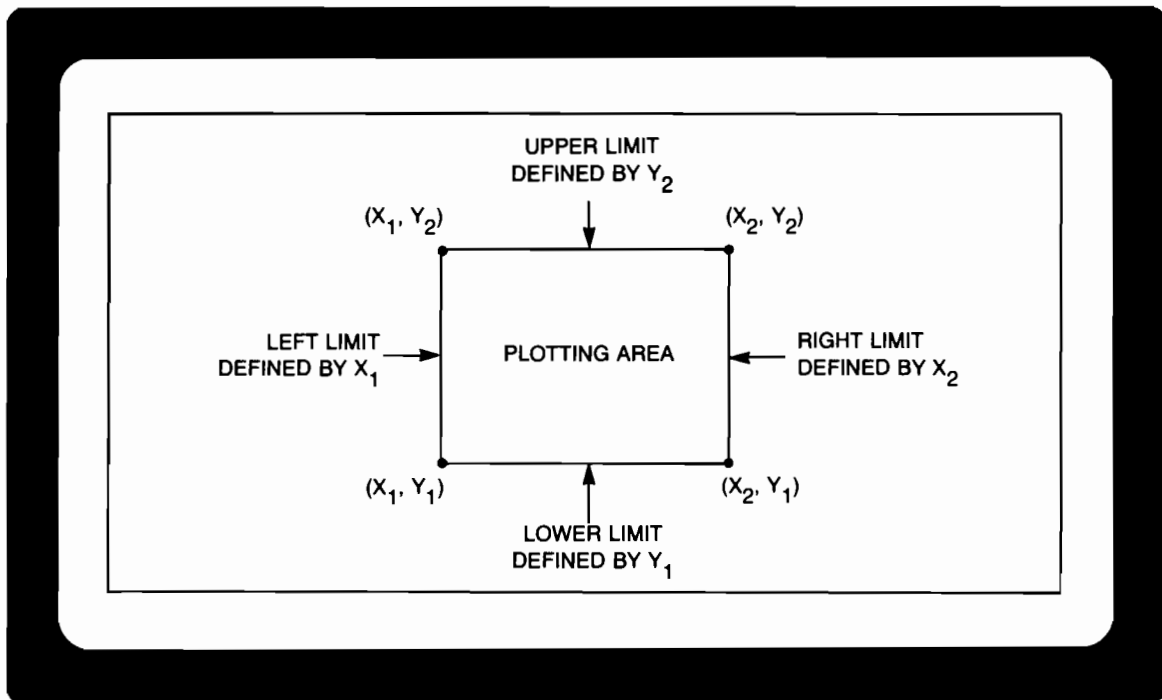
LIMIT

The LIMIT statement allows you to define the "plotting area" as a subset of the physical plotting area by setting the hard clip limits.

Syntax

LIMIT [(<X1>,<X2>,<Y1>,<Y2>)]

The first two parameters (X1 and X2) specify the left and right limits of the plotting area. The second two parameters (Y1 and Y2) specify the lower and upper limits of the plotting area.

Inside the figure:

UPPER LIMIT
DEFINED BY Y_2

(X_1, Y_2) (X_2, Y_2)

LEFT LIMIT
DEFINED BY X_1 → PLOTTING AREA ← RIGHT LIMIT
DEFINED BY X_2

(X_1, Y_1) (X_2, Y_1)

LOWER LIMIT
DEFINED BY Y_1

Explanation

In effect you are describing the size and location of a piece of paper or a specified area of the CRT to the computer. The computer is NOT allowed to move the pen outside the specified area without executing a new LIMIT statement with larger values for the parameters.

Because you are describing a physical piece of paper or plotting area to the computer, the units must be actual units of measure. For this syntax the units of measure are always in millimeters. The origin (point 0, 0) is at the lower-left physical limits of the plotting device.

If no parameters are supplied, AGL will execute the DIGITIZE statement twice to allow you to input the values with the graphics cursor. A message will appear in the message window giving the cursor location in millimeters (even though the label "UDUs"—user defined units— may be indicated in the message). (See the DIGITIZE statement.) The points may be entered in any order. The lower left point will define (X1, Y1) and the upper right (X2, Y2).

The LIMIT statement sets the plotting area to the user specification. It then calls GPON (level 2B) to reset the Hard Clip Limits on the plotting device, to redefine the GDUs, and to restore default conditions. The units for LIMIT are millimeters.

Example

```
10 PLOTR                          !Initialize AGL.
20 FRAME                          !Frame default hard clip region
30 LIMIT (15,15+150,10,10+100)    !Reset hard clip limits by
40                                !specifying a 150mm X 100mm
50                                !plotting area that is offset
60                                !from the plotter's lower left
70                                !physical bounds. The
80                                !specified offset for this
90                                !example is 10mm up and 15mm
100                               !to the right. Remember, units
110                               !are always in millimeters.
120 FRAME                         !Frame new hard clip region.
```
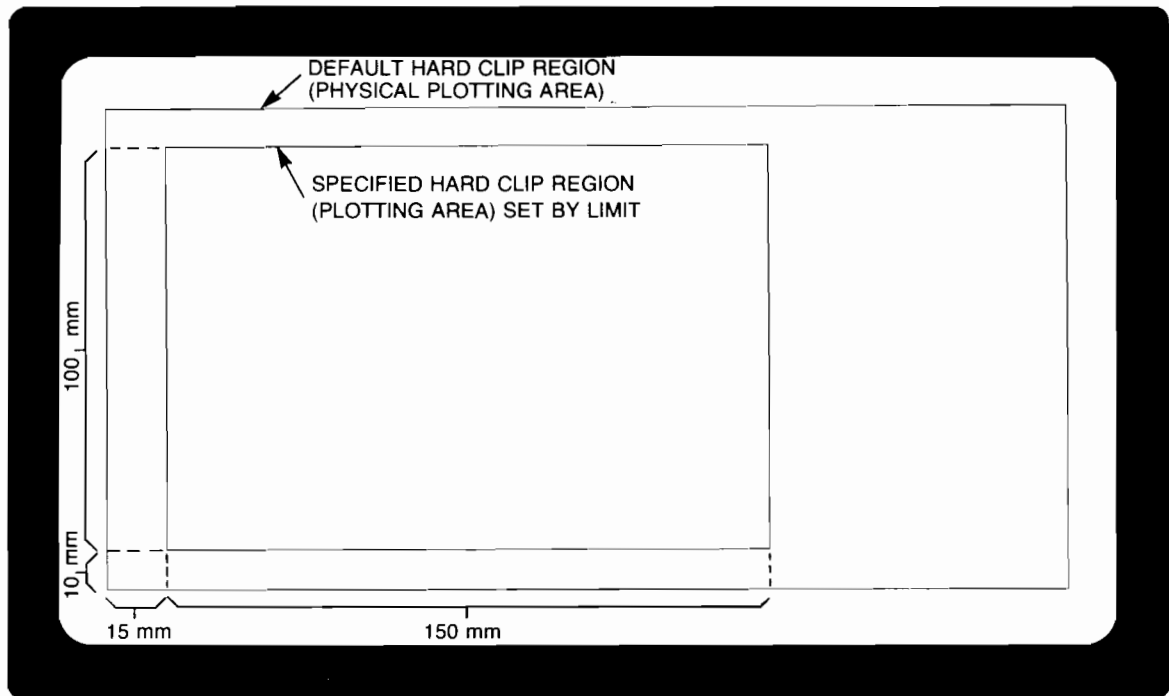

GCLR

The GCLR statement clears the plotted display on the terminal or plotting device and positions the pen at (0,0) (in GDUs).

Syntax

GCLR [(PAPERFEED)]

The optional paperfeed parameter is interpreted as follows:

If the parameter is greater than zero, it means form feed. The default paperfeed parameter is 1.

If the parameter is equal to zero, it means no action.

If the parameter is less than zero, it means previous page.

Explanation

The GPON statement should normally be used between plots, rather than GCLR, since GCLR does not restore the default conditions.

On fixed paper plotters (e.g. HP 9872) and on the CRT the paper advance parameter is ignored.

Example

```
10 REM...This example draws a grid pattern on the terminal
20 REM...display screen then clears the display screen.
30 PLOTR                    !Initialize AGL
40 LGRID (10,10,0,0,2,2)    !Draw grid pattern
50 LGRID (5,5,0,0,2,2)      !Draw grid pattern
60 LGRID (2.5,2.5,0,0,2,2)  !Draw grid pattern
70 GCLR                     !Clear graphics display
```
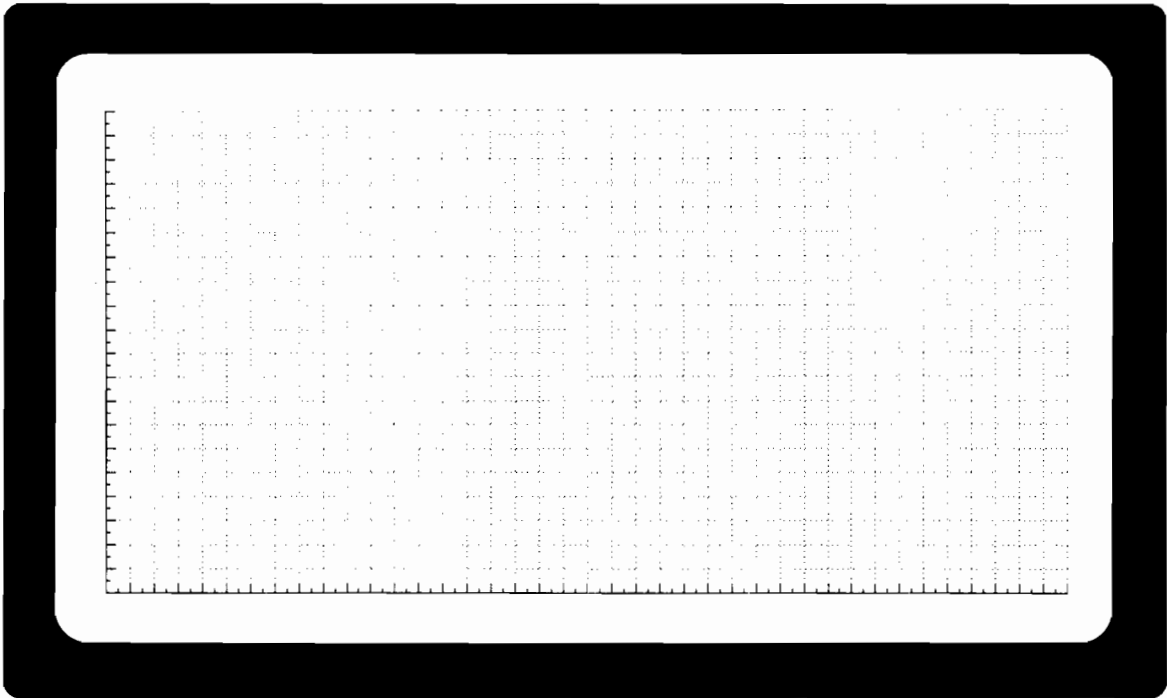

LOCATE

The LOCATE statement allows you to define an area within the hard clip limits onto which the SCALE statement maps or the SHOW statement fills. The LOCATE statement also defines and resets the default soft clip limits.

Syntax

LOCATE [(<X1>,<X2>,<Y1>,<Y2>)]

The first two parameters (X1,X2) specify the left and right boundary limits and the last two parameters (Y1,Y2) specify the lower and upper boundary limits.

All parameters are type REAL. The units are always in Graphics Display Units (GDUs) with the origin (GDU point 0,0) at the lower-left corner of the plotting area. The plotting area is defined by the physically imposed hard clip limits on the plotting device, or redefined by the hard clip limits set by the LIMIT statement.

The length of a GDU is equal to 1/100 (one percent) of the length of the short side of the plotting area. Therefore, the length of the short side is always 100 GDUs. The length of the long side of the plotting area is:

= (100 GDUs) * (long side/short side).

Explanation

With the LOCATE statement you are defining an equal or smaller area (e.g. a preprinted grid on graph paper) within the plotting area. It is into this area that the SCALE statement maps or the SHOW statement fills.

The LOCATE statement defines and resets the default soft clip limits. All lines drawn after executing the LOCATE statement will not extend beyond the soft clip limits. The pen can be positioned between the soft clip limits and the hard clip limits to draw labels, but plotted lines will not extend into this area. Soft clipping is enabled (see CLIPON).

The SCALE and SHOW statements assign user definable units of measure (UDUs) to the area specified by the LOCATE statement. The SCALE and SHOW statements act on this area in two very different ways as will be seen in the forthcoming discussion of those two instructions.

If LOCATE is to be used, it must be called before SCALE or SHOW. If no parameters are given, the DIGITIZE function will be used to allow you to input values using the graphics cursor.
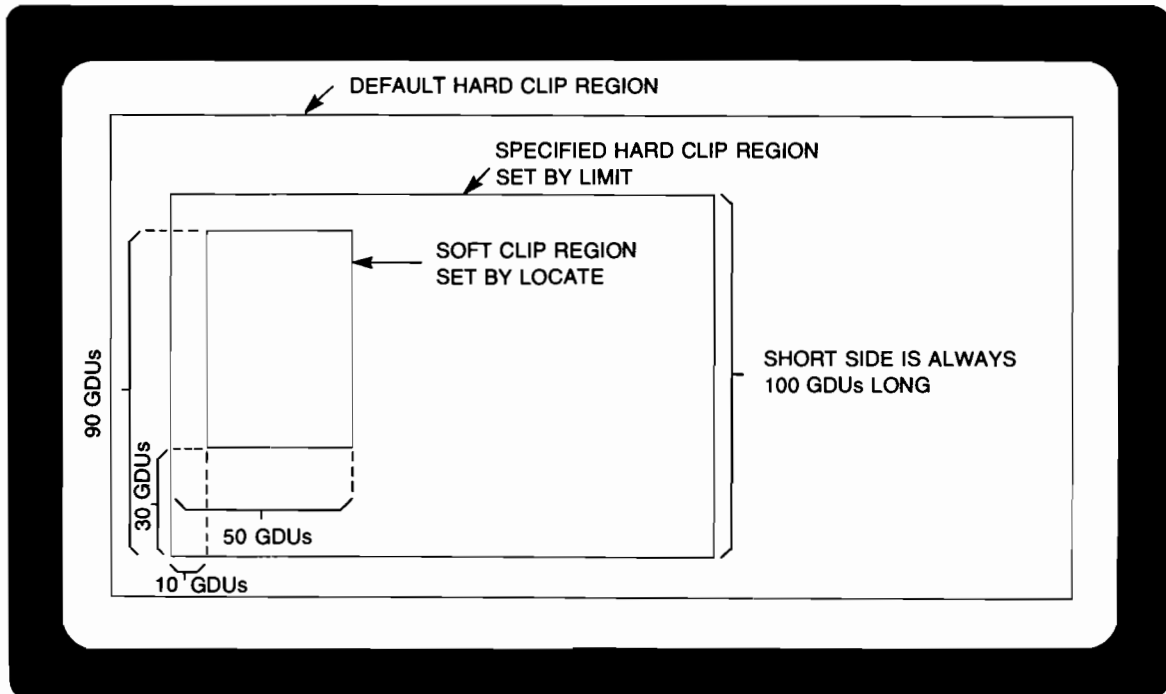
The default soft clip limits can be overridden by a call to the CLIP or CLIPOFF statements.

Example

```
10  PLOTR                        !Initialize AGL
20  FRAME                        !Frame default hard clip region
30  LIMIT (15,150,10,100)        !Redefine hard clip limits
40  FRAME                        !Frame new hard clip region
50  LOCATE (10,50,30,90)         !Set soft clip limits
60  FRAME                        !Frame soft clip region
```


MARGIN

The MARGIN statement specifies the LOCATE rectangle relative to the hard clip boundaries. This is done by the user defining the number of character-cell spacings or GDUs allowed between the hard and soft clip limits.

Syntax

MARGIN (left, right, bottom, top [,units])

All parameters (left, right, bottom, top, units) are type REAL.

The units parameter specifies the number of character-cell spacings or GDUs which will be allowed in a direction "inward" from the appropriate hard-clip boundary.

When units is 0 (default), the distance is measured in character-cells. When units is not 0, the distance is measured in GDUs (signs not significant).

For the left, right, bottom, and top parameters, values greater than zero (>0) imply upright characters (i.e. <bottom> and <top> are in linefeeds, <left> and <right> are spaces). Values less than zero (<0) imply sideways characters (i.e. <bottom> and <top> are in spaces <left> and <right> are in linefeeds).
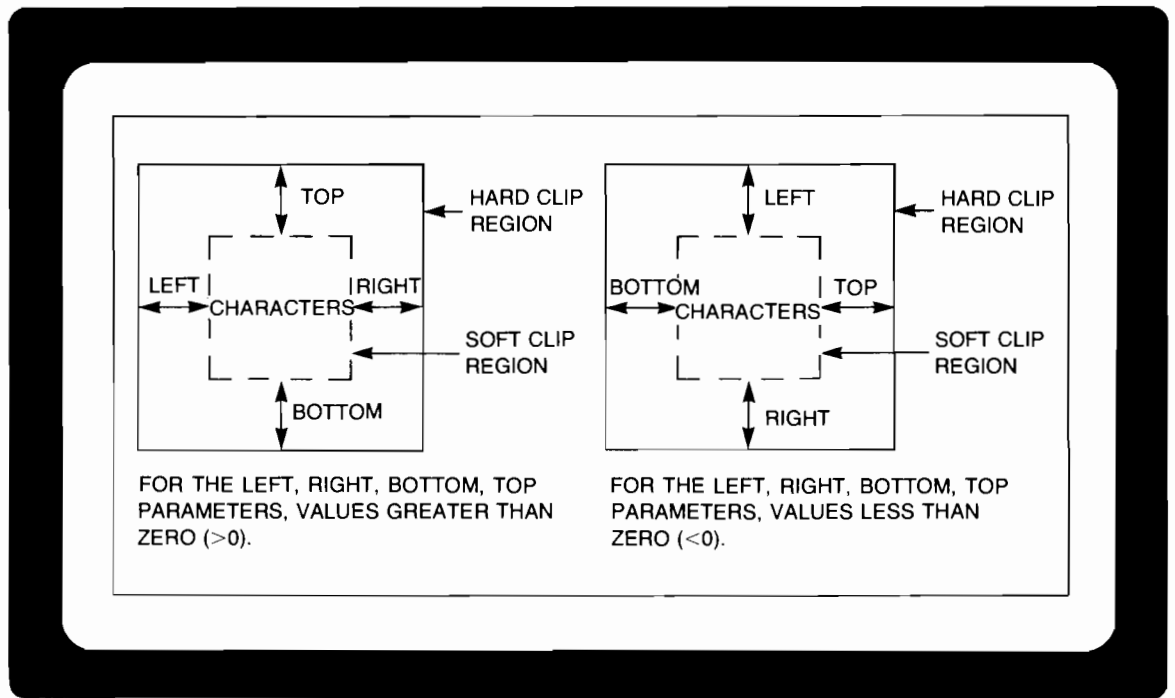
Example

```
10 PLOTR                    !Initialize AGL
20 LIMIT (0,200,0,100)      !Set hard clip limits
30 FRAME                    !Frame hard clip region
40 MARGIN (10,10,10,10)     !Define soft clip region
50 FRAME                    !Frame soft clip region
```

HARD CLIP REGION
DEFINED BY LIMIT

SOFT CLIP REGION
DEFINED BY MARGIN
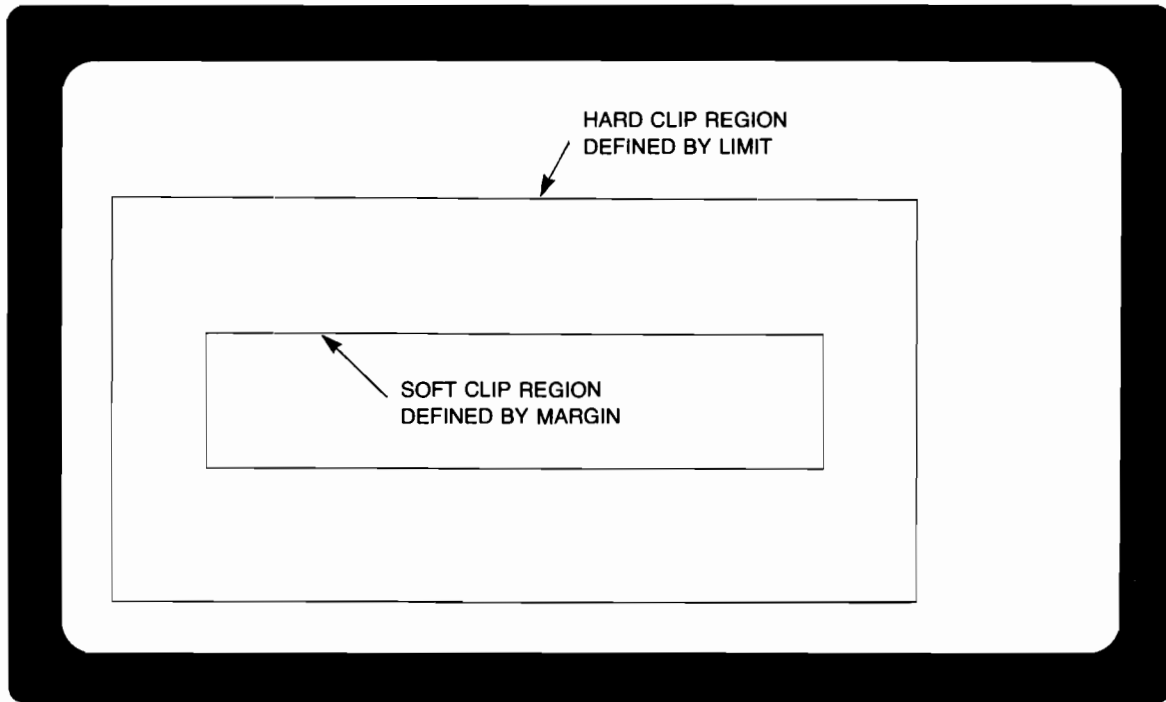
SCALE

The SCALE statement defines minimum and maximum values of X and Y for a user space. This user space is to be mapped exactly onto the plotter space defined in the LOCATE statement. Therefore, the SCALE statement allows you to specify your own units for plotting.

Syntax

SCALE (<X1>,<X2>,<Y1>,<Y2>)

The values used for X1,X2 and Y1,Y2 are in user units. The SCALE statement sets current units to User Units.

The first two parameters specify the values represented by the left and right boundary of the area specified by the LOCATE statement. The last two parameters specify the values represented by the lower and upper boundary in a similar manner.

For example, the first two parameters of the SCALE statement could specify the left edge of the plotting area as -20 and the right edge as 30. This has the effect of dividing the horizontal plotting distance into 50 units $(30 - (-20) = 50)$. The last two parameters could specify different values and therefore a different scale or units for the vertical direction. These units can be used to represent distance, volume, time or whatever units your specific problem requires.

Typically the units and scale factors will be different for the X and Y directions. The user space is stretched or shrunk independently in the X and the Y direction to fit the plotting area. Note that this is not the case with the SHOW statement.

If LOCATE is to be used, it must be called prior to SCALE.

Example

If you wanted to plot the average annual rainfall at a weather station for a 10 year period, the SCALE statement might look like this:

SCALE 1966, 1976, 0, 20

```
10 PLOTR                          !Initialize AGL.
20 FRAME                          !Frame default hard clip region.
30 LOCATE (40,100,20,70)          !Define soft clip region.
40 SCALE (1966,1976,0,20)         !Define values of limits of
45                                !soft clip region.
50 AXES (1/12,1,1966,0,12,10)     !Draw Axes.
60 MOVE (1966,0)                  !Move to start of plot
65                                !(YR 1966, 0 inch).
70 FOR M=1966 TO 1976 STEP 1/6    !For...Next generates random
80 PLOT (M,RND*2+7+2*SIN(M))      !data for this example.
90 NEXT M                         !Do it again for next point.
```
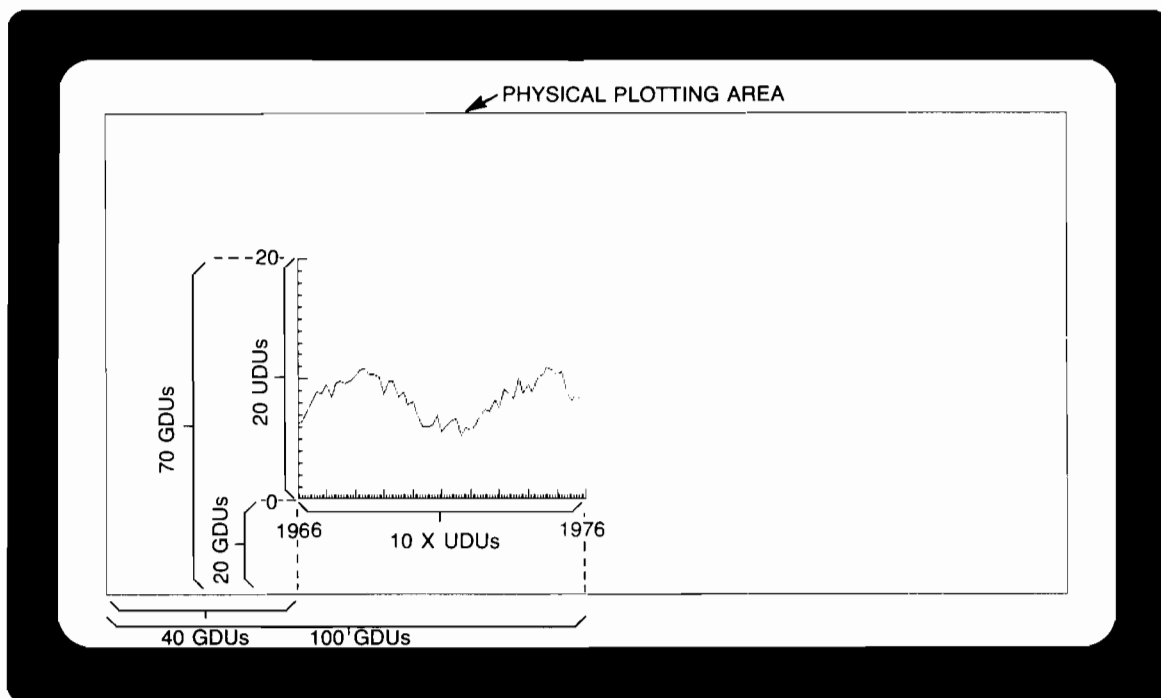

SHOW

The SHOW statement defines an area that is stretched or shrunk proportionately in the X and Y directions so that the specified area will fit in the plotting area defined by default or by the LOCATE statement.

Syntax

SHOW (<X1>,<X2>,<Y1>,<Y2>)

The first two parameters specify the acceptable upper and lower bounds for the X direction, while the last two parameters specify the acceptable bounds for the Y direction.

The units are always in user units. SHOW sets current units to user units (calls SETUU).

Explanation

The aspect ratio of the units will be forced to be equal to one where the left edge represents year 1966 and the right edge represents 1976. Rainfall would be plotted in the Y direction in volume units (e.g. inches). This allows the data to be plotted in years and volume units (e.g. 1976,7) directly on the plotter (1) by the SHOW statement. That is, one length of measure in the X direction will equal one length of measure in the Y direction, thus eliminating distortions.

The SHOW statement would not normally be used when you want to plot in two different units of measure, as in the previous example for the SCALE statement where inches of rainfall and years were used as units of measure. But, SHOW should be used when you want to plot something like a map where one mile along the X axes should equal one mile along the Y axes.

If LOCATE is to be used, it must be called prior to SHOW.

Example

```
10 PLOTR          !Initialize AGL
20 FRAME          !Frame default hard clip region
30 LOCATE (10,45,20,60)   !Define soft clip limits
40 FRAME          !Frame soft clip region
50 SHOW (20,45,20,60)     !Specify user units, using SHOW
60 CLIP (20,45,20,60)     !Redefine soft clip limits
70 FRAME          !Frame redefined soft clip limits
```
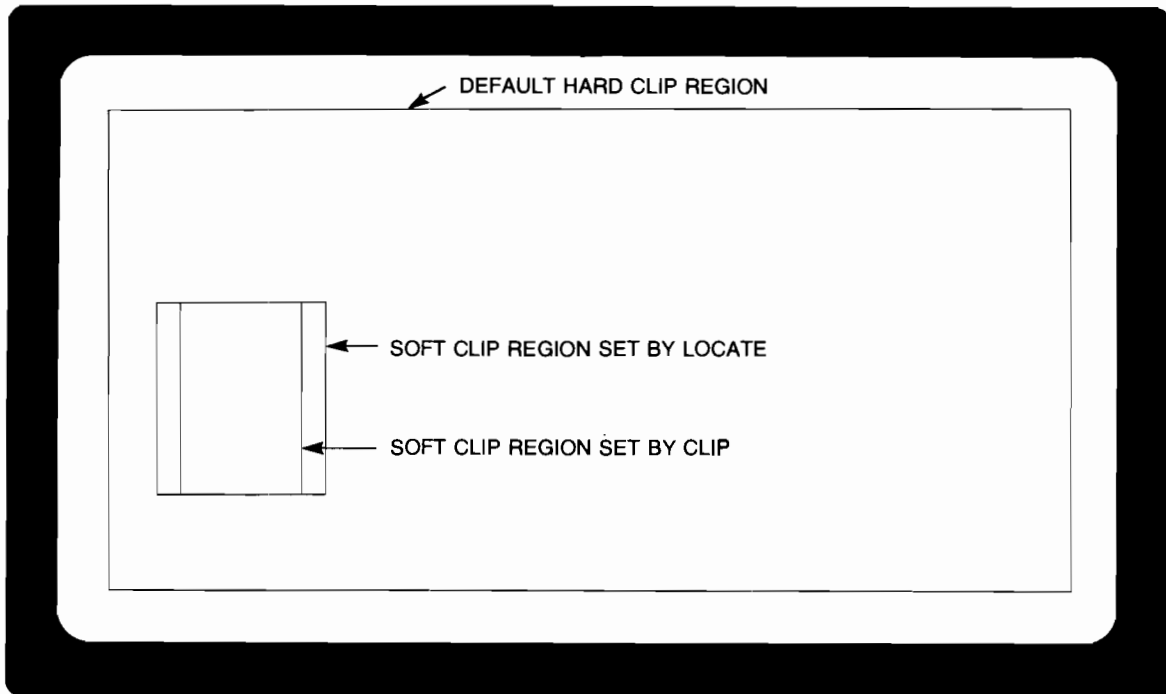
MSCALE

The MSCALE statement specifies that the current user units are millimeters. Also, MSCALE causes the origin (0,0) to be offset from the lower left corner of the hard clip region.

Syntax

MSCALE (<X reference>, <Y reference>)

The X reference and Y reference values must be given in millimeters. MSCALE uses these reference values to offset the origin from the hard clip corner.

MSCALE offsets the origin from point (X1,Y1) to point (X,Y). Refer to figure below.

Example

```
10 PLOTR                    !Initialize AGL
20 FRAME                    !Frame default hard clip region
30 MSCALE (20,50)           !Offset point (0,0) up 50mm and
35                          !right 20mm
40 CLIP (0,100,0,10)        !Define soft clip limits
50 FRAME                    !Frame soft clip region
60 AXES (2,10,0,10,5,10,5)  !Draw tic marks (axes) on scale
```
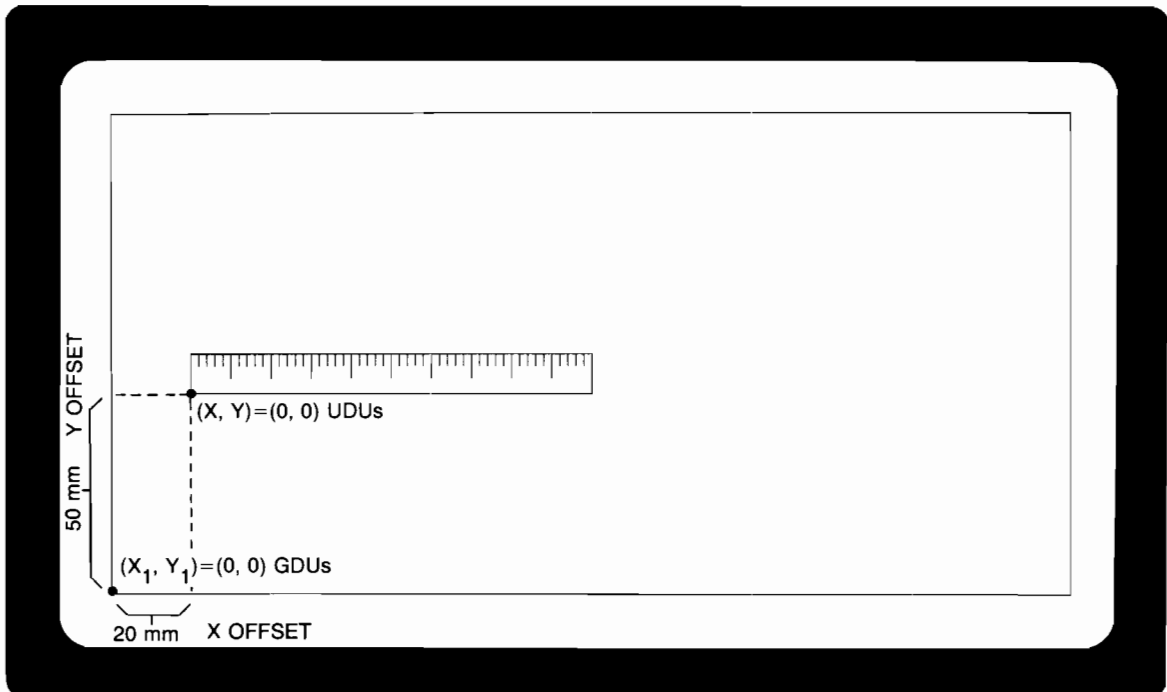
CLIP

The CLIP statement redefines the soft clip limits. This allows the soft clip limits to be moved from their default location set by the PLOTR, LIMIT, or LOCATE statements.

Syntax

CLIP (X1,X2,Y1,Y2)

The parameters (X1,X2,Y1,Y2) are interpreted according to the current units of measure.

The first two parameters specify left and right boundaries of the soft clip region. The last two parameters specify the lower and upper boundaries of the soft clip region.

Explanation

If a line is plotted from inside the soft clip region to a point outside of the soft clip region, the line will be cut off at the soft clip boundary.

If part of the soft clip region falls outside the hard clip boundaries, the soft clip boundaries are redefined to correspond to the intersection of the two regions (see shaded area in figure below).

If the soft clip region falls outside of the hard clip region, and there is no intersection between them, an error will occur.
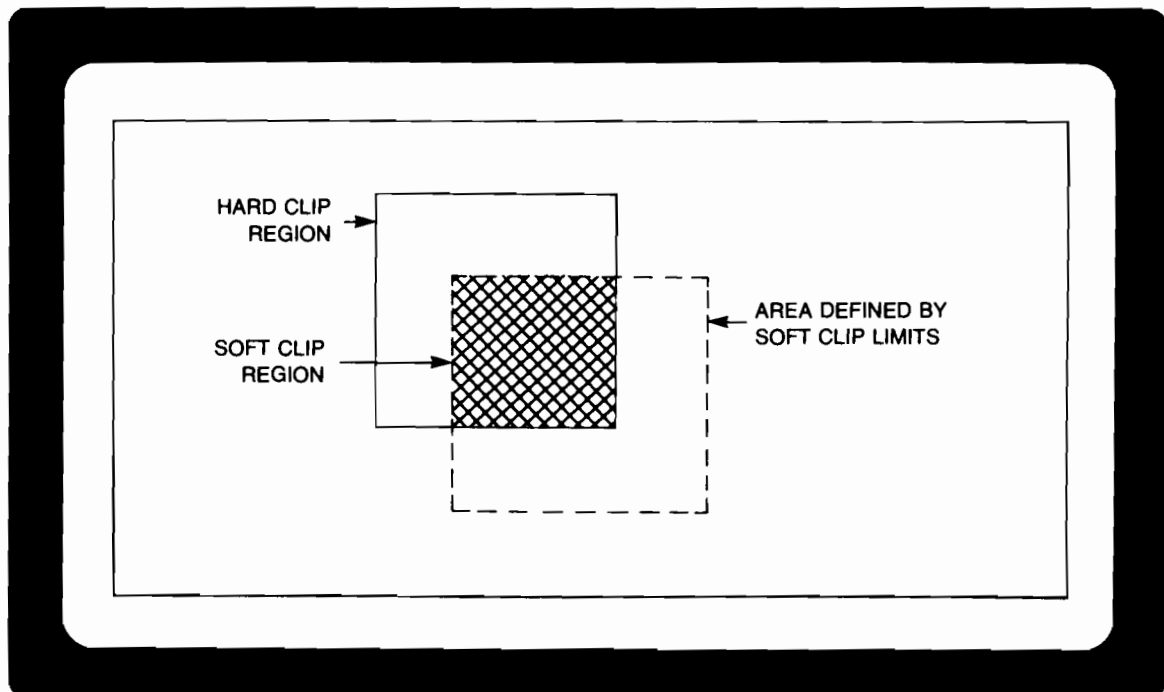
CLIPOFF/CLIPON

CLIPOFF

The CLIPOFF statement sets the soft clip limits equal to the hard clip limits. This allows you to draw lines anywhere in the plotting area defined by the LIMIT statement. The original soft clip limit values are retained for future use.

CLIPON

The CLIPON statement restores the soft clip limits.

Clipping is also restored by executing PLOTR, LIMIT, LOCATE, GPON(3), or CLIP. SCALE and SHOW do not affect clipping.

SETGU/SETUU

Several AGL statements (as indicated) interpret their parameters using the units currently selected. These two commands provide the means to change the units mode.

Syntax

SETGU

or

SETUU

Explanation

SETGU selects graphics display units (GDUs) and SETUU selects user-defined units (UDUs).

Graphic Display Units (GDUs) are primarily intended as a device-independent coordinate system used for positioning items on the plotting area. Simple programs can leave the current units mode set to UDU.

Axis and Labeling Functions

The axes and labeling functions use the clipping boundaries to define the end points of the axes. The soft clipping limit values are used for X1 and X2 (end points of x axis). This allows the Y limits to be placed to position the X axis conveniently. The X axis can even be placed outside of the current clipping area. This allows the placement of multiple X axes outside of the region of interest.

The Y tic marks are placed symmetrically about the X-axis and are not clipped. Grid lines extend from one plotting boundary to the other (Y1, Y2 are soft clip values when a UDUs, or hard clip limits when in GDUs).

XAXIS

XAXIS [(<x tic spacing>)[<x origin>,<y origin>
[,<x major count>[,<tic size>]]])]

where "x tic spacing" is interpreted in the current units mode. The sign is ignored. Default of 0=> no tics.

The "origins" are in current units. This allows proper placement of axis and tics. The "y origin" specifies the placement of the X axis and the major Y tics while the "x origin" specifies the placement of major X tics and grid lines. The default origin is 0,0.

The "x major count" is a unitless integer value which specifies the intervals between "major" tic marks as **numbers of** "minor" tics. The signs are ignored. The default is 1 => all major tics. 0 => all minor tics.

The "tic size" specifies the length of the minor tics (end to end; tics are symmetric about the axis). Signs are ignored for minor tics. If the sign is positive the major tics are twice as long as minor tics. If the parameter is negative, the major tics become grid lines which extend from one LOCATE boundary to the other. The default tic size is +2 GDUs. The parameter is type REAL interpreted in GDUs. XAXIS generates an X axis at y = y origin. Tic marks are positioned along the axis such that a major tic mark falls on the origin (whether visible or not); the origin may lie outside the current soft clipping region. Tic marks may or may not coincide with the edge of the clipping boundary.

A Graphics Language (AGL)

Examples

```
10 PLOTR
20 FRAME
30 XAXIS (2,10,50,5,-2)
```

x tic spacing   x origin   y origin   x major count   tic size
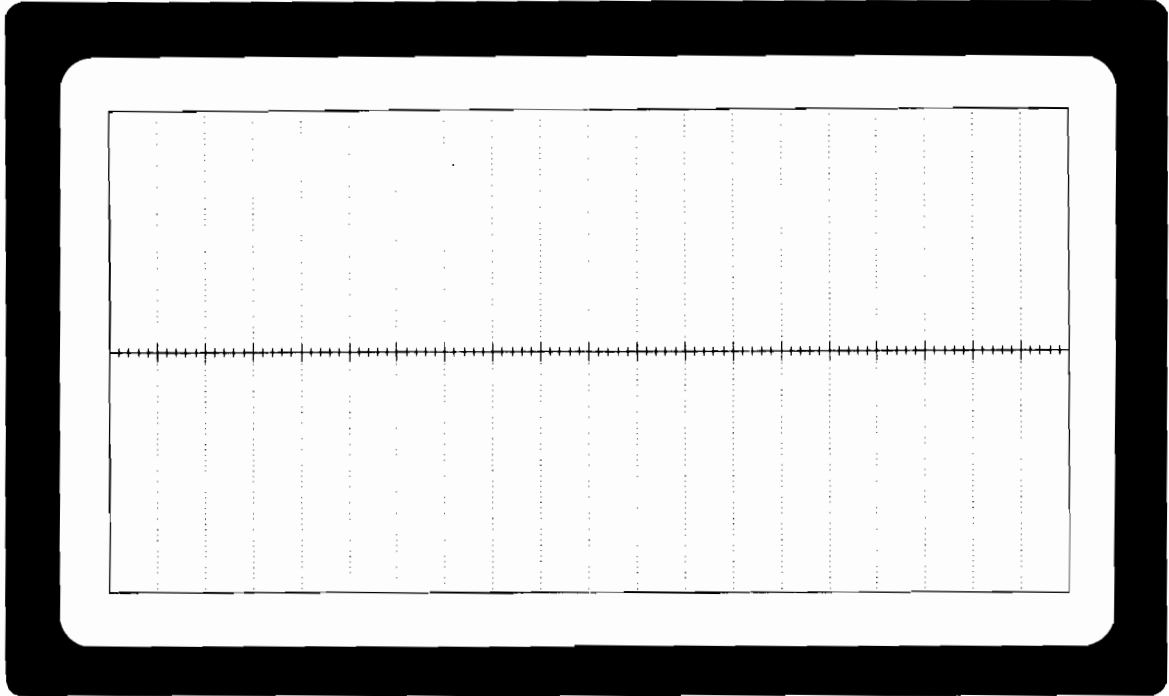

```
10 PLOTR
20 FRAME
30 XAXIS (2,10,50,5,2)
```
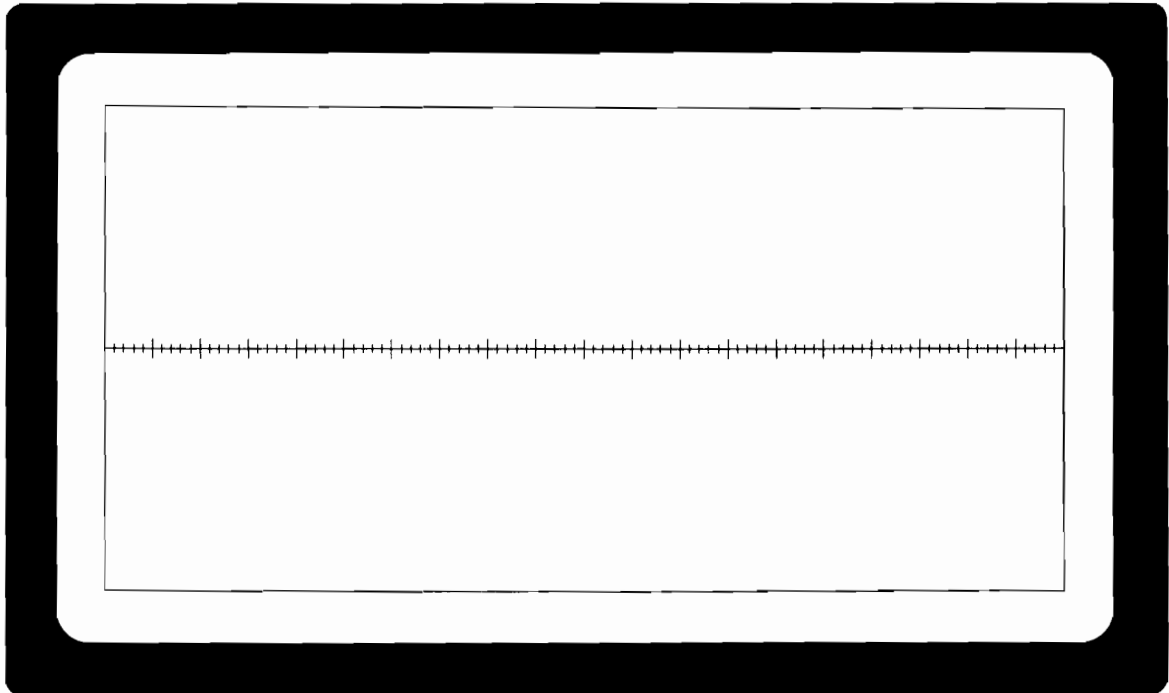
YAXIS

YAXIS [(<tic spacing>)[,<x origin>,<y origin>
[,<y major count>[,<tic size>]]])]

where all YAXIS parameters are defined as in XAXIS, except in the Y (vertical) direction.

YAXIS generates a Y axis at x = x origin. All parameters are interpreted as in XAXIS except that now all concern the drawing of a Y axis.

It is useful to note that a call to XAXIS followed by a call to YAXIS will generate a pair of perpendicular axes which specify a Cartesian Coordinate System in the current units mode.

Example:

```
10 PLOTR
20 FRAME
30 YAXIS (2,100,40,5,2)
```
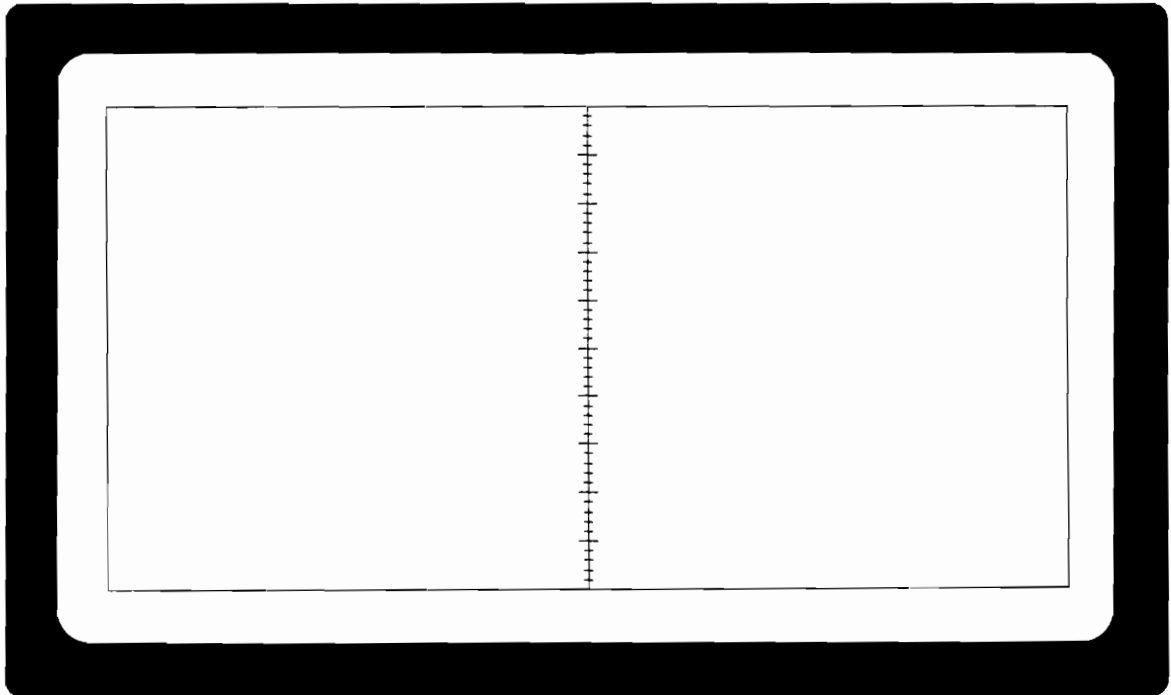
LXAXIS

LXAXIS [(<x tic spacing>[,<x origin>,<y origin>
[,<x major count>[,<tic size>[,<label loc>]]]])]

where the "tic spacing" is used to determine the orientation of labels: + for perpendicular to the corresponding axis; − for parallel. Otherwise, the tic spacing, origins, major count, and the tic size are all interpreted as in XAXIS. The "label loc" parameter is used to place the label relative to the axis. Use one of four label locations:

0: label below and outside the current vector clipping limits.

1: label immediately below the axis about two GDUs from the corresponding tic.

2: label immediately above the axis about two GDUs from the corresponding tic.

3: label above and outside the current vector clipping limits.

LXAXIS draws and labels the X axis in current units mode.

Example:

```
10 PLOTR
15 FRAME
20 SETUU
30 LOCATE (10,50,10,50)
40 FRAME
50 LXAXIS (2,10,30,5,2,1)
```
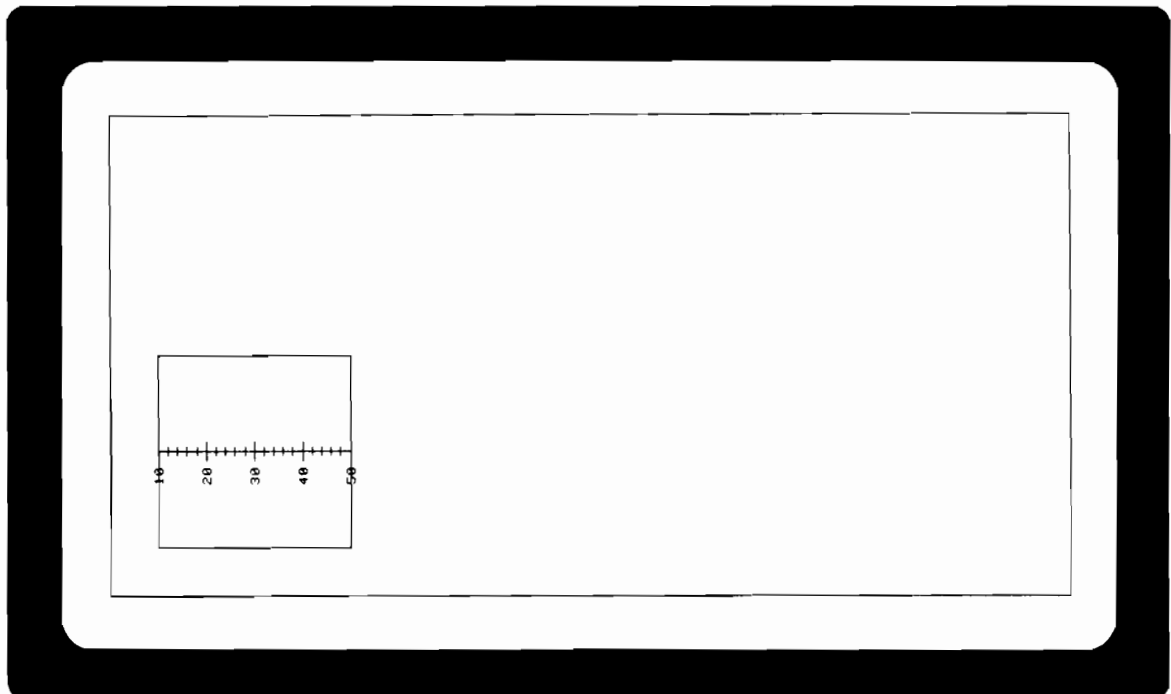
LYAXIS

LYAXIS [(<y tic spacing>[,<x origin>,<y origin>
      [,<y major count>[,<tic size>[,<label loc>]]]])]

where LYAXIS parameters are the same as LXAXIS, except in the Y (vertical) direction and the label locations. The label locations are as follows:

0: label on the left side, just outside of the current vector clipping limits.

1: label to the immediate left of the axis, about two GDUs from the corresponding tic.

2: label to the immediate right of the axis, about two GDUs from the corresponding tic.

3: label on the right side, just outside of the current vector clipping limits.

LYAXIS draws and labels the Y-axis.

Example:

```
10 PLOTR
15 FRAME
20 SETUU
30 LOCATE (10,50,10,50)
40 FRAME
50 LYAXIS (2,30,10,5,2,1)
```
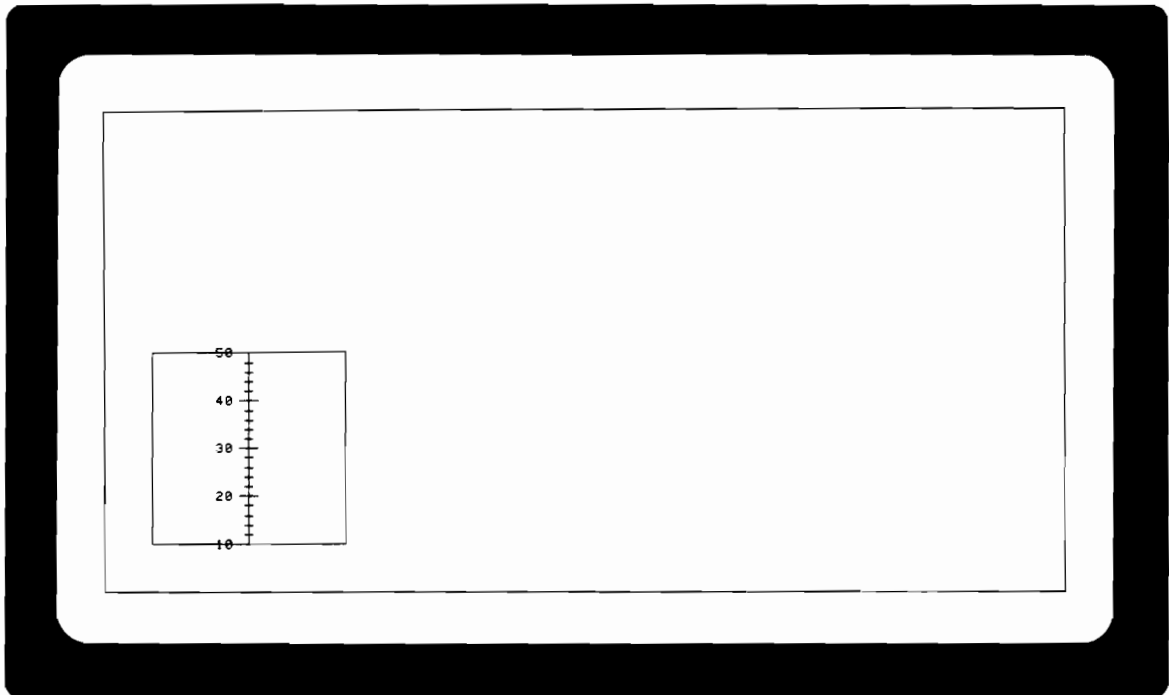
AXES

AXES[(<x tic spacing>,<y tic spacing>
        [, <x origin>, <y origin>
        [, <x major count>, <y major count>
        [, <minor-tic size>]]])]

where "tic spacings" are REAL values in current units. The signs are ignored. Default or
0 = > no tics.

The origins are also REAL values in current units. Default is (0,0).

The X and Y "major counts" are unitless integer values which specify the intervals between
"major" tic-marks as numbers of "minor" tics. The signs are ignored. Default is 1 (i.e. all
"major tics").

The "tic size" specifies the length of the minor tics (end-to-end; tics are symmetric about the
axes). Major tics are always twice as long as minor tics. The size is type REAL, in GDUs.
Default size is approximately 2 GDUs.

AXES generates an X-axis at y=y origin, and a Y-axis at x=x origin, assuming that these
values are within (or on) the current user-units clipping limit. The axes will extend across
that same clipping region. Tic marks will be clipped at the clipping limit.

Tic marks are positioned along each axis such that a major tic mark falls on the origin
(whether visible or not). Tic marks may or may not coincide with the edge of the clipping
boundary.

Axes and tic marks will be drawn using solid lines and the current "pen" (however, note that
selecting solid lines on some displays is equivalent to selecting pen 1).

Example:

The following two calls would generate the axes shown within the plotting region specified by LOCATE (or the default plotting region). Labels in parentheses are for reference only.

```
>SCALE (-5., 20., .8, 1.5)
>AXES (2.0,  .1,  0.0, 1.0, 5, 2)
```
```
         tics    origin   major interval
```

```
10 PLOTR
20 FRAME
30 LOCATE (20,80,20,80)
40 SCALE (-5,20,.2,5)
50 FRAME
60 AXES (2,.5,0,1,5,2)
```
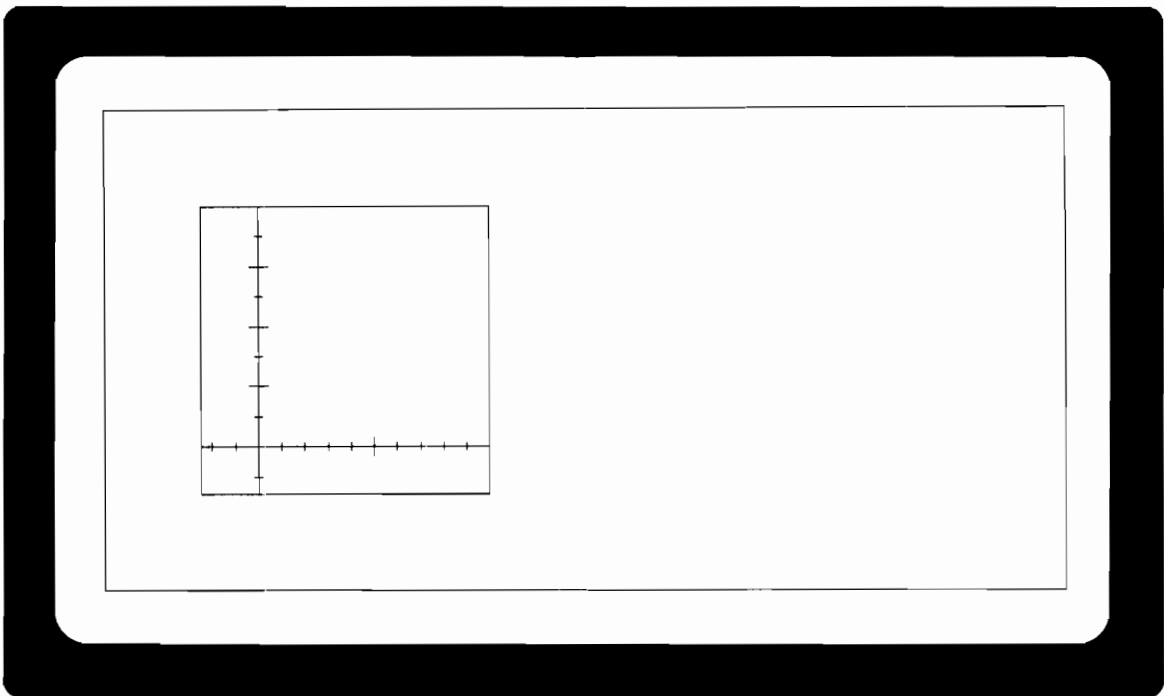
LAXES

LAXES [(<x tic spacing>, <y tic spacing>
   [, <x origin>, <y origin>
    [, <x major count><y major count>
     [, <minor tic size>]]])]

The LAXES parameters are the same as the AXES functions, except the signs of the tic spacings determine the orientation of the labels: + for perpendicular to the corresponding axis, and − for parallel.

Each major tic is labeled. All tics are considered major tics if no "major count" is specified. Labels perpendicular to the X-axis will be lettered bottom-to-top (right-justified along the bottom border). Labels are placed outside the clipping limit, and are limited to a maximum of seven visible characters.

Example:

The following three calls would generate the axes shown within the plotting region specified by LOCATE (not the default plotting region). Labels are outside the plotting region.

```
10  PLOTR
20  FRAME
30  LOCATE (20,80,20,80)
40  SCALE (-5,20,.2,5)
50  FRAME
60  LAXES (2,.5,0,1,5,2)
```
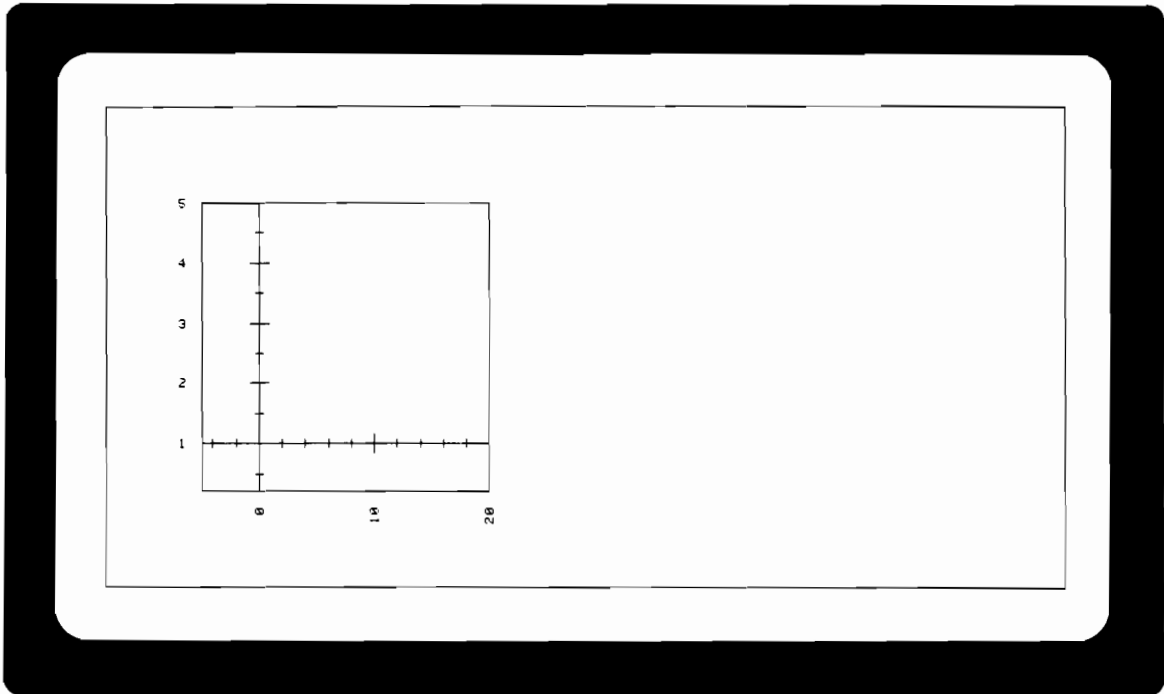
GRID

GRID [(<x tic spacing>, <y tic spacing>
    [, <x origin>, <y origin>
      [, <x major count>, <y major count>
        [, <minor tic size>]]])]

GRID may be used as an alternative to AXES when a full grid is desired. Heavy lines (LINE(0)) are used for the axes. Lighter lines are used for minor divisions. All the lines extend throughout the current soft clipping region. GRID function parameters are the same as AXES, except that light cross-lines replace the tic-length indications.

Example:

```
10 PLOTR
20 SCALE (-8,8,-8,8)
30 GRID (1,1,0,0,4,4,1)
40 FRAME
```
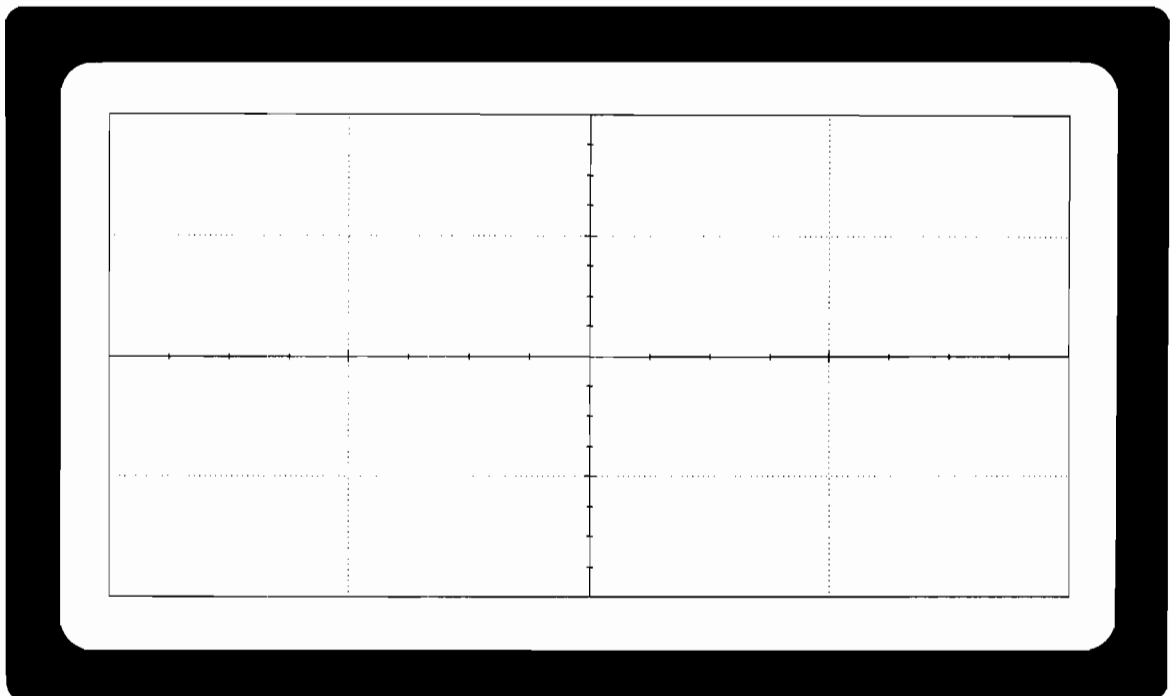
LGRID

LGRID [(<x tic spacing>, <y tic spacing>
[, <x origin>, <y origin>
[, <x major count>, <y major count>
[,minor tic size]]])]

LGRID draws a labeled grid and its characteristics are the same as for GRID, except that as in LAXES the signs of X and Y tic spacings are used to specify the label orientations. LGRID uses units in the number range set up by the SCALE statement.

FRAME

FRAME

The FRAME function draws a "box" around the current vector-clipping region using the current pen and linetype characteristics.

Example:

```
10 PLOTR
20 FRAME
```
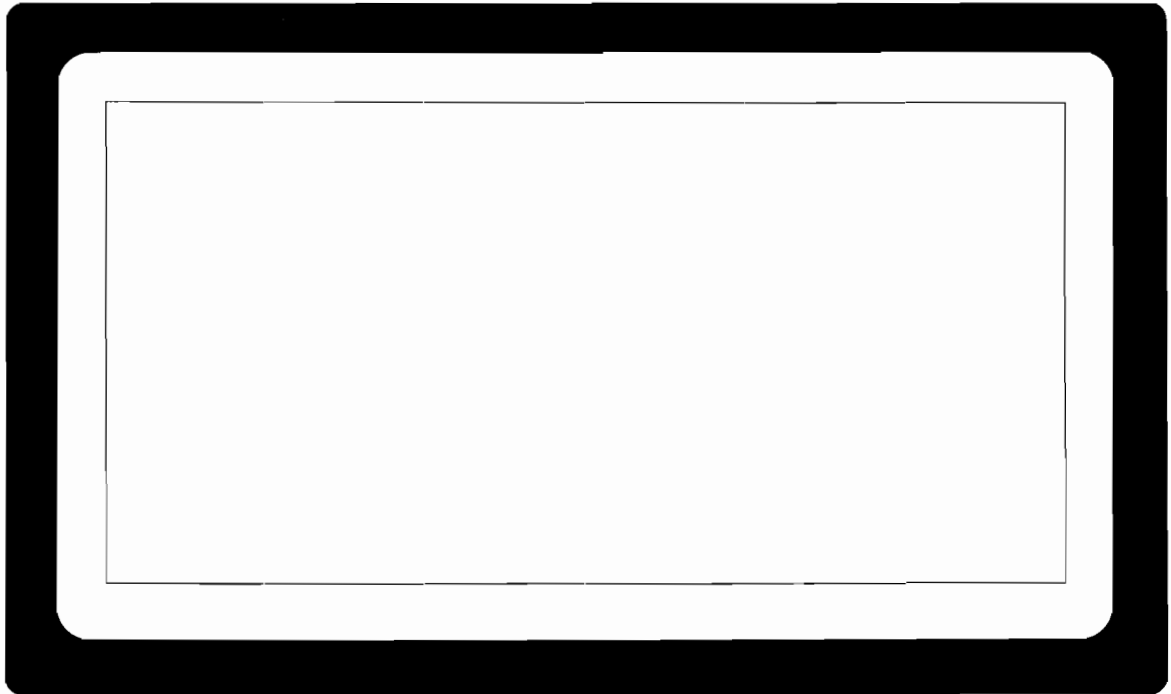
FXD

FXD [(X digits [,[Y digits]])]

where "digits" is a positive integer. FXD selects the Axis-labeling format. Using FXD(n) causes the labels generated by LAXES and LGRID to be printed in F7.n format. Leading zeros are suppressed. If the number will not fit in the space to the left of the decimal point, the decimal will be moved to the right. When that is insufficient or the number underflows, F7.0 format will be used.

Note that the F7.0 format does not allow for more than one significant digit if a number overflows the specification. If you used 1.E+07 for example, only one digit after the decimal point would be displayed.

The default value for X digits is 0, the default value for Y digits is the X digits value. The maximum number of digits that can be used is 5.

LORG

LORG (mode)

where mode is a number in the range 0-9. If the number is not selected. LORG defaults to 1.

LORG selects the label-origin position and determines how subsequent labels will be placed relative to where the pen is when each label is received. Labels can be automatically right or left justified, or centered about a specified point. LORG 0-9 indicates the origin (justification and base line) for characters with respect to the current pen position. Labels are output using the PRINT #0 statement.

A carriage return (CR) causes the pen to be returned to its original position, and a linefeed (LF) causes the pen position to move in the expected direction.

If a label is to be left justified, the current pen position is the left margin. Center causes the label to be centered on the pen position. Right justify selects the pen position as the right margin. Bottom, middle, and top select the base line for the labeling string.

In the following illustration, each number shows the initial position of the pen relative to the label when LORG selects a number from 0 through 9. For example, if the label was to be right justified and set with a base line on top of the normal character position, the number "9" would be used.

Label positioning is done by the plotting device. When centering or right justification is used, the labeling string is buffered (stored) until all of the characters in the string have been received. The string end is detected and displayed by a CR or LF. The maximum length of a string when centering or right justifying is 80 characters (blanks are included but CR or LF is not). In all cases, data written beyond the edge of the screen is lost. There is no automatic RETURN when the screen boundary is reached.

CHARACTER CELL

LDIR

LDIR (<angle in radians>)
or
LDIR (<run>, <rise>)

LDIR sets the lettering direction of labels. In general each device will use the available angle which is closest to the requested angle. Angles are measured in radians (1 radian = 57 degrees).

Since many devices cannot letter at angles other than multiples of 90 degrees (pi/2 radians), programs which use other angles may not run acceptably on all devices.

For angles in the range −90 degrees (−pi/2 radians) to +90 degrees (pi/2 radians), (<rise>/<run>) = TAN (<angle>).

Parameter interpretation depends on number of parameters. If only one parameter is supplied, it is assumed to be the counter-clockwise angle from the normal lettering direction (0 => towards 3 o'clock, 90 degrees => towards 12 o'clock, etc).

If two parameters are supplied, the angle used is that of an incremental vector plotted with arguments X increments = run, and Y increment = rise with X and Y in current units.

Example:

```
10 PLOTR                              !Initialize AGL
20 FRAME                              !Frame Plotting Area
30 SCALE (-10,10,-10,10)             !Specify User Units
40 LORG (2)                           !Center on left end
50 FOR Radian=0 TO 6.28 STEP 1.57    !Loop for radians
60 LDIR (Radian)                      !Specify a direction
70 MOVE (0,0)                         !Move to center point
80 PRINT #0;"------LDIR";Radian       !Label plotting area
90 NEXT Radian                        !Do it again until complete
```
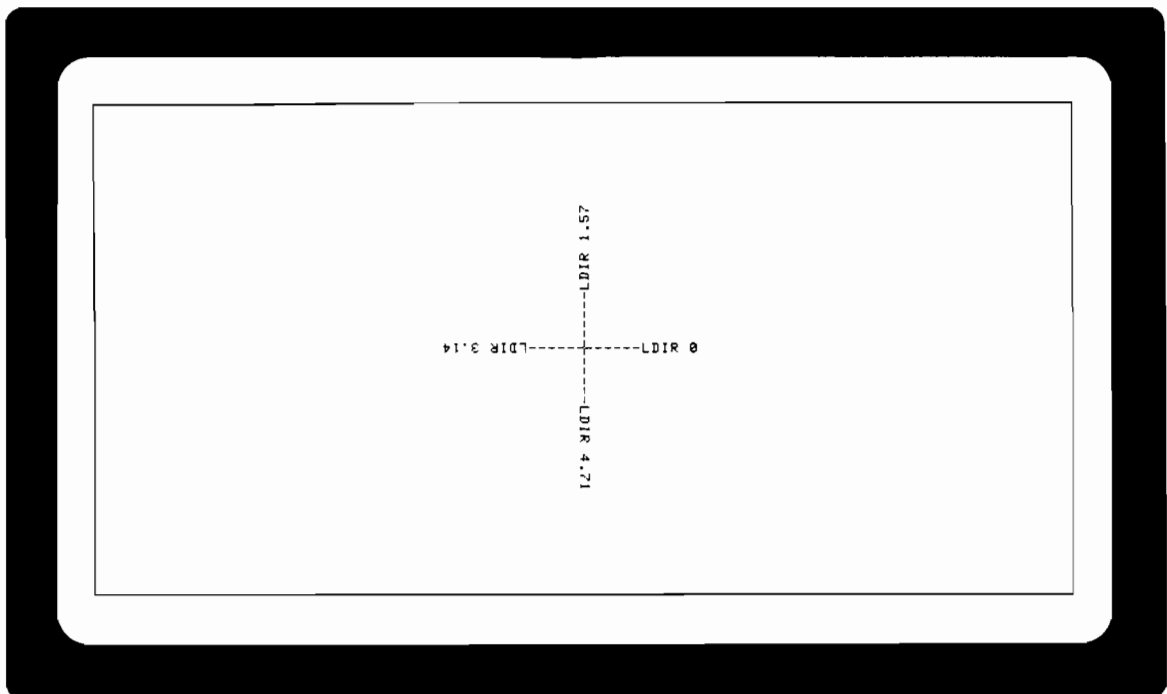
CSIZE

CSIZE (<height> [,<ratio> [,<slant in radians>]])

The character size (CSIZE) function specifies the size and aspect ratio of the alphanumerics or symbols to be drawn for labels. In general, if the requested size is not possible, the next smaller size (if any) should be used.

"Height" is the character cell height in GDUs. Default height is device-dependent.

"Ratio" is the aspect ratio of the character cell, defined as width/height. Default varies with device.

"Slant" specifies the clockwise rotation (in radians) of the character relative to its normal position. Horizontal lines are not affected. The character cell becomes a parallelogram (i.e. 20 degrees of slant gives an italic like character). Units are current angle-specifying units for the system. Default slant is 0 degrees.

Note that many devices are incapable of drawing characters with arbitrary sizes and aspect ratios. The minimum character height may be of the order of 4 or 5 GDUs, aspect ratio may be fixed (e.g. 5x7 or 7x9 dot matrix), and slanting may not be possible at all. You may request size, etc., with this command, but you should then use the CSIZE call to determine the actual sizes for purposes such as computing margins.

Width of character = <height> * <ratio>

Example:

```
10 PLOTR                        !Initialize AGL
15 FRAME                        !Frame Plotting Area
20 SCALE (-.5,10,-10,0)         !Specify User Units
24 !
25 !*****Label Plotting Area With Two Columns Of Example
26 !*****Character Sizes.
30 FOR Column=1 TO 2            !Specify One of Two Columns
35 FOR Char=1 TO 10             !Specify one of 10 Char Sizes
40 CSIZE (Char,Column)          !Specify Char Size and Aspect
45                              !Ratio
50 MOVE ((Column-.5)*4,-Char)   !Move to Label Position.
60 PRINT #0;"CSIZE";Char        !Label Plotting Area
70 NEXT Char                    !Get Next Size.
80 NEXT Column                  !Get Next Column.
```

```
CSIZE 1                CSIZE 1

CSIZE 2                CSIZE 2

CSIZE 3                CSIZE 3

CSIZE 4                CSIZE 4

CSIZE 5                CSIZE 5

CSIZE 6                CSIZE 6

CSIZE 7                CSIZE 7

CSIZE 8                CSIZE 8

CSIZE 9                CSIZE 9

CSIZE 10               CSIZE 10
```
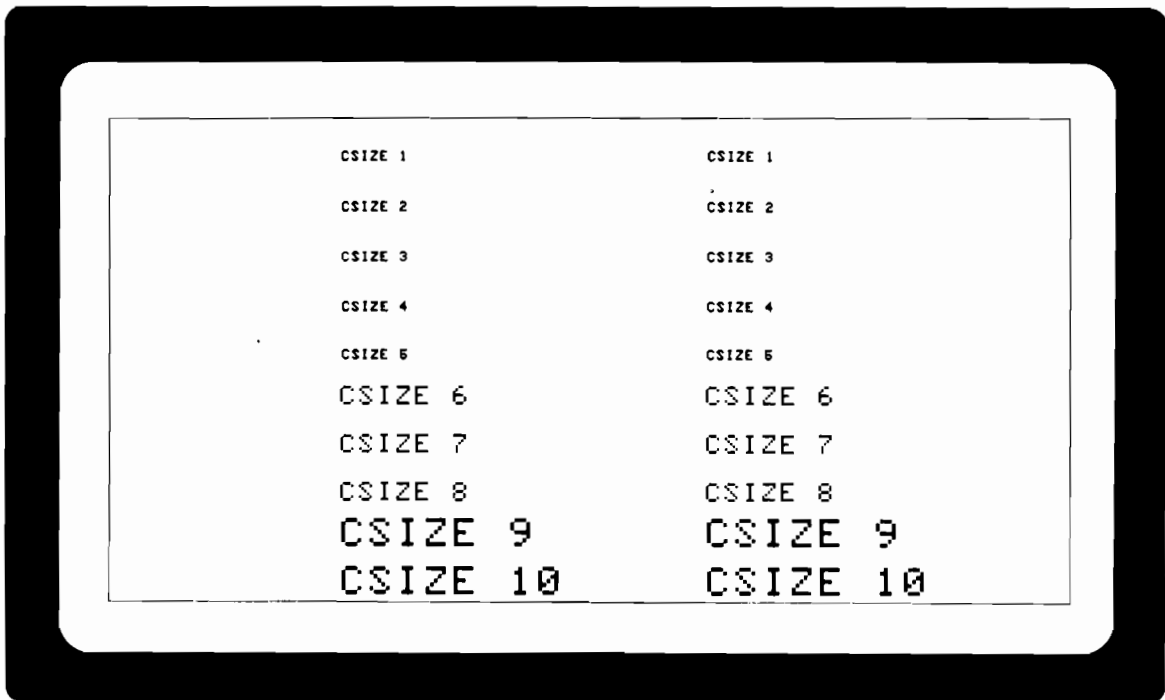
Plotting Functions

PEN

PEN (<pen number>)

where "pen number" is an integer in the range -2 to 8.

The PEN function selects any one of the pens found on the plotting device. This provides you with a convenient way of asking for one of four line types without knowing details of the device used.

PEN "0" selects a "blank" pen (returns all physical pens to holder, or selects a "NOP" mode in a CRT).

Pens 1-8 may be physical pens (presumably of several colors) or may be simulated by using line-types 0,2,3, or 4 (see "LINE") on a device without actual pens. If LINE has been called since the last call to GPON(2), the PEN statement is ignored. If a pen number greater than 8 is used, it will be interpreted modulo 8 (i.e. 9=1, 10=2, etc.).

A device with "m" pens will interpret the <pen number> modulo m.

Negative pen numbers are used for certain device dependent functions as on the terminal:

- PEN (0) is a blank pen.
- PEN (−1) is an "eraser" or clear mode.
- PEN (−2) complements the image along its path, such that a second complementing would restore the original display.

PEN(1) = LINE(0)
PEN(2) = LINE(2)
PEN(3) = LINE(3)
PEN(4) = LINE(4)

When a negative pen number is used on a plotter, this is interpreted as a request for a blank pen (PEN(0)).

Example:

```
1 REM...This program demonstrates how the pen statement is used
2 REM...to draw and erase lines on the CRT. A frame is drawn
3 REM...with a positive pen number and then erased with a
4 REM...negative pen number.
10 PLOTR                        !Initialize AGL
20 LET P=1                      !Initialize P, a pen that draws
30 REM...
40 PEN (P)                      !Specify draw or erase depending on P
50 FRAME                        !Frame the plotting area
60 P=-P                         !Switch to opposite value of P
70 IF P=1 THEN GOTO 30          !Repeat the loop
80 GOTO 30                      !Repeat the loop
```

PENUP/PENDN

PENUP
or
PENDN

where there are no parameters.

PENUP lifts the "pen" and PENDN lowers the "pen." PENUP declares that the following set of PLOT commands describe an object not to be connected to what has been drawn before. Typical use is to do a PENUP before a loop containing PLOT commands. A call to PENDN followed by a PENUP leaves a mark.

Examples:

30 PENUP
40 MOVE(100,200)
50 PENDN

LINE

LINE[(<linetype> [, <length>])]

LINE selects one of 8 (0-7) predefined line types. "Length" is a pattern repeat distance in GDUs. The default line type is solid (0). If a line type greater than 7 is used, it will be interpreted modulo 7 (i.e. 8=1, 9=2, etc.). Default length depends on device and pattern selected; values in the range of 4-10 GDUs are typical for plotters. On the terminal the length is 8 display dots.

- LINE 0 is always solid lines.
- LINE 1 is always "fainter" than type 0, for example, short or low duty-cycle dashes.
- LINE 0 or 2-4 are all distinct; additional distinct types may also be supplied by some devices.
- LINE types less than zero should not be used.

LINE patterns used by the terminal (3 cycles are shown):

```
0. ----------------------- (solid)
1. - - - - - - - - - - - - - (dim)
2. ----    ----    ----    (short dash)
3. ------   ------   ------ (long dash)
4. --- - --- - --- - (centerline)
5. .(starting point)    . (end points)
6. --- - - --- - - --- - -
7. --- --- --- --- ---
```

LINE 0 and the standard "fainter" LINE 1 can be used for drawing major and minor grid lines. LINE 0 and 2-4 are used to simulate the four "pens" on devices without color or physical pens.

Devices which support only types 0-6 should map larger type numbers onto standard types 0-6. Since not all devices can allow arbitrary lengths, the device should use the best available length. In some devices the pattern length will also vary line angle (e.g. by a factor of 1.414 for 45 degrees). Typically, pen plotters cannot support dim lines, so a request for LINE 1 results in use of LINE 0 or solid lines.

Note that GSTAT (9,1) will return the pattern repeat distance. If this value is 0, the device dependent default is being used.

```
1 REM...This program draws the 8 variable line types.
10 PLOTR                    !(Initialize AGL
20 FRAME                    !Frame plotting area
35 SCALE (0,10,-10,0)       !Specify user units
40 LORG (2)                 !Center labels on lines
50 FOR Mark=0 TO 7          !Specify one of eight lines
60 MOVE (1,-(Mark+1))       !Move to label position
70 LINE (0)                 !Specify line type for label
80 PRINT #0;Mark            !Label line
90 LINE (Mark,1)            !Specify line type for line
100 MOVE (2,-(Mark+1))      !Move to start of line
110 DRAW (8,-(Mark+1))      !Draw line
120 NEXT Mark               !Do it again for next line type
```
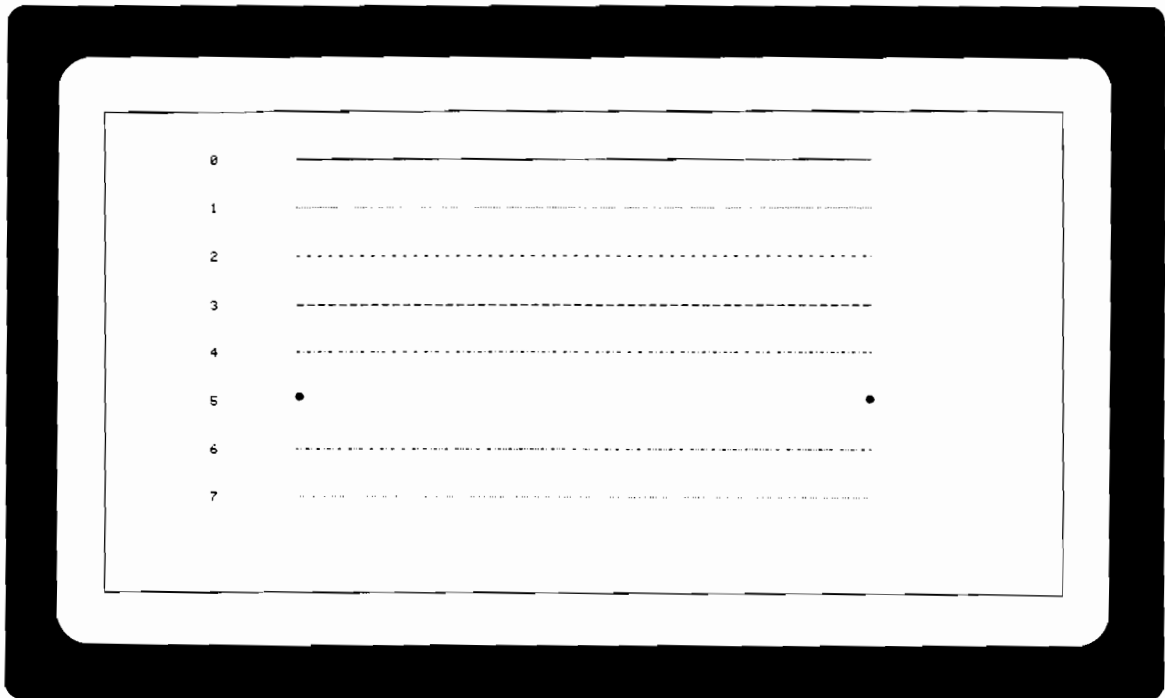
PLOT

PLOT (<x coord>, <y coord> [,<pen cntl>])

where X and Y coordinates are type REAL and are interpreted according to the current "units" mode. The pen-control parameter is an integer, and defaults to 1 (moves, then drops). The pen control parameter is interpreted as follows:

```
EVEN: lift pen (pen-up)
 ODD: drop pen (pen-down)
  + : pen change after motion
  - : pen change before motion
```

Examples:

```
+1 move or draw, then drop pen if up (default)
 2 move or draw, then lift pen if down
 0 move or draw, then lift pen if down
-1 lower pen, then DRAW
-2 lift pen, then MOVE
```

The PLOT function provides absolute data plotting with pen control. PLOT is the preferred pen moving command for automatically generated data, because the pen is under direct program control.

Pen motions will be clipped as shown under CLIP. PLOT commands will be affected by the current vector clipping limits (see CLIP). Clipping allows plotted data to be clipped at the plot boundary ("soft" clip), but to allow labels to be positioned outside the boundary, typically in GDUs.

In the typical case, a PENUP command is executed at the start of a program segment, and the default pen control mode is used in a subsequent plotting loop. The first data point plotted will then be isolated from any prior drawing.

Example:

```
10  PLOTR                         !Initialize AGL
15  REM...Define max and min values for scale statement
20  Xmin=-10
30  Xmax=10
40  Ymin=-10
50  Ymax=10
60  SCALE (Xmin,Xmax,Ymin,Ymax)   !Define user units
70  AXES (1,1,0,0,5,5)            !Draw axes
80  FRAME                         !Frame plotting area
90  REM...
100 READ Xvalue,Yvalue,Pc         !Read till out of data
110 IF Xvalue=0, THEN GOTO 390    !Stop is out of data
120 PLOT (Xvalue,Yvalue,Pc)       !Plot data
140 GOTO 90                       !Do it again until out of data
150 REM
151 REM...Data for first figure
160 DATA 2,2,1
170 DATA 8,2,-1
180 DATA 5,8,-1
190 DATA 2,2,2
191 REM...Data for second figure
200 DATA -4,2,-2
210 DATA -4,4,-1
220 DATA -2,4,-1
230 DATA -5,9,-1
240 DATA -8,4,-1
250 DATA -6,4,-1
260 DATA -6,2,-1
270 DATA -4,2,2
271 REM...Data for third figure
280 DATA -2,-2,1
290 DATA -8,-2,-1
300 DATA -8,-8,-1
310 DATA -2,-8,-1
320 DATA -2,-2,2
321 REM...Data for fourth figure
330 DATA 2,-2,1
340 DATA 8,-2,1
350 DATA 2,-8,-1
360 DATA 8,-8,-1
370 DATA 2,-2,2
371 REM...Data to end program
380 DATA 0,0,0
390 END
```
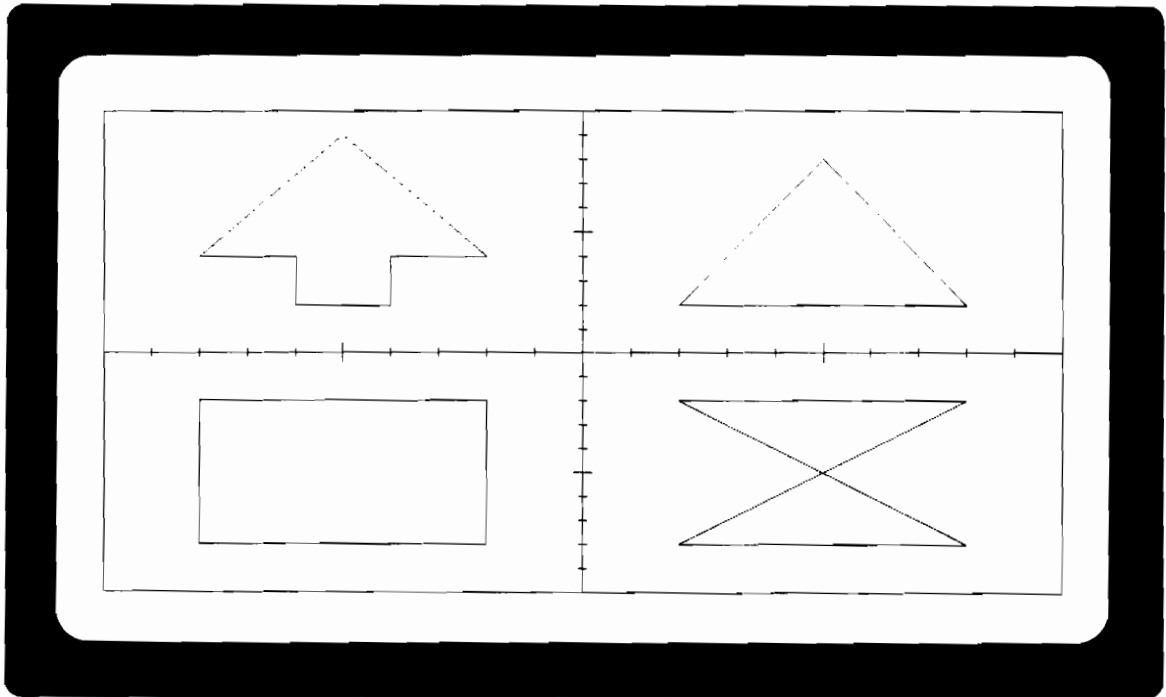
MOVE

MOVE (<x coord>, <y coord>)

where X and Y coordinates are interpreted according to the current units used. A MOVE (X,Y) is equivalent to a PLOT (X,Y,−2).

The MOVE function lifts the pen and moves it to the absolute X,Y coordinate. MOVE is allowed to define a logical pen position beyond the normal clipping limits. If the pen is off-page to the left and a label is started, the first few characters are invisible until the pen moves back within the hard-clipped region. Then the remaining characters are drawn normally.

Example:

```
50 MOVE (5,10)
```

DRAW

DRAW (<x coord>, <y coord>)

where X and Y coordinates are interpreted according to the current units used. A DRAW (X,Y) is equivalent to a PLOT (X,Y,−1).

The DRAW function drops the pen and moves it (within the soft clip region) to the absoute X,Y coordinate. DRAW allows an easy way of drawing a line from the current pen location to a new location without regard to whether the pen is up or down.

Example:

```
60 DRAW (30,30)
```

RPLOT

RPLOT (<x coord>, <y coord> [,<pen cntl>])

RPLOT provides relative plotting capability with pen control from the last plotted point. RPLOT interprets its coordinate values as being relative to a relocatable origin, whose location is the last point addressed by an absolute PLOT, IPLOT, MOVE, or DRAW, except if PORG has been executed (see PORG). This relocatable coordinate system may also be rotated about this origin relative to the master coordinate system by means of a plot direction (PDIR).

Example:

        200 RPLOT (10,20,2)

IPLOT

IPLOT (<x incr>,<y incr>[,<pen cntl>])

The IPLOT function provides incremental plotting capability with pen control. IPLOT plots incrementally from the current pen position. IPLOT causes the relocatable origin to be updated unless the PORG statement has been used. (See PORG.)

Example:

        30 IPLOT (5,-5,3)

PRINT #0

PRINT #0; text

The PRINT #0 statement is used to perform graphics labeling. The current graphics character size, slant, and origin is used. Appending a ";" (semi-colon) will not suppress a carriage return linefeed in the text line.

If the text contains a null character, output will be terminated by the null.

Example:

        60 PRINT #0; "This is the X-axis"

PDIR

PDIR (<angle in radians>)
or
PDIR (<x component>, <y component>)

The plotting direction (PDIR) function sets the angle of rotation for relative (RPLOT) and incremental (IPLOT) plotting. PDIR sets the orientation of the local coordinate system explicitly or by supplying a direction vector for the local X axis.

- If only one parameter is supplied, it is assumed to be the counter-clockwise angle in radians from the "right horizontal" direction (0 => towards 3 o'clock, 90 degrees => towards 12 o'clock, etc.). Angles are measured in whatever units would be expected by the trigonometric functions of the system at the time of the call.

- If two parameters are supplied, the angle used is that of an incremental vector plotted with X equal to the run and Y equal to the rise in current units.

Example:

```
10 PLOTR                            !Initialize AGL
20 SHOW (-6,6,-6,6)                 !Specify user units
30 FRAME                            !Frame plotting area
40 REM...Draw the face of a clock
50 CSIZE (8)                        !Specify character size
60 LORG (5)                         !Specify to center numbers
70 FOR Minute=1 TO 60               !For...next to specify
75                                  !minute mark
80 MOVE (0,0)                       !Move to center of clock
90 PDIR (6.2831853*(1-Minute/60))   !Plot direction for minute
100 IPLOT (0,4.5,-2)                !Move to rim of clock
110 IPLOT (0,.1,-1)                 !Draw minute mark
120 IF Minute MOD 5 THEN GOTO 160   !if not on 5 minute mark,
125                                 !skip to next
130 IPLOT (0,.2)                    !Draw longer 5 minute mark
140 IPLOT (0,.7,-2)                 !Move to center of label
150 PRINT #0;Minute/5              !Label with a number
160 NEXT Minute                     !Do it again for next
165                                 !minute
166 REM...Draw hands on clock
170 MOVE (0,0)
180 DRAW (0,4)
190 MOVE (0,0)
200 DRAW (3,0)
```
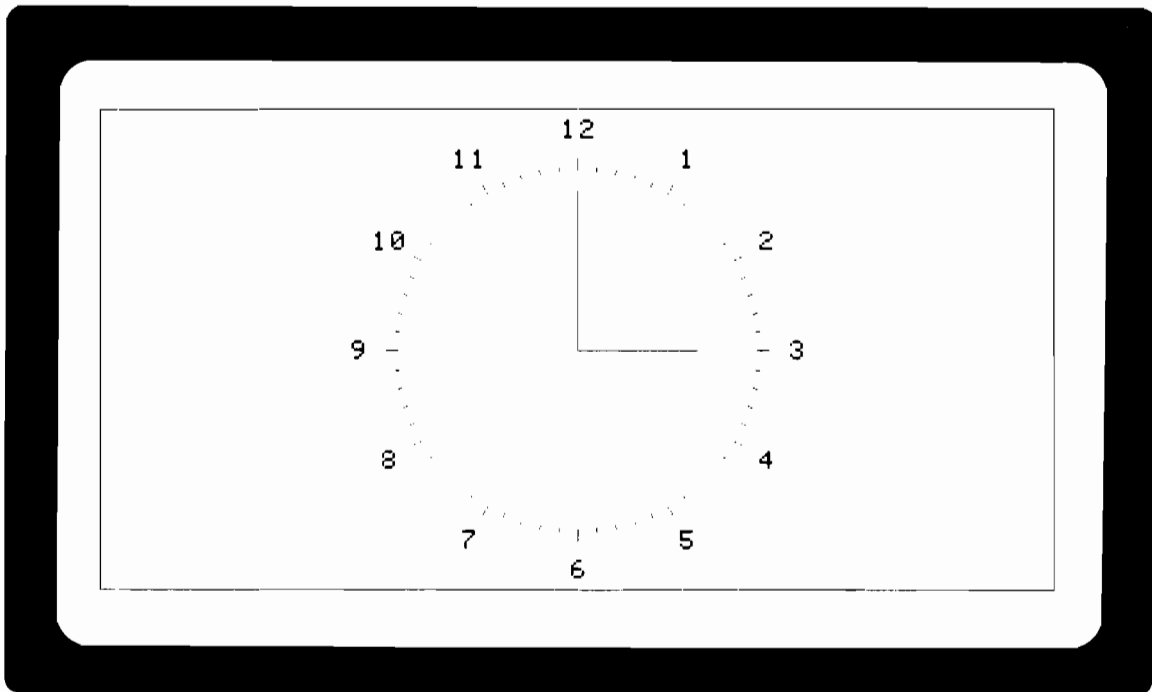
PORG

PORG [(<x coord>,<y coord>)]

The PORG defines the local origin for relative plotting (RPLOT). The X and Y coordinates specify a point where the relocatable origin is to be placed. The values are real and are interpreted in the current units mode. Default values are zero for both. The coordinates given override any previously defined points.

When the PORG statement is executed, it prevents the relocatable origin from being updated by MOVE, DRAW, and PLOT statements. Executing a GPON(2) statement will enable the updating of the relocatable origin by a MOVE, DRAW, or PLOT statement.

Example:

```
50 PORG (50,100)
```

Interactive Functions
WHERE

WHERE (<x coord.>, <y coord.>[, <pen var.>])

The WHERE function returns the pen location of the last plotted point. WHERE also determines the logical pen location and whether it is up or down. If the pen is up, pen status will be a 0; if it is down, it will be a 1.

This function can be used to allow an "isolated" software module to determine the pen position. The coordinates are type REAL and are in the current units mode. The pen status is an integer with a value of 1 or 0.

Example:

```
70 WHERE (x1,y1,p1)
```

POINT

POINT (<x coord>, <y coord>)

The POINT function positions the cursor under program control at the specified absolute location and specifies the cursor type. The X and Y coordinates are type REAL and are in the current units mode.

Example:

```
90 POINT (90,75)
```

CURSOR

CURSOR (<x coord.>, <y coord.>[<"z" coord.>])

The CURSOR function reads the cursor position without waiting for operator input. The X,Y coordinates are REAL values in current units. The X,Y coordinates are the same as for the POINT function. The Z coordinate is a binary valued integer: 1 = pendown and 0 = penup.

Example:

```
400 CURSOR (X3,Y3,Z)
```

DIGITIZE

DIGITIZE (<x coord.>, <y coord.>, <z coord.>)

DIGITIZE waits for a user response and then reads the coordinates of the cursor. The X and Y coordinates as well as the pen state (plotters) or key pressed (terminals) are returned in the digitize variables. The coordinates are in the current units system. If a user prompt is desired it must be output using a PRINT statement.

Press Control-G. CURSOR to see the terminal message line display the cursor coordinates. The coordinates are updated each time the cursor control keys are used. The display is initially in current units. Pressing Control-G. CURSOR will cause the units to be displayed in machine units. Pressing Control-G. CURSOR again will cause the message line to be cleared.
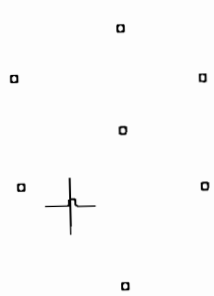
If a terminal is used as the input device, the graphics cursor will be turned on, indicating the current cursor position. You can then position the cursor using the graphics cursor keys.

Once the cursor is positioned, press any of the ASCII character keys. This will cause the cursor position to be read and returned in the X and Y variables. The ASCII code (0-127) for the key pressed will be returned in the Z variable.

If a plotter is used as the input device, the plotter's ENTER key is used to indicate that the pen has been positioned. The X and Y variables are set to the pen's coordinates and the Z variable is set to a "0" if the pen is up and a "1" if the pen is down.

```
>1 PLOTR
>2 LOCATE(0,200,0,100)
>3 SCALE(0,719,0,359)
>5 INTEGER X,Y,P
>10 DIGITIZE(X,Y,P)
>20 PLOT(X,Y,-2)
>26 READ X,Y
>27 IF (X=0 AND Y=0) THEN 46
>28 IPLOT(X,Y,-1)
>29 GOTO 26
>30 DATA 5,0,0,5,-5,0,0,-5,0,0
>46 RESTORE
>60 GOTO 10
>RUN
>
```

GPMM

$<$variable$>$ = **GPMM** ($<$millimeter value$>$)

GPMM (GDUs per millimeter) converts millimeters to GDUs. GPMM is a function which returns the number of GDUs corresponding to the number of millimeters specified by the argument. For CRT-like devices, the conversion is typically only an approximation.

GPMM may be imbedded in functions such as CSIZE to allow specifications in millimeters, or may be called once with an argument of 1 to get a multiplier to be used in similar situations. In the latter form it may also be used as a divisor to convert GDUs to mm.

Example:

```
20 N = GPMM (220)
```

DSIZE

DSIZE ($<$GDU X limit$>$,$<$GDU Y limit$>$
 [,$<$cell height$>$,$<$cell aspect ratio$>$
 [,$<$X resolution$>$,$<$Y resolution$>$]])

The DSIZE function returns the display size available for the plotting device.

The X,Y limits define G2 in GDUs and G1 is assumed set at 0,0. This is the available display space in GDUs. The character cell dimensions give the height in GDUs and the aspect ratio of the current character. The X,Y resolution parameters in GDUs represent the minimum available step size in each direction.

Example:

```
10 DSIZE (X0,A0,h,Ratio)
```

DSTAT

$<$ string variable $>$ = **DSTAT**

The DSTAT function returns a string containing the model number of the plotting device (i.e., "9872" for the HP 9872 plotter). If the device is not available, the DSTAT function returns a null string.

Example:

```
20 A$=DSTAT
```

GSTAT

< variable > = GSTAT [(<Index>[,<Subscript>])]

The GSTAT function returns the graphics display status of the plotting device. Index specifies a group from 0-14 and subscript specifies an element within the group from 0-5. The default parameters are 0 for Index and 0 for Subscript. The table below shows the relationship between the Index and Subscript parameters.

Example:

```
40 N = GSTAT (I,S)
```

*Given that Xu and Yu are coordinate values in a user coordinate system, the equivalent machine coordinates Xm and Ym are:

```
Xm = A * Xu + B
Ym = C * Yu + D
```

The scale factor for X is A. The offset for X is B. The scale factor for Y is C. The offset for Y is D.

In the example below, Xu and Yu are 50,50:

```
A = 10, B = -10
C = 20, D =   0

Xm = 10 * (50) + (-10) =   490
Ym = 20 * (50) +   0   = 1000
```

INDEX	SUBSCRIPT	DESCRIPTION
0		Pen Position
	0	X Pen Position in MUs
	1	Y Pen Position in MUs
1		Pen
	0	Pen State (0=up; 1=down)
	1	Pen Number
2		GDU Space Limits
	0	GX1 (MUs) low value
	1	GX2 (MUs) high value
	2	GY1 (MUs) low value
	3	GY2 (MUs) high value
3		Locate Mapping Points
	0	VX1 (CUs)
	1	VX2 (CUs)
	2	VY1 (CUs)
	3	VY2 (CUs)
4		Soft Clip Limits
	0	SX1 (CUs)
	1	SX2 (CUs)
	2	SY1 (CUs)
	3	SY2 (CUs)
5		User to Machine Scale Factors*
	0	A
	1	B
	2	C
	3	D
6		GDU to Machine Scale Factors*
	0	A
	1	B
	2	C
	3	D
7		Millimeter to Machine Scale Factors*
	0	A
	1	C
8		Current Units Mode (0=GDUs; 1=UDUs)
9		Line Type Information
	0	Line Type Argument
	1	Pattern Repeat Distance (GDUs)
10	0	LORG Argument
11		LDIR Argument
	0	Run
	1	Rise
12		PDIR Argument
	0	Run
	1	Rise
13		FXD Argument
	0	X
	1	Y
14		Relocatable Origin
	0	X (CUs)
	1	Y (CUs)

This section provides the detailed description of BASIC commands and statements. Refer to Section 11 for descriptions of the AGL statements.

Commands

Commands can be entered whenever a program is not executing. Some commands can be used in a BASIC program. The commands that can be used in a program are RUN, EXIT, and SET. The SET SIZE version of the SET command (2647A only) cannot be used in a program. The BASIC commands allow you to load and store program files, manipulate program listings, establish default parameters for the BASIC interpreter, execute programs, and return to normal terminal operation. The available BASIC commands are:

- AUTO   - GET   - RENUM
- CSAVE   - GO   - RUN
- DELETE   - LIST   - SAVE
- EXIT   - MERGE   - SCRATCH
- EXTEND   - REMOVE   - SET

AUTO

AUTO [starting line [,[increment]]]

The AUTO command causes the interpreter to generate line numbers automatically as the program is entered. The numbering will begin with the starting line number given in the command. If no starting line number is given, the numbering will begin with line "10". If an increment is given, the line numbers will be stepped by this value. If no increment is given, the line numbers will be stepped by 10. If only the comma is entered, (with or without a starting line number), the program will be stepped using the last increment used by the AUTO command. (This value is set to 10 whenever BASIC is reloaded.)

If the AUTO command is used during program modification and generates a line number already present in the program, an error message is generated and AUTO is terminated. Entering a Control-A will turn off the autonumbering feature.

Example:

```
AUTO  RETURN
10

AUTO 5,1  RETURN
5
```

CSAVE

CSAVE [range] [[TO] filename] [,SECURE]

The CSAVE command is similar to the SAVE command except that it causes a condensed version of the program to be kept. If SECURE is used, the kept program can be loaded but not listed by subsequent users.

Example:

```
CSAVE "FLEXY", SECURE  [RETURN]
```

DELETE

DELETE [range]

The DELETE command allows you to delete lines from your program. The range gives the line numbers to be deleted.

<range> = first line to be deleted — last line to be deleted

If the last line number is not used, only the one indicated line will be deleted. If a starting line number is followed by a dash only, lines are deleted from the starting line to the end of the program.

Example: In the program resulting from the last RENUM example, a command of DELETE 15-35 [RETURN] would result in:

```
10 LET X=0
40 END
```

The same result could be achieved by DELETE 11-39 [RETURN].

EXIT

EXIT

The EXIT command ends the BASIC interpreter and restores normal terminal operation.

GET

GET "<filename>"

The GET command reads in a copy of a program from a file and clears the current program from the workspace. If the file is not specified, the program is loaded from the last specified source device. The initial default source device is the "left tape" in emulation mode.

Examples:

```
GET  [RETURN]
GET "CHART"  [RETURN]
```

GO

GO [linenumber]

The GO command causes program execution to resume after the break character (normally a CONTROL-A) has been entered from the keyboard or a STOP statement has been executed. Execution resumes at the statement after the break occurred or following the STOP statement. If the terminal is waiting for input when a break occurs, the program will again request input. The input prompt (?) or string will also be reprinted.

The GO command is useful in debugging. It will allow you to interrupt an infinite loop, use direct computation statements to display or change variable values and then resume execution using the GO command.

Note that you cannot modify the program while the program is stopped and then resume execution. If statements are changed, you must rerun the program.

LIST

LIST [range] [[TO] filename]

The LIST command displays the current program in line number order. The range parameter can be used to list a sequential block of lines. If no range is specified, the entire program will be listed. The range parameter is of the following format:

<range> = first line to be listed — last line

The last line number is optional. If the last line number is not used, only the single specified line will be listed. If the first line is followed by a dash only, the program will be listed from the starting line to the end of the program.

Example:

```
LIST 30-50
```

MERGE

MERGE [filename]

The MERGE command loads a copy of the specified file into the BASIC workspace. The new program is merged with any existing program. If there are conflicting line numbers, the old lines are replaced by the new ones. If a filename is not specified, the program is loaded from the current source device.

Example:

```
MERGE "SALES" RETURN
```

REMOVE

REMOVE STD
or
REMOVE STDX

The REMOVE command is used to remove from the interpreter any commands or statements that have been added using the EXTEND command. REMOVE STD removes the extensions prepared by Hewlett-Packard (e.g., AGL).

REMOVE STDX removes the PRINT USING, PRINT # USING, PACK, IMAGE, CALL, SUB, SUBEND, and error messages from the BASIC interpreter. The statements removed with REMOVE STDX provide an additional 5K bytes of user workspace in the 2647A.

Examples:

```
REMOVE STD  RETURN
```

RENUM

RENUM [new starting line[,increment
[,old starting line [,old ending line]]]]

The RENUM command allows you to change or rearrange the line numbers in your program. If the starting line number is not given, the first line will be numbered 10. The increment is optional and specifies the increment between line numbers. If no increment is given, a value of 10 will be used. Line numbers less than the old starting line number will not be renumbered. If the old starting line number is not specified, the program will be renumbered beginning with the first line. If an old ending line is specified, the renumber process will continue until the specified point in the program is reached. Line numbers after the old ending line are unchanged.

Note that the RENUM command is not used to move lines from one position to another. All that it does is renumber lines, it does not change their order.

Example: Assume that the following program is present:

```
10 LET Y=0
11 LET X=0
17 INPUT X,Y
70 LET Z=X+Y
71 PRINT X,Y,Z
90 GOTO 17
999 END
```

Entering RENUM RETURN will cause the program to be renumbered to the following:

```
10 LET Y=0
20 LET X=0
30 INPUT X,Y
40 LET Z=X+Y
50 PRINT X,Y,Z
60 GOTO 30
70 END
```

Note that line numbers used in GOTO or GOSUB statements are automatically changed to the new line numbers.

Using the same program, RENUM 100 [RETURN] would cause the following:

```
100 LET X=0
110 LET Y=0
120 INPUT X,Y
130 LET Z=X+Y
140 PRINT X,Y,Z
150 GOTO 120
160 END
```

Specifying a statement increment, RENUM, 5 [RETURN] would result in the following:

```
10 LET X=0
15 LET Y=0
20 INPUT X,Y
25 LET Z=X+Y
30 PRINT X,Y,Z
35 GOTO 20
40 END
```

RUN

RUN [linenumber]
or
RUN "<filename>" [, linenumber]

The RUN command is used to execute your program. If the optional line number is not used, the program will begin executing from the first statement. If a line number is specified, program execution will begin with the indicated line.

When a filename is used, the program will first be loaded from the indicated file and then run. If no filename is specified and there is no program in the workspace, a program will be loaded from the current source file ("LEFT TAPE" in emulation mode) and then run.

Example: Execute the first program shown under the RENUM command beginning at the statement INPUT X,Y.

```
RUN 17 [RETURN]
```

RUN may also be used as a statement in a BASIC program, allowing one program to activate another. Run without a filename initializes program variables and continues execution at the specified line number. RUN with a filename deletes the current program and loads and runs the named program.

SAVE

SAVE [range] [[TO] filename] [BASIC]

The SAVE command stores a copy of the current program on a file. The filename selected must be different from those which already exist. If the file is not specified, the program is stored on the last specified destination device. The initial destination file is the "right tape" in cartridge tape emulation mode. Files stored to this destination will be stored as ASCII text files.

Example:

```
SAVE [RETURN]
SAVE "ROBOT" [RETURN]
```

*The BASIC option is used to save a copy of the current BASIC interpreter. This is useful when the interpreter has been modified using the EXTEND command. The range option cannot be used with the BASIC option.

Examples:

```
SAVE "RIGHT TAPE", BASIC [RETURN]
```

SCRATCH

SCRATCH
***SCRATCH [string space]**

The SCRATCH command deletes the current program and variables from the workspace. SCRATCH can be abbreviated SCR. If the string space parameter is used (2647A only), the memory space used to store string data is set to string space + 1 (string space must be an INTEGER expression).

Example:

```
SCRATCH [RETURN]
```

SET

SET <condition>
where <condition> is one of:

- **MULTIPLE**
- **SINGLE**
- **SHORT**
- **LONG**
- **INTEGER**
- **PROMPT**
- **KEY**
- ***SIZE**

The SET command allows you to select the default modes of operation for the interpreter. Only one condition can be used in a SET command. You cannot, for example, enter SET SINGLE, LONG.

The SET MULTIPLE command tells the BASIC interpreter whether or not to allow multiple statements per line number. The SET SINGLE command disables the interpreter's ability to use multiple statements in a line. The multiple statement capability can be reestablished with the SET MULTIPLE command or by reloading BASIC.

*2647A only.

12-6

When the SET MULTIPLE command is entered, the interpreter will accept more than one statement in the same line. The statements must be separated by a "\" (backslash). The line has only a single line number at the left margin as for a normal statement. Multiple statements are executed in order from left to right. The multiple statement capability is initially on.

```
100 READ A
200 PRINT A \ READ B \ PRINT B
300 FOR I=1 TO N \ A(I)=I^2 \ NEXT I
```

The main advantage of multiple statements is that you can save program storage space by not numbering each statement.

Example:

```
SET MULTIPLE  RETURN
```

If multiple statements are entered and the multiple statement capability is not turned on, the entire line is rejected and an error message is displayed.

The SET SHORT, SET LONG, and SET INTEGER commands are used to set the default data type for numeric variables. The last default data type used continues in effect until changed. If BASIC is reloaded, the default type is set to SHORT.

Examples:

```
SET INTEGER
SET LONG
```

The SET PROMPT command allows you to change the prompt character. The default character is ">". Note that only a single character can be used.

Example:

```
SET PROMPT ="/"
```

The SET KEY command allows you to select the control character to be used as a local "break" character (a letter A-Z). Entering the "break" character will cause execution of the local BASIC program to stop. The default "break" character is CONTROL-A.

Example:

```
SET KEY ="B"
```

In the 2647F the SET SIZE command reinitializes BASIC; it has no effect on the workspace.

In the 2647A the SET SIZE command allows you to allocate terminal memory resources between the terminal display and the BASIC workspace. Decreasing the size of the display memory will increase the amount of memory available to your BASIC program. <size> is the number of bytes to be allocated to the workspace.

Example:

```
SET SIZE = 21000
```

Note: the SET SIZE command cannot be executed from a BASIC program.

Statements

This section contains descriptions of the statements available in BASIC/47. The statements are listed in alphabetic order. Each statement is shown with statement syntax, a brief description, and statement examples. Most statements can be used in direct computation mode. Some cannot. Refer to Section 1 of this manual for additional information on direct computation mode.

Items in uppercase type must be included whenever the statement is used. The letters "LET" in the LET statement are an exception.

An ellipsis ("...") indicates items that can repeat indefinitely.

Lists consist of one or more of the items specified. Items within lists are separated by commas unless otherwise specified.

All punctuation marks (quotes, commas, colons, and semicolons) must be included.

ASSIGN

ASSIGN "filename" TO # filenumber   (To open file)
or
ASSIGN * TO # filenumber   (To close old file with that number and open new file)

The ASSIGN statement allows you to equate a filename to a logical filenumber. The filename can be a string variable or string expression. The file named must already exist. If it does not exist, ensure that a CREATE FILE command precedes the ASSIGN statement in the program. The filenumber is used by BASIC statements to select the file to be used in a file operation. This numerical reference makes your program independent of the name used for a data file. The filenumber must be between 1 and 127.

You can rerun the program with a different filename simply by changing the filename used in the ASSIGN statement. The ASSIGN statement allows you to reassign files while the program is running so that several files can be acted on in succession.

```
200 COMMAND "CREATE FILE 'DATA1'"
300 ASSIGN "DATA1" TO #3
400 ASSIGN * TO #2
500 ASSIGN "TE#10" TO #1
600 ASSIGN "H#6" TO #2
700 ASSIGN "DA" TO #4
```

CALL

CALL <subprogram name> [(<parameters>)]

The CALL statement is used to transfer control to a subprogram. The parameters can be numeric or string variables or expressions, or they can be logical file numbers. When a CALL statement is executed, the values of the CALL parameters are equated to similar variables in the SUB statement. (A SUB statement must be the first statement in the CALLed program.) Note that the variables in both the CALL statement and the SUB statement must be of the same type and order. Refer to the SUB statement for additional information.

```
10 FOR I=1 TO 10
20 INPUT A$
30 CALL TEXT(I,A$)
40 NEXT I
   .
   .
   .
100 SUB TEXT(J,B$)
110 PRINT "LINE#";J;TAB(10);B$
120 SUBEND
```

COMMAND

COMMAND commandstring [,variable]

The COMMAND statement allows you to execute terminal commands from your BASIC program. If a variable is specified, the execution status of the command is returned in the variable. If the return variable is not used and the command does not execute properly, your program will be terminated. If the variable is used and the command executes properly, the variable will be set to zero before execution of the next program statement. On failure of the command, the variable will be set to one of several error codes depending on the nature of the error. Refer to Appendix E for a description of the error codes.

If a quote is required within the command string, use the apostrophe character (') as required.

Examples:

```
10 COMMAND "CR F 'SAMPLE'"   ! Create a file named "SAMPLE"
20 COMMAND A$
```

DATA

DATA datalist

Provides data to be read by READ statements. DATA statements may be located anywhere in the program; all DATA statements in a program are considered part of a single continuous list of data for that program. Data items may be numeric or string constants.

```
10 DATA 10
20 DATA "CALIFORNIA"
30 DATA 405,"OREGON",999
```

DIM

DIM itemlist [arraysize and/or stringlength]

Specifies the maximum length of strings and the maximum array size of numeric or string variables. String lengths are enclosed in brackets; array dimensions in parentheses. Numeric arrays declared in DIM statements assume the default numeric type. (Refer to LONG, INTEGER, and SHORT for additional information.)

```
70 DIM A$[25]
90 DIM K(18)
130 DIM P$(11,11)[25]
```

END

END

Terminates execution of the program. The END statement can be placed anywhere in a program except following a SUBEND statement.

```
90 END
```

ERROR

ERROR errornumber [, message]

This statement causes an immediate branch to the error handling routine. The error code placed in ERRN is the integer value of the errornumber parameter. Errornumber can be any integer expression. The ERROR statement can be used to cause application error conditions to be acted on by the same routines that would normally handle programming errors. Make sure that the error numbers that you use are different from any of the BASIC or terminal system errors. If they are not different, you will not be able to tell an application error from a programming error. (See the ON ERROR statement for a list of error numbers.) If no message exists for the error, a special message will be printed to indicate that there is no standard error message.

The optional message can be any string expression. If the message parameter is used, the user defined message will be displayed, replacing any standard message.

```
100 ERROR N+40
```

FOR...NEXT

FOR variable = initial TO final [STEP size]

.

.

.

NEXT variable

The FOR...NEXT statements allow repetition of a group of statements between FOR and NEXT. The number of repetitions is determined by the initial and final values of a loop variable, and by an optional STEP size. The loop variable cannot be an array element or of type LONG.

```
110 FOR I=1 TO 5
120 FOR J=1 TO -3 STEP -1
130 PRINT J
140 NEXT J
150 NEXT I
```

GETDCM ON/OFF

GETDCM ON
or
GETDCM OFF

The GETDCM statement enables and disables the operation of the GETDCM function. GETDCM ON enables the GETDCM function (described later in this manual) and disables interrupts from interrupt characters received from the host computer over the terminal's data communications port. Note that interrupts from the terminal's keyboard are not affected. (Refer to SET KEY and ON KEY for additional information on interrupt keys). GETDCM OFF disables the GETDCM function and enables data communications interrupts.

```
10 GETDCM ON
```

GETKBD ON/OFF

GETKBD ON
or
GETKBD OFF

The GETKBD statement enables and disables the operation of the GETKBD function. GETKBD ON enables the GETKBD function and disables program interrupts from the keyboard. (Refer to SET KEY and ON KEY for additional information.) GETKBD OFF disables the GETKBD function and enables the keyboard interrupt keys. Note that GETKBD ON/OFF does not affect program interrupts received through the data communications port.

```
20 GETKBD ON
```

GOSUB

GOSUB line number

Transfers control to the specified statement. The line number must refer to the first statement of a subroutine within the current program unit. A RETURN statement is required in the subroutine to return control to the statement following the GOSUB statement.

The GOSUB statement differs from the CALL statement in that the GOSUB statement can not pass parameters to the subroutine.

```
80 GOSUB 1000
90 REM CONTINUE PROCESSING
 .
 .
 .
1000 REM: START OF SUBROUTINE
1010 RETURN
```

GOTO

GOTO line number

Transfers control to the specified statement in the current program unit.

```
160 GOTO 200
```

IF...THEN...ELSE

IF expression THEN statement [ELSE statement]

Evaluates a conditional expression and specifies an action to be taken if the condition is true. A conditional expression is considered true if its value is non-zero, false if its value is zero. The action may be a transfer to a line number or one or more executable statements.

```
180 IF A<0 THEN 400
190 IF A=B THEN PRINT B
```

IMAGE

IMAGE formatstring

The IMAGE statement provides format specifications to be used in a PRINT USING statement. The PRINT USING statement must reference the line number of the IMAGE statement to be used. The IMAGE statement cannot be used in a multistatement line and connot contain comments.

```
200 PRINT USING 300; A$,B,C$
300 IMAGE "MONTH",3A,DD.DD,4A
```

INPUT

INPUT ["message",] variablelist

The INPUT statement requests input to one or more variables by displaying a "?" prompt. The program will then accept string or numeric data from the keyboard (See also KEYCDE.) If an optional message is used, the message replaces the "?" prompt.

```
10 INPUT A$,B
20 input "ENTER TIME",T
```

INTEGER

INTEGER itemlist (arraysize)

Declares that the variables in the itemlist are type INTEGER. If array size is specified, the maximum size for the array is also set. Integers must be in the range -32768 to $+32767$.

        20 INTEGER B(20)

KEYCDE

KEYCDE (shift state, keynumber, new keycode)

KEYCDE assigns the code values to be used for decoding the keyboard. Each of the keys on the keyboard is mapped to a code value (normally ASCII). The keys are numbered according to the diagram shown in figure 12-1. When a key is pressed, its location number is returned to the terminal. The number is used to select the proper ASCII code out of a table. For example, key number 109 would normally be the "A" key. Element U(109) would be equal to 65 (the ASCII decimal code for A).

The valid key numbers are 0-112. Keycodes can be between 0 and 255. The left and right shift keys, CNTL, AUTO LF, BLOCK MODE, CAPS LOCK, REMOTE, and RESET keys cannot be redefined.

There are three possible code tables from which the keycode can be obtained. The S parameter in the statement is used to indicate which one of the tables is being defined with the current array. S selects the keycode table as follows:

S = 0 Unshift character codes
    1 Left shift key characters
    2 Right shift key characters
    3 Left and right shift characters (defines both shift tables)

When the left shift key (#4 in figure 12-1) is pressed in addition to the character key, the left shift array will be used to select the character code. The right shift key (#5 in figure 12-1) will cause the character to be selected from the right shift table. Table 12-1 lists the default values for the keycode tables.
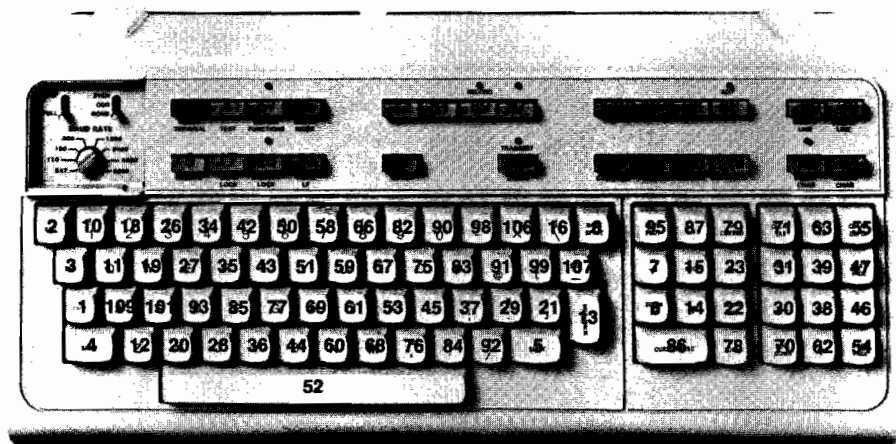
UNUSED KEYNUMBERS: 49 94 100 102 103 108 110 111

Figure 12-1. Terminal Keyboard Layout

Table 12-1. Default Keyboard Codes

Key #	UNSHIFT Code	UNSHIFT Key	SHIFT Code	SHIFT Key
1	0	[CNTL]	same	same
2	27	[ESC]	same	same
3	9	[TAB]	same	same
4	0	L [SHIFT]	same	same
5	0	R [SHIFT]	same	same
6	139	[ZOOM IN]	141	[CLEAR]
7	165	[◄]	144	[DRAW]
8	8	[BACK SPACE]	same	same
9	134	[REMOTE]	same	same
10	49	1	33	!
11	113	q	81	Q
12	122	z	90	Z
13	13 / 239	[RETURN]	same	same
14	164	[▼]	150	[T ANG]
15	135	[STOP]	143	[A DSP]
16	92	\	124	|
17	131	[CAPS LOCK]	same	same
18	50	2	34	"
19	119	w	87	W
20	120	x	88	X
21	93	]	125	}
22	140	[ZOOM OUT]	151	[T SZE]
23	163	[►]	145	[MOVE]
24	155	[COMMAND]	same	same
25	236 / 237	[MEMORY LOCK]	same	same
26	51	3	35	#
27	101	e	69	E
28	99	c	67	C
29	58	:	42	*
30	196	[←]	same	same
31	211	[ROLL UP]	same	same
32	156	[READ]	same	same
33	132	[AUTO LF]	same	same
34	52	4	36	$
35	114	r	82	R
36	118	v	86	V
37	59	;	43	+
38	232	[◄]	same	same
39	193	[↑]	same	same
40	157	[RECORD]	same	same
41	250	[TAPE TEST]	same	same
42	53	5	37	%
43	116	t	84	T
44	98	b	66	B
45	108	l	76	L
46	195	[→]	same	same
47	213	[NEXT PAGE]	same	same
48	158	[SOFTKEYS]	same	same
49	0	(not used)		
50	54	6	38	&
51	121	y	89	Y
52	32	SPACE	same	same
53	107	k	75	K
54	214	[PREV PAGE]	same	same
55	202	[CLEAR DSPLY]	same	same
56	209 / 210	[INSERT CHAR]	same	same

Key #	UNSHIFT Code	UNSHIFT Key	SHIFT Code	SHIFT Key
57	217 / 154	[DISPLAY FUNCTIONS]	same	same
58	55	7	39	'
59	117	u	85	U
60	110	n	78	N
61	106	j	74	J
62	194	[↓]	same	same
63	177	[SET TAB]	same	same
64	208	[DELETE CHAR]	same	same
65	133	[BLOCK MODE]	same	same
66	56	8	40	(
67	105	i	73	I
68	109	m	77	M
69	104	h	72	H
70	212	[ROLL DOWN]	same	same
71	178	[CLEAR TAB]	same	same
72	205	[DELETE LINE]	same	same
73	240	[f1]	same	same
74	152	[ENTER]	same	same
75	111	o	79	O
76	44	'	60	<
77	103	g	71	G
78	138	[ZOOM]	149	[TEXT]
79	136	[G CURSOR]	142	[G DSP]
80	204	[INSERT LINE]	same	same
81	241	[f2]	same	same
82	57	9	41	)
83	112	p	80	P
84	46	.	62	>
85	102	f	70	F
86	166	[CURSOR FAST]	146	[AUTOPLOT MENU]
87	162	[▲]	137	[RB LN]
88	247	[f8]	same	same
89	242	[f3]	same	same
90	48	0	same	same
91	64	@	96	`
92	47	/	63	?
93	100	d	68	D
94	0	(not used)		
95	147	[AUTO PLOT]	148	[AXES]
96	246	[f7]	same	same
97	243	[f4]	same	same
98	45	-	61	=
99	91	[	123	}
100	0	(not used)		
101	115	s	83	S
102	0	(not used)		
103	0	(not used)		
104	245	[f6]	same	same
105	153	[BREAK]	same	same
106	94	^	126	~
107	95	—	127	▮
108	0	(not used)		
109	97	a	65	A
110	0	(not used)		
111	0	(not used)		
112	244	[f5]	same	same

LET

LET variablelist = expression

The let statement assigns a value to a variable. The keyword LET may be omitted. Multiple variables can be assigned a common value by using a comma to separate the variables.

```
10 LET X = 10
20 Y=20
30 X,Y = 30
```

LINPUT

LINPUT ["message",] stringvariable

An entire line of input is accepted as a single string. The data can be assigned to a string or substring. Extra characters are truncated if the destination string is not large enough. If the optional message is used, the message will be printed on the display in place of the prompt.

```
30 LINPUT H$
90 LINPUT "Name:", N$
```

LINPUT

LINPUT # filenumber [BYTE count]; stringvariable

The LINPUT # statement is similar to LINPUT except that the line of input data is read from the file specified by the file number. File numbers are assigned using the ASSIGN statement.

If the optional byte count is used, only the number of bytes (characters) specified will be read into the string variable. If a byte count of 0 is given, the length of the string as determined from dimension or data type declarations will be used to determine the number of characters to be input.

```
10 LINPUT #3;B$
20 LINPUT #4 BYTE 30; A$
```

LONG

LONG itemlist (arraysize)

The LONG statement is used to declare the variables in the itemlist to be of type LONG.

```
600 LONG A,B(4,4)
```

NEXT (See FOR...NEXT)

OFF KEY

OFF KEY # keycode

Disables any interrupt associated with the indicated key. The keycodes are shown in Table 12.1 (see KEYCDE). Refer to ON KEY # for additional information.

```
10 OFF KEY # 32
```

ON END

ON END # filenumber CALL subprogram
or
ON END # filenumber GOSUB linenumber
or
ONE END # filenumber GOTO linenumber

The ON END statement causes a transfer of control to the specified statement or subprogram when an end of file condition is detected. Note that the transfer will occur for the specified file only. The ON END statement must be executed prior to the end of file condition in order to be effective. Subsequent ON END statements can change the destination subprogram or linenumber. (Refer to the ON ERROR statement for additional information.)

The ON END # statement has effect only within the current program unit. It will not affect operation of a subsequently called subprogram. A new ON END # statement must be executed in each subprogram.

If an end of file condition is reached and an ON END statement has not been previously executed for the file, the program will be terminated with an error.

```
10 ON END # 1 CALL Help
20 ON END # 2 GOSUB 300
30 ON END #3 GOTO 500
   .
   .
   .
110 PRINT #1;X,Y
120 READ #2;A$
130 PRINT #3;A$,X,Y
   .
   .
   .
700 SUB Help
710 PRINT "NEED BIGGER FILE";
720 SUBEND
```

ON...GOSUB

ON expression GOSUB linelist

Causes the program to execute a subroutine at one of several line numbers. The expression is evaluated and rounded to an integer. This integer selects the first, second, or nth line number from the linelist. If the integer is less than 1 or greater than the number of lines in the linelist, the statement following the ON...GOSUB statement is executed instead.

Upon return from the selected subroutine the program will continue execution with the statement following the ON...GOSUB statement.

```
10 ON N+M GOSUB 300,400,500,600,700
```

ON...GOTO

ON expression GOTO linelist

Causes a branch to one of several statements. The expression is evaluated and rounded to an integer. The integer is then used to select a statement from the linelist in the same manner as the ON...GOSUB statement. If the integer is less than 1 or exceeds the number of line numbers in the linelist, the statement following the ON...GOTO statement will be executed.

```
400 ON N+2 GOTO 100,200,300
```

ON ERROR

ON ERROR GOTO linenumber

When an error is detected by the interpreter, a branch to the indicated linenumber will occur. The type of error can then be examined and the appropriate action taken. The RESUME statement should always be used to continue program execution following the error handling routine.

Once an ON ERROR statement has been executed, all errors will cause a branch to the indicated error handling routine. If the linenumber given is not valid, a special error message will be generated.

```
10 ON ERROR GOTO 1000
```

If you specify a linenumber of 0, the ON ERROR branch is disabled and subsequent errors will cause the program to halt and an error message to be printed. If the ON ERROR GOTO 0 statement is contained in the error handling routine itself, it will cause the program to stop and print the error message. You should use ON ERROR GOTO 0 in error handling routines when there is no recovery procedure for a particular error.

If an error occurs within the error handling routine, the program will halt and an error message for the error occurring in the error handling routine will be printed. You cannot trap errors that occur within the error handling routine.

The ON ERROR GOTO statement must be executed before the error occurs. You can test for the type and location of the error using the error variables ERRL and ERRN. ERRL contains the line number in which the error occurred. If there is no error, ERRL will be 0. In Direct Computation mode ERRL will always be 65535. ERRN contains the code number of the error. If there is no error ERRN will be 0.

ERRL and ERRN cannot be used on the left side of the "=" sign in a LET or assignment statement.

A list of error types and messages is given at the back of this manual.

The following example shows how a simple error handling routine would work.

```
200 ON ERROR GOTO 400
210 INPUT "WHAT ARE THE VALUES TO DIVIDE?";X,Y
220 Z=X/Y
230 PRINT "THE QUOTIENT IS ";Z
240 GOTO 210
400 IF ERRN=1034 AND ERRL=220 THEN 420
410 ON ERROR GOTO 0
420 PRINT "DIVIDE BY ZERO ERROR"
430 RESUME 210
```

ON KEY

ON KEY # keycode CALL subprogram
or
ON KEY # keycode GOSUB linenumber
or
ON KEY # keycode GOTO linenumber

The ON KEY # statement provides a program interrupt when the indicated key is pressed. The default keycodes are listed in Table 12-1 (see KEYCDE). When the selected key is pressed, the program will perform the specified transfer as soon as the current statement has completed execution. The RETURN from the GOSUB will return the program to the statement following the point where the key interrupt occurred. Interrupt keys are not detected if they are entered as part of the response to an input or linput statement.

```
10 ON KEY # 27 CALL Gotcha
20 ON KEY # 32 GOSUB 400
30 ON KEY # 12 GOTO 100
```

PACK

PACK string variable USING format string ; item list
or
PACK string variable USING IMAGE line number ; item list

Allows formatted data to be written to a string variable rather than a file. The string variable can then be manipulated as a variable. The formatting symbols used in the format string are the same as those used in PRING USING and IMAGE statements. Refer to Table 12-2 for examples and descriptions of format symbols.

PRINT

PRINT printlist

Prints the values of a list of expressions on the terminal screen. Items in the list may be separated by commas or semicolons. Commas space output in 15-character fields. Semicolons cause output to be printed without extra spaces. A comma or semicolon at the end of the print list suppresses the final carriage return and linefeed. The print list may also include print functions to control output format.

```
480 PRINT A,5,8/E;B$[4]&C$&"Q=9*",
490 PRINT "NAME",SPA(3);"ADDRESS"
```

PRINT

PRINT # filenumber [BYTE count]; printlist
or
PRINT # filenumber [BYTE count] USING format;
printlist

PRINT # writes values of expressions in the printlist serially to the specified file. A record is sent to the file each time a line feed character is encountered in the printlist (unless the BYTE option is used) and at the end of the printlist. If the printlist is ended with a comma or semicolon the data is sent to the specified file without carriage return and line feed characters at the end of the record.

The filenumber parameter is normally an integer between 1 and 127. If a filenumber of 0 is used the operation will be interpreted as an AGL operation. Refer to the AGL (A Graphics Language) description later in this manual for a description of PRINT # 0.

Note that if you attempt to print 256 or more consecutive characters to a given file, a buffer overflow error will result.

If the optional byte count is used, BASIC will transfer the specified number of bytes to the file. This allows the transfer of 8-bit data and suppresses transmission of records upon encountering the line feed character. If a byte count of 0 is given, the actual byte count will be determined from the printlist items.

CAUTION

You cannot write 8-bit data to ASCII files.

The PRINT # USING statement allows you to format data as it is written to the file. Refer to the PRINT USING statement.

```
10 PRINT #3; A$,R
30 PRINT #N; "HELLO"
40 PRINT #1 BYTE 24; A$,B$
50 PRINT #1 BYTE 80 USING 200; A,B,C$
```

PRINT USING

PRINT USING format ; printlist
or
PRINT USING image line ; printlist

The PRINT USING statement prints the values of the items in the printlist according to the specification given in the format string or in the referenced IMAGE statement. If the end of the format string is reached before all of the items in the printlist have been output, the format string is repeated.

The examples below show three ways of referencing format strings. The first (530) contains the format string in the PRINT USING statement itself. The second (540) uses a string variable to reference the format string. The third refers to an IMAGE statement containing the format string.

```
530 PRINT USING "3A,3A";A$,"ABC"
540 PRINT USING F$;N,M,C$
550 PRINT USING 200;A,B,D
```

Format Symbols

The format string used in PRINT USING and IMAGE statements is made up of the following format control characters:

Table 12-2. Format Symbols

Symbol	Description	Example
Strings		
A	ASCII Character	AAA
K	Compacted format	K
Blanks		
X	Blank space	XXX
Separators		
,	Separator only	AA,DD
/	Separates specifications and begins a new line or a new record if writing to a file.	AA/DD
@	Separates specifications and, on lineprinter output, begins a new page.	AA@DD
	Characters (used at the end of the print list)	
Carriage Control	Suppress linefeed	
+		
	Suppress carriage return	
−		
	Suppress both linefeed and	
#	carriage return	
Replicators		
n	Single replicator	3A
n( )	Group replicator	3(3A,2X)
Numeric Specifications		
S	Sign character (+ or −)	SDD
M	Minus sign	MDD
D	Numeric digit, blank fill	DDD
.	Decimal point (.)	DDD.DD
R	European Decimal point (,)	DDDRDD
C	Comma	DDDCDD
P	Period (European comma)	DDPDDD
K	Compacted format	"ONLY",XKX,"ITEMS"

READ

READ variablelist

The READ statement assigns numbers and strings from one or more DATA statements to the variables specified in the variable list.

```
570 READ X,A$,B$,Y
575 READ A(N,M),B(N,M)
```

READ

READ # filenumber; variablelist

The READ # statement assigns numbers and strings from the specified file to the variables in the variable list. Reads the file serially beginning at the current position of the file pointer. The file is read until the variable list is exhausted. The read may cross record boundaries. Data entries in the file must be separated by commas.

```
580 READ #5; A, B$
595 READ #1; A(N,M)
```

REMARK

REM comments or ! comments

Allows you to place remarks within program listing. Any characters may be used. Note that the "!" character can be used following a statement in a line to indicate that the remaining characters on the line are comments. The "!" form of comment cannot be used in IMAGE or SUBEND statements.

```
640 REM ---ANY TEXT---
```

RESTORE

RESTORE [linenumber]

The RESTORE statement resets the data pointer to the specified DATA statement. If the linenumber is omitted, the pointer is reset to the lowest numbered DATA statement in the current program unit.

```
650 RESTORE 60
```

RESTORE

RESTORE # filenumber

Repositions the file pointer to the beginning of the specified file.

```
660 RESTORE #2
```

RESUME

RESUME [linenumber]
or
RESUME NEXT

Causes the program to continue execution after an error recovery routine (ON ERROR) has been executed. You can resume execution at one of the following points:

- the statement that caused the error (no linenumber)
- the statement following the error (NEXT)
- some specified line number

To continue execution at the statement causing the error use the following statement:

    40 RESUME

To continue execution at the statement following the error use the following statement:

    40 RESUME NEXT

To continue execution at a specified line number use the following statement:

    40 RESUME 500

Note that the linenumber parameter must be a valid line number (>0). Refer to the ON ERROR statement for additional information.

RETURN

RETURN

Returns control from a subroutine to the statement immediately following the most recently executed GOSUB statement.

SHORT

SHORT itemlist (arraysize)

Declares the variables in the itemlist to be type SHORT. If arraysize is specified, the maximum size for the array is also set.

    730 SHORT A,B(4,4)

SLEEP

SLEEP

The SLEEP statement allows you to place the interpreter into a dormant state. Execution of the BASIC program is suspended and the terminal returns to normal operation. The BASIC interpreter will continue to monitor the data comm and keyboard input however. The terminal will remain in the normal mode of operation until either the "break" key is entered (normally a CONTROL-A) or a key is pressed for which an ON KEY statement was executed in the suspended program. If the "break" key is used to return to the BASIC interpreter, the suspended program will resume at the statement following the SLEEP statement.

If a keycode interrupt is used to return to BASIC, the suspended program will execute the statement(s) specified in the ON KEY statement. If the ON KEY...GOSUB or the ON KEY...CALL forms were used, the indicated subroutine or subprogram is executed and then the program will return to the SLEEP state. If the ON KEY...GOTO form of statement was used, the program will not return to the SLEEP state unless a SLEEP statement is again executed.

The WAKEUP statement can also be used in an ON KEY subroutine or subprogram to cancel the current SLEEP statement.

Examples:

```
20 ON KEY # 160 GOSUB 200
30 ON KEY # 170 CALL Sbproga
40 ON KEY # 180 GOTO 300
50 SLEEP
    .
    .
    .
200 PRINT "TEXT1"
210 RETURN
300 STOP
    .
    .
    .
400 SUB Sbproga
410 PRINT "TEST2"
420 WAKEUP
430 SUBEND
```

STOP

STOP

The STOP statement terminates program execution. The program can be resumed with the GO command.

```
740 STOP
```

SUB

SUB subprogramname [(parameters)]

The SUB statement must be the first statement in a subprogram. It provides the name of the subprogram and marks the entry point for the main or calling program. The SUB statement also lists any parameters that are to be passed between the calling program and the subprogram. The last statement in the subprogram must be a SUBEND statement.

```
300 READ #1 ;A$
400 C=NUM(A$[1,1])
410 IF C>48 AND C<57 THEN 440
420 CALL Alpha (C)
430 GOTO 300
440 CALL Numeric (C)
    .
    .
    .
500 SUB Alpha (N)
502 PRINT CHR$(N);
504 SUBEND
600 SUB Numeric (N)
602 K=K*10+N
604 SUBEND
```

SUBEND

SUBEND

The SUBEND statement is used to indicate the end of a subprogram. Refer to the SUB statement for additional information.

The SUBEND statement must be the only statement on the line. It cannot be used in multistatement lines or with the "!" form of comments or with the REMARK statement. The only statement that can follow the SUBEND statement is a SUB statement beginning another subprogram. If any other statement type is used, the message MISSING SUB will be displayed and the program will be halted.

```
10 SUB Test (N,K,A$)
   .
   .
   .
50 SUBEND
```

WAKEUP

WAKEUP

The WAKEUP statement cancels the effect of the last executed SLEEP statement.

Example:

```
400 WAKEUP
```

Built-In Functions

The following paragraphs describe the built-in functions available in BASIC. The functions are divided into the following categories:

- Numeric
- Print
- Trigonometric
- Input/Output
- String
- Other

Within each category, the functions are listed in alphabetic order. The letter which appears after most function names specifies the type of result that is returned by the function. These letters are interpreted as follows:

S — short
I — integer
L — long
ST — string
T — same type as X

Numeric Functions

Function Type Description

ABS(X) T Absolute value of X.

```
10 A = ABS(2)!  A = 2
20 B = ABS(-2)! B = 2
```

CSENA(R,C) I Returns the absolute row and column position of the cursor in R and C. C is 1 to 80 and R is 1 to 528. The function value is set to the number of the current user window (1-4).

```
10 A = CSENA(P,Q)
```

CSENS(R,C) I Returns the screen relative row and column position of the cursor in R and C. C is 1 to 80 and R is 1 to 24. The function value is set to the number of the current user window (1-4).

```
10 B = CSENS(A(1),B(1))
```

EXP(X)S The natural log e raised to the X power (E X).

```
70 G=EXP(2) ! G = E^2 = 7.39
```

INT(X) I Largest integer $<=$ X.

```
80 H=INT(6.999)!  H=6
```

LOG(X) S Natural logarithm ; $X>0$

```
100 J=LOG(2)
```

LONG(X) L Returns the LONG value of X.

```
120 R=LONG(Y)
```

RND S The next pseudo-random number in a standard sequence of numbers $>= 0$ and <1.

```
130 M=RND
```

SGN(X) I The sign of X; -1 if $X<0$, 0 if $X=0$, and $+1$ if $X>0$.

```
140 N=SGN(J)
```

SHORT(X) S X rounded to SHORT representation.

```
145 M=SHORT(N)
```

SQR(X) S The positive square root of X.

```
160 Q = SQR(169)
```

Trigonometric Functions

ATN(X) S Returns the arctangent of X in radians.

```
200 E = ATN(Y)
```

COS(X) S Returns the cosine of X where X is in radians.

```
300 X = R*COS(A)
```

SIN(X) S Returns the sine of X where X is in radians.

```
400 Y = R*SIN(A)
```

TAN(X) S Returns the tangent of X where X is in radians.

```
500 T = TAN(A)
```

String Functions

CHR$(X) ST Returns a string character with the value of the numeric expression X. The value of X must be between 0 and 255.

```
10 LET T$=CHR$(32) ! T$ = " " (blank)
```

DATE$ ST Returns the current date as an ASCII text string from the terminal's internal calendar.

```
20 LET D$=DATE$[6,11]   !DATE$=TUE, JAN 12, 1982
30 PRINT D$             !Print selected portion of date string
         JAN 12
```

LEN(S$) I Returns the number of ASCII characters in the string S$.

```
10 DIM A$[20]
20 LET A$ = "1234"
30 LET B  = LEN(A$)!  B = 4
```

NUM(S$) I Returns a numeric value between 0 and 255 corresponding to the first character of the string S$. S$ cannot be the null string "".

```
10 LET T$ = "Payroll"
20 LET B  = NUM(T$[2;1])!  B = 97
```

POS(S1$,S2$) I Searches the string S1$ for the first occurrence of the string S2$. Returns the starting index if found, otherwise returns 0.

```
10 LET T$="Television"
20 LET V$ = "vision"
30 LET C = POS(T$,V$) !  C = 5
```

PROC$(X) ST Returns the string representation of the key with a code value of X. If the key is a function key, the appropriate escape sequence is returned. PROC$ cannot process all of the terminal keycodes.

RPT$(S$,X) ST Repeats the string in S$ X times.

```
10 LET T$ = "Ring!"
20 LET E$ = RPT$(T$,3)
E$ = "Ring!Ring!Ring!"
```

TIME$ ST Returns the current time from the terminal's internal clock as an ASCII text string.

```
10 LET K$="TIME SENT"
20 LET H$=TIME$
30 PRINT K$;H$
          TIME SENT 3:45:34 AM
```

TRIM$(S$) ST Returns a string which is equal to S$ with all leading and trailing blanks stripped off.

```
10 LET T$ = "   Pay roll   "
20 LET F$ = TRIM$(T$)
F$ = "Pay roll"
```

UPC$(S$) ST Returns a string made up of the uppercase equivalents of each of the characters in S$.

VERS$ ST Returns the BASIC version identification string printed in the BASIC welcome message.

```
200 N = VAL(N$)+I
```

VAL$(X) ST Returns the value of the numeric expression X expressed as a string of ASCII digits.

```
10 LET T$ = VAL$(128+34) ! T$ = "162"
```

VERS$ Returns the BASIC version identification string printed in the BASIC welcome message.

```
20 LET P$=VERS$
30 PRINT P$
       A-2142-42
```

Print Functions

The following functions can only be used in a PRINT statement. If they are used in a PRINT USING statement they will cause your program to end with an error. The MOVC functions do not return a function value and will cause an error if used in PRINT # statements. The LIN, SPA, and TAB functions will affect the selected file when used in the PRINT # statement.

LIN(N) Skips N lines in the alphanumeric display (current window).

SPA(N) Prints N blanks in the alphanumeric display (current window).

TAB(N) Prints blanks up to column N in the alphanumeric display (current window).

MOVCS(R,C) Moves the cursor to row R, column C. R and C are screen relative coordinates (1,1 to 24,80).

MOVCA(R,C) Moves the cursor to row R, column C. R and C are absolute display memory addresses (1,1 to 255,80).

MOVCR(R,C) Moves the cursor to the current row plus R and the current column plus C. R and C can be positive or negative.

Input/Output Functions

GETKBD(X) I Returns in X the keycode associated with the next key struck. The function value is 1 if a keycode is returned and 0 if no keycode was found. This function can only be executed if the GETKBD flag is set (see GETKBD ON).

DSPIN$(L,X) ST Returns a string from the display, starting at the current cursor position. The absolute value of L is the length of the returned string. If format mode is on, the text returned will be limited to the current unprotected field. If L is negative, character values in the range 128-255 are ignored. If L is positive, characters in the 128-255 range are expanded to their representative escape sequences. In the terminal, characters with values greater than 127 are used to indicate display enhancements, field types, etc. Additional information on these character values is given in the *HP 2647F Reference Manual*.

X must be a numeric variable and will be set to 0 if the end of a terminal data field, the end of the current line, or the end of the display is not encountered before the specified number of characters is read. If the end of a data field or the end of the current line is encountered, X will be set to -1. If the end of the display is reached, X will be set to 1.

GETDCM(S$) I Returns one character from the datacomm without wait. The character is stored in S$. The value of the function is 1 if a character was returned and 0 if there was no character. This function can only be executed if the GETDCM flag is set (see GETDCM ON).

PUTDCM(S$) I Sends the first character in S$ to the datacomm. The function value is 1 if the transfer was successful, 0 if the transfer was unsuccessful, and -1 if the datacomm was busy.

Other

ERRL I Returns the line number in which the last error occurred. See ON ERROR.

ERRN I Returns the error code of the last error. See ON ERROR.

2647A

FRE(expression) I Returns the amount of available memory space. If the expression is numeric, the returned value is the amount of remaining program and variable storage. If the expression is a string, the returned value is the amount of remaining string storage space.

2647F

FRE returns the total amount of available workspace (both numeric and string). The terminal's workspace is fixed at 64K.

ASCII Character Set _____ A

Table A-1. ASCII Character Set

DECIMAL VALUE	GRAPHIC	COMMENTS	ALTERNATE CHARACTER	DECIMAL VALUE	GRAPHIC	COMMENTS
0	Nu	Null	@c	64	@	Commercial at
1	SH	Start of heading	Ac	65	A	Uppercase A
2	Sx	Start of text	Bc	66	B	Uppercase B
3	Ex	End of text	Cc	67	C	Uppercase C
4	ET	End of transmission	Dc	68	D	Uppercase D
5	EQ	Enquiry	Ec	69	E	Uppercase E
6	AK	Acknowledge	Fc	70	F	Uppercase F
7	BL	Bell	Gc	71	G	Uppercase G
8	BS	Backspace	Hc	72	H	Uppercase H
9	HT	Horizontal tabulation	Ic	73	I	Uppercase I
10	LF	Line feed	Jc	74	J	Uppercase J
11	VT	Vertical tabulation	Kc	75	K	Uppercase K
12	FF	Form feed	Lc	76	L	Uppercase L
13	CR	Carriage return	Mc	77	M	Uppercase M
14	SO	Shift out	Nc	78	N	Uppercase N
15	SI	Shift in	Oc	79	O	Uppercase O
16	DL	Data link escape	Pc	80	P	Uppercase P
17	D1	Device control 1 (X-ON)	Qc	81	Q	Uppercase Q
18	D2	Device control 2	Rc	82	R	Uppercase R
19	D3	Device control 3 (X-OFF)	Sc	83	S	Uppercase S
20	D4	Device control 4	Tc	84	T	Uppercase T
21	NK	Negative acknowledge	Uc	85	U	Uppercase U
22	SY	Synchronous idle	Vc	86	V	Uppercase V
23	EB	End of transmission block	Wc	87	W	Uppercase W
24	CN	Cancel	Xc	88	X	Uppercase X
25	EM	End of medium	Yc	89	Y	Uppercase Y
26	SB	Substitute	Zc	90	Z	Uppercase Z
27	EC	Escape	[c	91	[	Opening bracket
28	FS	File separator	\c	92	\	Reverse slant
29	GS	Group separator	]c	93	]	Closing bracket
30	RS	Record separator	^c	94	^	Circumflex
31	US	Unit separator	_c	95	—	Underscore
32		Space (Blank)		96	`	Grave accent
33	!	Exclamation point		97	a	Lowercase a
34	"	Quotation mark		98	b	Lowercase b
35	#	Number sign		99	c	Lowercase c
36	$	Dollar sign		100	d	Lowercase d
37	%	Percent sign		101	e	Lowercase e
38	&	Ampersand		102	f	Lowercase f
39	'	Apostrophe		103	g	Lowercase g
40	(	Opening parenthesis		104	h	Lowercase h
41	)	Closing parenthesis		105	i	Lowercase i
42	*	Asterisk		106	j	Lowercase j
43	+	Plus		107	k	Lowercase k
44	,	Comma		108	l	Lowercase l
45	-	Hyphen (Minus)		109	m	Lowercase m
46	.	Period (Decimal)		110	n	Lowercase n
47	/	Slant		111	o	Lowercase o
48	0	Zero		112	p	Lowercase p
49	1	One		113	q	Lowercase q
50	2	Two		114	r	Lowercase r
51	3	Three		115	s	Lowercase s
52	4	Four		116	t	Lowercase t
53	5	Five		117	u	Lowercase u
54	6	Six		118	v	Lowercase v
55	7	Seven		119	w	Lowercase w
56	8	Eight		120	x	Lowercase x
57	9	Nine		121	y	Lowercase y
58	:	Colon		122	z	Lowercase z
59	;	Semicolon		123	{	Opening (left) brace
60	<	Less than		124	l	Vertical line
61	=	Equals		125	}	Closing (right) brace
62	>	Greater than		126	~	Tilde
63	?	Question mark		127	●	Delete

ASCII Character Set

Table A-2 ASCII (7-Bit) Character Codes

GRAPHIC	DEC	OCT	HEX	GRAPHIC	DEC	OCT	HEX
NUL	0	0	0	@	64	100	40
SOH	1	1	1	A	65	101	41
STX	2	2	2	B	66	102	42
ETX	3	3	3	C	67	103	43
EOT	4	4	4	D	68	104	44
ENQ	5	5	5	E	69	105	45
ACK	6	6	6	F	70	106	46
BEL	7	7	7	G	71	107	47
BS	8	10	8	H	72	110	48
HT	9	11	9	I	73	111	49
LF	10	12	A	J	74	112	4A
VT	11	13	B	K	75	113	4B
FF	12	14	C	L	76	114	4C
CR	13	15	D	M	77	115	4D
SO	14	16	E	N	78	116	4E
SI	15	17	F	O	79	117	4F
DLE	16	20	10	P	80	120	50
DC1	17	21	11	Q	81	121	51
DC2	18	22	12	R	82	122	52
DC3	19	23	13	S	83	123	53
DC4	20	24	14	T	84	124	54
NAK	21	25	15	U	85	125	55
SYN	22	26	16	V	86	126	56
ETB	23	27	17	W	87	127	57
CAN	24	30	18	X	88	130	58
EM	25	31	19	Y	89	131	59
SUB	26	32	1A	Z	90	132	5A
ESC	27	33	1B	[	91	133	5B
FS	28	34	1C	\	92	134	5C
GS	29	35	1D	]	93	135	5D
RS	30	36	1E	^	94	136	5E
US	31	37	1F	_	95	137	5F
SP	32	40	20	`	96	140	60
!	33	41	21	a	97	141	61
"	34	42	22	b	98	142	62
#	35	43	23	c	99	143	63
$	36	44	24	d	100	144	64
%	37	45	25	e	101	145	65
&	38	46	26	f	102	146	66
'	39	47	27	g	103	147	67
(	40	50	28	h	104	150	68
)	41	51	29	i	105	151	69
*	42	52	2A	j	106	152	6A
+	43	53	2B	k	107	153	6B
,	44	54	2C	l	108	154	6C
-	45	55	2D	m	109	155	6D
.	46	56	2E	n	110	156	6E
/	47	57	2F	o	111	157	6F
0	48	60	30	p	112	160	70
1	49	61	31	q	113	161	71
2	50	62	32	r	114	162	72
3	51	63	33	s	115	163	73
4	52	64	34	t	116	164	74
5	53	65	35	u	117	165	75
6	54	66	36	v	118	166	76
7	55	67	37	w	119	167	77
8	56	70	38	x	120	170	78
9	57	71	39	y	121	171	79
:	58	72	3A	z	122	172	7A
;	59	73	3B	{	123	173	7B
<	60	74	3C	:	124	174	7C
=	61	75	3D	}	125	175	7D
>	62	76	3E	~	126	176	7E
?	63	77	3F	∎	127	177	7F

Compatibility_____ B

This appendix lists some of the differences between the BASIC language interpreter used in the terminal and the ANSII standard for minimal BASIC as well as other BASIC interpreters offered by Hewlett-Packard.

- FOR...NEXT loops
- Arithmetic operator hierarchy
- Reserved words
- Default data type
- User defined functions
- RANDOMIZE
- OPTION BASE

FOR...NEXT Loops

BASIC/47 tests the loop variable after the loop has been executed. This means that all FOR...NEXT loops will be executed at least once, even if the loop variable is initially larger than the loop limit value. The ANSII standard specifies that the loop variable will be tested before the loop is executed.

Arithmetic Operator Hierarchy

The ANSII standard does not specify an operator hierarchy. Some BASIC interpreters do define the order in which operators are executed. The hierarchy used by BASIC/47 is given in Section 3.

Reserved Words

BASIC/47 has several "reserved words." These are letter combinations that cannot be used for variable or subprogram names. A list of reserved words is given in Appendix C.

Default Data Type

ANSII standard BASIC does not specify different data types. BASIC/47 uses SHORT as the default data type. If you do not specify the data type of a numeric variable, it will be treated as SHORT. Other BASIC interpreters may use type LONG as the default data type. Data types are discussed in Section 2.

User Defined Functions

BASIC/47 does not allow user defined functions. These are functions which are defined using a DEF statement to create extensions to BASIC.

RANDOMIZE

BASIC/47 does not have a RANDOMIZE statement. The RANDOMIZE statement is used to select a new "seed" or starting point for a series of random numbers. Refer to Section 3 for a description of the random number function (RND).

OPTION BASE

BASIC/47 does not have an OPTION BASE statement. This statement is used to set the starting point for array indexes to "0" or "1". BASIC/47 uses "0" as the first element in an array.

Reserved Words_____C

When the BASIC interpreter is analyzing a newly entered line of program text, it first attempts to determine whether it can recognize keywords. Since a variable name may contain any number of characters (with only the first 15 being significant), variable names cannot contain a keyword in the leading characters of the name. The BASIC interpreter assists you in determining whether a keyword has accidentally crept into the leading characters of a variable name. All text is formatted by the BASIC interpreter such that:

- keywords (also called reserved words) are converted to all uppercase letters.

- variable names are converted to an uppercase letter for the initial letter and all successive letters are forced to lowercase.

Upon listing a program, all unquoted text appears with this format. If an intended variable name appears with more than one uppercase letter at its head, then it contains a keyword in its leading characters. A variable name may contain any keyword within its characters as long as the leading characters do not form a keyword.

Table C-1. List of Reserved Words

ABORTIO	INT	RESUME
ABS	INTEGER	RETURN
AND	KEY	RND
ASSIGN	KEYCDE	RPT$
AUTO	LEN	RUN
BASIC	LET	SAVE
BYTE	LIN	SCR
CALL	LINPUT	SCRATCH
CHR$	LIST	SECURE
CMP	LOCAL	SENDBUS
COMMAND	LOCKOUT	SET
CSAVE	LONG	SGN
CSENA	MERGE	SHORT
CSENS	MOD	SINGLE
DATA	MOVCA	SIZE
DATE$	MOVCR	SLEEP
DELETE	MOVCS	SLOAD
DIM	MULTIPLE	SPA
DIV	NEXT	SQR
DSPIN$	NOT	STATUS
ELSE	NUM	STD
END	OFF	STEP
ENTER	ON	STOP
ERRL	OR	SUB
ERRN	OUTPUT	SUBEND
ERROR	PACK	TAB
EXIT	PASS CONTROL	THEN
EXTEND	POS	TIME$
FOR	PPOLL	TIMEOUT
FRE	PRINT	TO
GET	PROC$	TRIGGER
GETDCM	PROMPT	TRIM$
GETKBD	PUTDCM	UPC$
GO	READ	USER
GO TO	REM	USING
GOSUB	REMOTE	VAL
GOTO	REMOVE	VAL$
IF	RENUM	VERSION$
IMAGE	RESET	WAKEUP
INPUT	RESTORE	XOR

Additional keywords in some versions of BASIC:

ATN	GPMM	PEN
AXES	GRID	PENDN
CLIP	GSTAT	PENUP
CLIPOFF	IPLOT	PLOT
CLIPON	LAXES	PLOTR
COS	LDIR	POINT
CSIZE	LGRID	RPLOT
CURSOR	LIMIT	SCALE
DIGITIZE	LINE	SETAR
DRAW	LOCATE	SETGU
DSIZE	LOG	SETUU
DSTAT	LXAXIS	SHOW
EXP	LYAXIS	SIN
FRAME	MARGIN	TAN
FXD	MOVE	WHERE
GCLR	MSCALE	XAXIS
GPON	PDIR	YAXIS

Summary of BASIC_____D

COMMANDS	
AUTO	Generates statement numbers automatically as program statements are entered.
REMOVE	Removes commands or statements added by the EXTEND command.
GO	Resumes program execution after a break character or STOP statement.
CSAVE	Saves a condensed version of your program on a disc or a cartridge tape.
DELETE	Deletes statements from your program.
EXIT	Exits the BASIC interpreter.
EXTEND	Allows you to add functions, commands, and statements to the BASIC interpreter
GET	Loads your program from a disc or cartridge tape.
LIST	Lists your program.
MERGE	Merges a program on a disc or tape with one in the terminal.
RENUM	Renumbers statements in your program.
RUN	Starts execution of your program.
SAVE	Stores your program on a flexible disc or cartridge tape.
SCRATCH	Deletes the current program from the terminal.
SET	Sets default modes of operation (MULTIPLE, SINGLE, SHORT, LONG, INTEGER, SIZE, PROMPT, and break character).
STATEMENTS	
ASSIGN	Equates a file name with a logical file number.
CALL	Transfers control to a named subprogram.
COMMAND	Allows you to execute any terminal command.
DATA	Holds data accessed by READ statements.
DIM	Sets the size of strings and arrays.
END	Terminate the program.
ERROR	Causes a branch to the error handling routine.
FOR . . . NEXT	Allows you to execute program steps a specified number of times.

GETDCM ON/OFF	Enables and disables the GETDCM function.
GETKBD ON/OFF	Enables and disables the GETKBD function.
GOSUB ... RETURN	Allows you to transfer control to a subroutine and then back to the main program.
GOTO	Causes a branch to the indicated statement.
IF ... THEN ... ELSE	Allows conditional processing, or branching.
IMAGE	Provides format information for PRINT USING statements.
INPUT	Inputs data values to your program from the keyboard.
INTEGER	Declares variables to be of type INTEGER.
KEYCDE	Assigns code values to keys on the terminal keyboard.
LET	Provides an equate or assignment function.
LINPUT	Allows input of a full record from the keyboard.
LINPUT#	Inputs a record from a file.
LONG	Declares variables to be of type LONG.
OFF KEY#	Disables interrupts on the indicated key.
ON END#	Causes a branch when the end of a file is reached.
ON ... GOSUB	Branch to one of several subroutines.
ON ... GOTO	Branch to one of several statements.
ON ERROR	Branch on error conditions.
ON KEY#	Branch when the indicated key is struck.
PACK	Prints formatted data to a string variable.
PRINT	Print values to the display.
PRINT#	Print data to the specified file.
PRINT USING	Print data to the display using a format.
PRINT # USING	Print data to a file using a format.
READ	Input data stored in a program statement.
READ #	Input data stored on a file.
REMARK	Places comments in the program.
RESTORE	Reset the data pointer to the first item in a DATA statement.
RESTORE #	Reset the file pointer to the beginning of a file.
RESUME	Causes program to continue following an error recovery procedure.

SHORT	Declares variables to be of type SHORT.
SLEEP	Causes the BASIC program to be suspended.
STOP	Causes the program to stop.
SUB	Defines the beginning of a subprogram.
SUBEND	Defines the end of a subprogram.
WAKEUP	Reverses the effect of a SLEEP statement.
FUNCTIONS	
ABS(X)	Returns the absolute value of X.
CSENA(R,C)	Returns the absolute alphanumeric cursor position in R and C.
CSENS(R,C)	Returns the screen relative alphanumeric cursor position in R and C.
DATE$	Returns the current date as an ASCII text string from the terminal's internal calendar.
DSPIN$(L,X)	Returns a string from the display, starting at the current cursor position and having length L. X will be set to 0 if the end of display memory is not found during input.
FRE *FRE(X) or FRE(X$) }	Returns the amount of free memory space available.
GETDCM(S$)	Returns a single character from the datacomm in S$.
GETKBD(X)	Returns the code of a single character from the keyboard in X.
INT(X)	Returns the integer value of X.
LEN(S$)	Returns the length of S$.
LIN(X)	Prints X blank lines.
LOG(X)	Returns the natural log of X.
LONG(X)	Converts the value of X to type LONG.
MOVCA(R,C)	Positions the cursor at the absolute coordinates row and column.
MOVCR(R,C)	Positions the cursor at the relative coordinates row and column.
MOVCS(R,C)	Positions the cursor at the screen relative coordinates row and column.
NUM(S$)	Returns the ASCII code value of the first character in S$.
POS(S1$,S2$)	Returns the first character position of S2$ in S1$.
PROC$(X)	Returns the character(s) whose ASCII (or terminal) code is X.
PUTDCM(S$)	Sends the first character in S$ to the datacomm.
RND	Returns a random number between 0 and 1.
RPT$(S$,X)	Repeats the string S$, X times.

*2647A only.

SGN(X)	Returns the sign of the value of X.
SHORT(X)	Changes the value of X to type SHORT.
SIN(X)	Returns the SINE of X.
SPA(X)	Causes X blank characters to be printed.
SQR(X)	Returns the square root of X.
TAB(X)	Causes blank characters to be printed up to column X.
TAN(X)	Return the tangent of X.
TIME$	Returns the current time from the terminal's internal clock as an ASCII text string.
TRIM$(S$)	Returns a string equivalent to S$ with the leading and trailing blanks removed.
UPC$(S$)	Returns a string equivalent to S$ with each of the characters shifted to upper case.
VAL(S$)	Returns a numeric equivalent of the numerals in S$.
VAL$(X)	Returns the numeral equivalent of the value in X.
VERSION$	Returns the BASIC version identification string.

AGL FUNCTIONS

SETUP	
CLIP	Move Soft Clip Limit.
CLIPOFF	Suspend Soft Clip.
CLIPON	Restore Soft Clip.
GCLR	Clear Display.
GPON	Power On Reset Plotter.
LIMIT	Set Hard Clip Limit.
LOCATE	Define Plotting Area.
MARGIN	Define Plotting Area.
MSCALE	Set Up Metric Scaling.
PLOTR	Select and Initialize.
SCALE	Define User Units.
SETAR	Set Aspect Ratio.
SETGU/SETUU	Select GDU's User Units.
SHOW	Define User Units.

PLOTTING

DRAW	Absolute Draw.
IPLOT	Incremental Plot.
LINE	Select Dash Pattern.
MOVE	Absolute Move.
PDIR	Plot Direction — RPLOT/IPLOT.
PEN	Select a "Pen".
PENUP/PENDN	Lift Drop "Pen".
PLOT	Absolute Plotting.
PORG	Set Relocatable Origin.
PRINT #0	Graphics Labeling.
RPLOT	Relative Plotting.

AXIS LABELING

AXES	Draw Linear Axes.
CSIZE	Set Character Size.
FRAME	Outline Soft Clip Area.
FXD	Set Label Format.
GRID	Draw Linear Grid.
LAXES	Draw and Label Axes.
LDIR	Set Label Direction.
LGRID	Draw and Label Grid.
LORG	Set Label Origin Mode.
LXAXIS	Draw Labeled X Axis.
LYAXIS	Draw Labeled Y Axis.
XAXIS	Draw Linear X Axis.
YAXIS	Draw Linear Y Axis.

INTERACTIVE

CURSOR	Read Cursor Position.
DIGITIZE	Read Cursor with Wait.
DSIZE	Return Size to Data.
DSTAT	Display Status.
GSTAT	Graphics Package Status.
POINT	Set Cursor Position.
WHERE	Read Pen Position.

Error Messages _____E

A BASIC program executing in the terminal may generate BASIC, AGL, or command execution errors. These errors will normally cause your program to halt and display an error message. If you use the ON ERROR statement, you can create your own error handling routine. All errors have a decimal code. Table E-1 contains a list of the error codes. The error code number, error message, and a description is given for each error. Refer to Section 10 for a description of error handling. Error Messages are listed in alphabetical order beginning on page E-15.

Table E-1. Terminal Error Codes (in numerical order)

CODE	MESSAGE	EXPLANATION
00257	**COMMAND** INVALID DEVICE SPECIFIED	Incorrectly spelling a device name; correct spelling.
00258	INVALID FILE/ID	File-id passed to F/S intrinsics is incorrect; program error.
00259	INVALID FILE ACCESS	Reading a write only file or writing to read only file.
00260	ACTIVE FILE TABLE FULL	No more files can be opened until another file is closed first.
00261	DUPLICATE DEVICES SPECIFIED	More than one of the same device name in multiple device string specification; delete one.
00262	"SOURCE" = "DESTINATION"	Trying to read and write to the same device; change one of them.
00263	TOO MANY DEVICES SPECIFIED	Can't have more than one source device; can't have more than one destination device for a compare operation.
00264	CONFLICTING I/O	A write to display from I/O buffer is occurring during a GET operation.
00265	DIFF. LENGTH RECORDS	Record lengths on source and destination devices are different on a compare operation.
00266	DIFF. IN BYTE x RECORD y FILE z	A mismatch on a byte on a compare operation.
00267	DIFFERENCE IN RECORD TYPE	Record types did not agree from source and destination devices.
00268	ILLEGAL PARAMETER IN COMMAND	Unrecognized keyword in command parameter list.
00269	EXTRANEOUS PARAMETER IN	Too many keywords in command parameter list.
00270	MISSING PARAMETER IN COMMAND	Expected keyword in command parameter list not found.

Table E-1. Terminal Error Codes (in numerical order) (Continued)

CODE	MESSAGE	EXPLANATION
	COMMAND	
00271	NON-NUMERIC PARAMETER IN	Illegal numeric digit found in command parameter list.
00272	EXCESSIVE NUMERIC PARAMETER IN COMMAND	Numeric digit too large in command parameter list.
00273	ASSIGN TABLE FULL	Can't fit new assignment into assign table; delete another entry to make room.
00274	NAME NOT IN ASSIGN TABLE	Name requested wasn't found in assign table.
00275	RE-ASSIGNMENT NOT ALLOWED	Devices cannot be reassigned in Edit Mode or Data Logging Mode. To reassign devices, disable Edit or Data Logging Mode, reassign, then enable the mode.
00276	ILLEGAL ASSIGN NAME	Incorrect character used in assign name. Names are truncated to 11 characters.
00277	PROGRAM NOT FOUND	Unable to find program with name specified in command channel.
00278	UNRECOGNIZED COMMAND	Unable to find command to match command specified in command channel.
00279	APPLICATION NOT FOUND	Unable to find a running application program.
00280	EXECUTE FILE NOT FOUND	Unable to find an active execute file.
00281	VOLUME TABLE FULL	Too many devices currently active. De-activate one of the devices.
	CARTRIDGE TAPE	
00288	RUNOFF - ON LEFT DRIVE	Tape ran off; remove tape and rethread it.
00289	ABORTED ON LEFT DRIVE	Tape operation aborted; remove tape.
00290	END OF TAPE ON LEFT DRIVE	No more room on tape.

Table E-1. Terminal Error Codes (in numerical order) (Continued)

CODE	MESSAGE	EXPLANATION
00291	**CARTRIDGE TAPE** END OF DATA ON LEFT DRIVE	End of valid data found on tape.
00292	PROTECTED ON LEFT DRIVE	Tape write protected; replace write enable tab.
00293	NO TAPE ON LEFT DRIVE	Tape not inserted.
00294	STALL ON LEFT DRIVE	Tape stalled; remove tape.
00295	READ FAIL ON LEFT DRIVE	Unable to read record from tape after nine retries.
00296	WRITE FAIL ON LEFT DRIVE	Unable to write to tape while in verify mode.
00297	FAIL ON LEFT DRIVE	Tape failed during a CTU test.
00298	END OF FILE ON LEFT DRIVE	End of file reached on tape during file compare operation.
00299	FILE MISSING ON LEFT DRIVE	Missing file mark found during a SKIP or FIND file command.
00304	RUNOFF - ON RIGHT TAPE	Tape ran off; remove tape and rethread it.
00305	ABORTED ON RIGHT TAPE	Tape operation aborted; remove tape.
00306	END OF TAPE ON RIGHT DRIVE	No more room on tape.
00307	END OF DATA ON RIGHT DRIVE	End of valid data found on tape.
00308	PROTECTED ON RIGHT DRIVE	Tape write protected; replace write enable tab.
00309	NO TAPE ON RIGHT DRIVE	Tape not inserted.
00310	STALL ON RIGHT DRIVE	Tape stalled; remove tape.
00311	READ FAIL ON RIGHT DRIVE	Unable to read record from tape after nine retries.
00312	WRITE FAIL ON RIGHT DRIVE	Unable to write to tape while in verify mode.
00313	FAIL ON RIGHT DRIVE	Tape failed during a CTU test.
00314	END OF FILE ON RIGHT DRIVE	End of file reached on tape during file compare operation.
00315	FILE MISSING ON RIGHT DRIVE	Missing file mark found during SKIP or FIND file command.
00336	**PRINTER** PRINT FAIL	General external printer fail; unspecific.
00337	NO PAPER ON EXTERNAL PRINTER	Paper out; resupply paper.
00338	NO EXTERNAL PRINTER	No printer connected, or no PCA board.
00368	NO SHARED PRINTER	No shared printer connected to HP-IB.

Table E-1. Terminal Error Codes (in numerical order) (Continued)

CODE	MESSAGE	EXPLANATION
00369	**PRINTER** PRINTER IS BUSY, RETRY	Shared printer is currently being used; wait until free.
00370	NO READ FROM PRINTER	Can't read from shared printer.
00371	NO PP	Printer did not respond in time.
00352	**DATACOMM** TERMINAL NOT IN REMOTE	Terminal remote switch not down.
00353	DATACOMM ERROR	Unspecific datacomm transmit or receive error.
00384	**HP-IB** HP-IB TIME-OUT	Data transfer didn't complete before allotted time.
00385	NO HP-IB PCA	No HP-IB PCA in terminal.
00386	ILLEGAL HP-IB ADDR	HP-IB address specified did not exist.
00387	HP-IB BUSY, RETRY COMMAND WHEN FREE	Shared HP-IB device currently busy; wait until free.
00388	NO HP-IB CONTROLLER	No terminal on the HP-IB has responded to a request for control.
00389	HP-IB TEST FAIL xx	Self test of HP-IB failed.
00390	HP-IB TEST NOT ATTEMPTED, ADDR=x, SYSCTL=YES or NO, CIC=YES or NO	Self-test of HP-IB cannot be performed under current conditions or configuration.
00391	NOT HP-IB TALKER in BASIC.	Terminal not addressed to talk during a SENDBUS command.
00392	NOT HP-IB SYSTEM CONTROLLER	HP-IB protocol violated; REN and IFC control lines may only be accessed by system controller.
00400	HPIB TE#x TO TE#y TEST FAIL	Transfer from terminal to terminal failed over HP-IB
01026	**BASIC** CHECKSUM ERROR IN BASIC	Something has caused the BASIC interpreter code to be altered. Reload the BASIC interpreter.
01027	NEXT WITHOUT FOR	A NEXT statement is encountered and the corresponding FOR statement is missing or some intervening loop used the same loop variable. Add the appropriate FOR statement or change the loop variable in the intervening loop.
01028	LOOP VARIABLE CAN'T BE LONG	A FOR statement must specify a simple variable of type INTEGER or SHORT. Declare the loop variable to be of type INTEGER or SHORT.

Table E-1. Terminal Error Codes (in numerical order) (Continued)

CODE	MESSAGE	EXPLANATION
01029	**BASIC** RETURN WITHOUT GOSUB	A RETURN statement is encountered and no GOSUB is currently active. Change the program to avoid accidentally falling into a subroutine or remove the offending RETURN.
01030	NO RETURN	The end of program text is encountered while executing a subroutine. Add a RETURN statement.
01031	RESUME WITHOUT ERROR	A RESUME statement is encountered while no error handling routine is being executed. Remove the RESUME statement.
01032	NO RESUME	The end of program text is encountered while executing the error handling routine (see ON ERROR). Add a RESUME statement.
01033	SUB WITHOUT SUBEND	Two SUB statements exist with no intervening SUBEND statement; or a SUB statement exists with no SUBEND statement prior to the end of program text. Add an appropriate SUBEND statement.
01034	MISSING SUB	A SUBEND statement occurs prior to any SUB statement. Add the appropriate SUB statement.
01035	SUBPROGRAM NOT FOUND	A subprogram is referenced in a CALL statement but no corresponding SUB statement is found. Change the CALL or supply the subprogram text.
01036	MISMATCHED QUOTES	A statement or input response or data record for READ # contains a quoted string which is not enclosed in matched quotes. Re-enter the statement or input response; or correct the data record.
01037	MISMATCHED ELSE	More ELSE's occur in a statement than IF...THEN's. Correct the program logic.
01038	SYNTAX ERROR	The statement syntax does not match the program text. Typically, the error occurs because a keyword is misspelled, a comma (or some other punctuation) is inappropriately used or omitted, parentheses are mismatched in an expression, a function call contains extra parameters, etc. Check the manual for the specific syntax for the statement.
01039	MISSING OPERAND	A function or subprogram call requires more parameters; or an expression terminates with an operator; or a READ # statement has no semicolon and/or variable list; or a PRINT # statement has a semicolon and no print list. Correct the syntax in the statement with the error.

Table E-1. Terminal Error Codes (in numerical order) (Continued)

CODE	MESSAGE	EXPLANATION
01040	**BASIC** NUMERIC OVERFLOW	A hex or octal constant exceeds the integer value range; or the results of an arithmetic calculation exceed the allowed value range. Correct the hex or octal constant; or change the program logic to prevent exceeding the appropriate value range.
01041	DIVISION BY ZERO	A division operation has a zero divisor. Change program logic to prevent zero divisors.
01042	UNDEFINED LINE NUMBER	A RESTORE, GOTO, GOSUB, or RESUME statement references a line number which does not exist in the current program unit. The GOTO or GOSUB may be invoked by the ON END, ON ERROR or ON KEY statements. Supply the missing line number or correct an erroneous line number specification.
01043	LINE ALREADY EXISTS	The AUTO command is attempting to overwrite a program line which already exists. Specify the AUTO command so that it does not overwrite already existent program text, or delete the program lines which would be overwritten prior to initiating the AUTO command.
01044	STATEMENT MUST HAVE LINE	A statement which can be executed only as part of a program has been attempted as an unnumbered command. Execute that statement only as part of a program.
01045	STATEMENT MUST BE DIRECT	Execution of a command which is disallowed in a program has been attempted. Remove the command from the program text.
01046	SUBSCRIPT OUT OF RANGE	A subscript value exceeds the limits of an array dimension (which were established upon first reference to that array). Explicitly declare the array with appropriate dimensions, or change the array declaration, or change program logic to avoid exceeding the dimension limits.
01047	DIMENSION MISMATCH	The number of dimensions in an array reference does not match the number of dimensions specified in the first reference to that array; or the number of dimensions in an array passed to a subprogram as an actual parameter does not match the number of dimensions for the corresponding formal parameter. Change any array references which specify an incorrect number of dimensions.
01048	REDECLARED VARIABLE	A declaration statement (DIM, INTEGER, SHORT or LONG) is executed more than once; or a variable occurs in more than one declaration statement. Change program logic to avoid executing declaration statements more than once; or remove extraneous declarations or a variable.

Table E-1. Terminal Error Codes (in numerical order) (Continued)

CODE	MESSAGE	EXPLANATION
01049	**BASIC** TYPE MISMATCH	An attempt is made to assign a string value to a numeric variable or a numeric value to a string variable; or the value or variable passed to a function or subprogram is not of the required type. Change the variable specification; or change the program to pass the correct type of value or variable to a function or subprogram.
01050	STRING TOO LONG	A string expression results in a string longer than 255 characters. Change program logic.
01051	STRING FORMULA TOO COMPLEX	The evaluation of a string expression requires more intermediate results than the BASIC Interpreter allows. Break the string expression evaluation down so that you provide your own intermediate results.
01052	NONCONTIGUOUS STRING	The character position preceding a substring (to which a value is being assigned by a LET statement or a function) is undefined. Prevent assignments to substrings which are not immediately preceded by defined characters.
01053	SUBSTRING DESIGNATOR ERROR	The last character position in a substring specification precedes the first character position; or the last character position in a substring specification exceeds 255. Change the substring specification.
01054	VALUE OUT OF RANGE	A value specified for a command or function parameter lies outside the permitted range, or an octal constant digit is greater than 7. The value may be a line number, a specific type of character (e.g., letter as versus punctuation mark), an integer (frequently positive and less than 256), have a required relationship to another value, etc. For example, the NUM function returns this error if the string argument is null; and the DELETE command returns this error if the last line in the line number range precedes the first line number. Correct the command or function parameter.
01055	VARIABLE/NAME REQUIRED	The syntax requires a variable name (as versus a constant) in a function call; or a statement must start with a variable name if it does not start with some recognized statement initiator (i.e., an implied LET statement is the default statement type); or a subprogram name must follow CALL or SUB. Change the statement to provide variable or subprogram names as required.
01056	FUNCTION USAGE ERROR	A function invocation occurs in a disallowed context; e.g., as the first action in a statement, or a cursor movement function in a PRINT # statement, or a GETKBD function call prior to executing GETKBD ON. See the manual for the proper context for the function usage and correct the context.

Table E-1. Terminal Error Codes (in numerical order) (Continued)

CODE	MESSAGE	EXPLANATION
01057	**BASIC** BUFFER NOT AVAILABLE	Two different files are being accessed by READ # or ENTER statements which have not used all the contents of the last record read for each file and an attempt is made to access a third file through a READ # or ENTER statement. Restructure the file contents of the input files or change the program so that no more than two input files are actively using buffers at the same time.
01058	BUFFER OVERFLOW	The maximum record size of 255 characters in a PRINT # statement was exceeded. Change program logic to prevent records containing more than 255 characters. Or, a statement continued on another line using & contains more than 250 characters. Shorten statement.
01059	LOADER FORMAT ERROR	A record which is supposed to be in the format for the terminal's binary loader fails to match that format. Recreate the file in the binary loader format.
01060	UNASSIGNED/DISALLOWED FILE NUMBER	A READ #, LINPUT #, or PRINT # statement is attempted for a file which has not been specified in an ASSIGN statement; or the file number in an ASSIGN statement is less than 1 or greater than 255. Add an ASSIGN statement for the file number; or change the file number.
01061	DATA INPUT MISSING	A READ, READ # or ENTER statement encounters consecutive commas in the DATA statement, or the input data, or the DATA statement, or input data record ends with a comma. Correct the DATA statement or the input data record so that they do not end in commas and that consecutive commas do not occur.
01062	DATA ERROR	The data in a DATA statement or in an input buffer does not match the requirements of an item in the read list of a READ, READ #, or ENTER statement. Change the data or change the read list so that the data requirements match.
01063	OUT OF DATA	A READ is executed and all DATA statements have already been used; or a READ # statement is executed and end of file is encountered for a file for which no ON END statement was executed in the current program unit. Supply the appropriate DATA or ON END statement.
01064	OUT OF MEMORY* (2647A ONLY)	Insufficient memory exists to store the program text and all variables. Use SET SIZE to obtain more BASIC workspace or shorten the program or change the program structure to require less variable storage.

Table E-1. Terminal Error Codes (in numerical order) (Continued)

CODE	MESSAGE	EXPLANATION
01065	**BASIC** FORMAT ERROR	A format specification is non-existent, or an edit symbol is undefined, or a simple replicator is used with an invalid edit symbol, or a simple replicator exceeds 255, or parentheses are mismatched, or parentheses contain no edit symbols, or a dis-allowed combination of edit symbols is used for one print list item. Check the edit symbol specifi-cations in the manual and correct any misuses.
01066	NOT IN PROGRAM BREAK	GO is attempted when program execution was not halted by a STOP statement or a CONTROL break (default CONTROL-A) or when the program text was altered after being halted by a STOP statement or a CONTROL break. Restart the program.
01067	MULTIPLE STATEMENTS DISALLOWED	The initial entry (i.e., not through GET or MERGE) of a line containing multiple statements is attempted after the SET SINGLE command has disallowed multiple statements per line. Enter the line as separate lines or specify SET MULTIPLE.
01068	FEATURE NOT PRESENT	A statement, command, or function is referenced and the current version of BASIC does not sup-port that operation. This may occur when trying to run a program which was saved by one version of BASIC (e.g., a version of BASIC for which REMOVE STDX has not been specified) with a different version of BASIC (e.g., a version of BASIC for which REMOVE STDX has been specified) which does not support the statement, command, or function referenced. Load the ap-propriate version of BASIC prior to loading and/or executing any BASIC operations.
01069	NO MESSAGE FOR ERROR CODE	The error message routine is invoked (probably by the ERROR statement) and no error message text exists for the specified error code. Issue the direct statement PRINT ERRN, ERRL to determine which error code and program line caused this error message to be displayed. Add custom error message text to the ERROR statement or change the error code which is specified.
01070	NESTING EXCEEDS LIMIT	More than 255 levels of subprograms are cur-rently active.
01071	NOT ALLOWED W/SECURE PGM	Secured programs cannot be listed or saved without the SECURE option.
01072	OUT OF WORKSPACE	Too many variables or too much program text. Shorten strings or arrays, or variable names. Use Integer type numeric data.
01073	DISPLAY MEMORY FULL	12 K of display memory is locked and full.

Table E-1. Terminal Error Codes (in numerical order) (Continued)

CODE	MESSAGE	EXPLANATION
01074	OUT OF FILE DESCRIPTORS	Too many levels of files, expressions, or variables passed as parameters in Call statement.
01075	OUT OF STACK	Program contains too many nested "For" loops and/or too many GOSUBS. Check or alter program logic. Remove STD/STDX.
01076	TOO MANY OPEN FILES	The maximum number of files which may be opened simultaneously is 15.
00449	**FLEXIBLE DISC** FILE NAME NOT FOUND	File name does not exist on the volume, or it exists as a purged file. Use SHow Files to list the unpurged files on the LOG device; use SHow Purged or SHow Purged <filename> to list the purged file(s) on the LOG device.
00450	FILE NAME ALREADY EXISTS	Attempted to Copy, Transfer, or Record data to a file already present on the volume. Either specify a new filename in the command, REName the existing file, or Purge the existing file and execute the command.
00451	ILLEGAL FILE NAME	File name used contained illegal characters. Legal characters are alpha, numeric, underscore (_), or apostrophe (').
00452	FILE SIZE TOO LARGE	Attempted to CReate a file that is too large for the remaining space on the volume. Either PAck the disc to eliminate purged files, use a new volume, or reduce the file size.
00453	VOLUME NAME NOT FOUND	Volume specified in pattern parameter not found.
00454	VOLUME NAME ALREADY EXISTS	Attempted to name a volume with the same name as the volume on the other drive.
00455	ILLEGAL VOLUME NAME	Attempted to use illegal characters in the volume name when using the CReate or REName commands. They must be either alpha or numeric.
00456	VOLUME NAME NOT ALLOWED	Attempted to specify a volume name or disc drive # as part of the new file name when executing the REName File command. Omit the volume name when specifying the new file name.
00457	VOLUME IS WRITE-PROTECTED	The write-enable notch on the flexible disc is covered. Remove the tab covering the notch.
00458	VOLUME HAS CHANGED	The volume was replaced with another volume while a file was open (i.e., while a file-read or a file-write was suspended).

Table E-1. Terminal Error Codes (in numerical order) (Continued)

CODE	MESSAGE	EXPLANATION
00459	**FLEXIBLE DISC** DIRECTORY IS FULL	Attempted to add more than 240 files to a volume. If the volume contains purged files, PAck the disc to gain file entry space.
00460	IMPROPER DIRECTORY DATA	The volume has been removed from the drive with a file open. The file that was left open is now being accessed.
00461	DISC PACK NOT ALLOWED	PACK command was attempted while a file-read or file-write was suspended (file was left open). Usually happens with CTU emulator files. Execute Rewind command to close file. If error occurs while in Edit Mode, exit Edit Mode before executing PAck command.
00462	CREATE VOLUME NOT ALLOWED	Volume has not been purged; purge the volume, then execute the CReate Volume command.
00463	FORMAT VOLUME NOT ALLOWED	Attempted to format an unpurged volume; purge the volume, then execute the FORmat Volume command.
00464	RENAME FILE NOT ALLOWED	The file is open. (i.e., REName file command was attempted while a file-read or file-write was suspended. Probably, a CTU emulator file is still active; execute the Rewind command, or exit Edit Mode, then execute the REName File command.)
00465	RENAME VOLUME NOT ALLOWED	A file is open (i.e., REName Volume command was attempted while a file-read or file-write was suspended. Probably, a CTU emulator file is still active; execute the Rewind command, or exit Edit Mode, then execute the REName Volume command.)
00466	PURGE FILE NOT ALLOWED	The specified file is open (i.e., Purge File command was attempted while a file-read or file-write was suspended. Probably, a CTU emulator file is still active; execute the Rewind command, or exit Edit Mode, then execute the Purge File command.)
00467	PURGE VOLUME NOT ALLOWED	A file is open (i.e., Purge Volume command was attempted while a file-read or file-write was suspended. Probably, a CTU emulator file is still active; execute the Rewind command, or exit Edit Mode, then execute the Purge Volume command.
00468	END OF DISC FILE	Not enough space left in the file for data during an APpend. Copy the file to a new file, then execute the APpend command.

Table E-1. Terminal Error Codes (in numerical order) (Continued)

CODE	MESSAGE	EXPLANATION
00469	**FLEXIBLE DISC** DISC DATA DID NOT VERIFY	The Enable Verify command (Write-backspace-read mode) is active and the data did not write properly after four retries on the disc. The disc may be faulty; insert a new, formatted disc to continue. If the problem continues, call your local Hewlett-Packard Service Representative.
00470	DATA ON VOLUMES DID NOT COMPARE <m>	Data bytes are not the same on the two volumes being compared. <m> is the number of the sector at which COmpare ended. Possible faulty disc. Recopy volume data to a new disc.
00471	BAD DISC UNIT	"DISC#" was not used or misspelled, or no disc number given in the command sequence.
00472	ILLEGAL UNIT NUMBER	A character other than "1" or "2" was used with "DISC #".
00473	DISC SEEK FAIL	Door may not be closed on drive; close door. Disc may be faulty; use new disc. If problem continues, call your local HP Service Representative.
00474	DISC WRITE FAIL	Disc may be faulty or hardware failure. Use new disc. If problem continues, call your local HP Service Representative.
00475	DISC READ FAIL	Disc may be faulty or hardware failure. Use new disc. If problem continues, call your local HP Service Representative.
00476	DISC FORMAT FAIL	Disc may be faulty or hardware failure. Use new disc. If problem continues, call your local HP Service Representative.
00477	DISC VERIFY FAIL	The track address data or the track CRC data on the volume did not verify while executing the VERify Volume command. Possible faulty disc. Data on the volume can be recovered only by a qualified HP Service Representative using the DIAGNOSTIC commands.
00478	DISC INITIALIZE FAIL	The disc controller failed to mark a bad track. Probable hardware failure; call your local HP Service Representative.
00479	DISC CONTROLLER TIME-OUT	Disc drive not connected to terminal, or controller PCA not present in terminal. Check cable connections; check presence of controller in terminal.
00480	BAD DISC DSJ	Possible hardware failure; call your local HP Service Representative.

Table E-1. Terminal Error Codes (in numerical order) (Continued)

CODE	MESSAGE	EXPLANATION
00481	ILLEGAL DISC ADDRESS	Disc may be faulty or hardware failure. Use new formatted disc. If problem continues, call your local HP Service Representative.
00482	NO DISC CONTROLLER ATTACHED	Disc controller PCA not present in terminal.
00483 00484	DISC SELF TEST FAIL	Refer to 2647F *Reference Manual*, Self-Test Chapter for details.
00485	DISC CONTROL BLOCK NOT AVAILABLE	Diagnostic message for use by HP Service Representative.
00488	BAD BUFFER POINTER	Diagnostic message for use by HP Service Representative.
00489	ILLEGAL DATE	Attempted to use incorrect format for date; use <mm>/<dd>/<yy> in SET Date command.
00490	DIAL NOT ALLOWED	Attempted to execute DIAL command with the modem not connected, the incorrect datacomm PCA installed, or the multipoint ROMs installed.
00491	PHONE CONNECTION NOT ESTABLISHED	Autodial was unsuccessful (i.e., computer modem did not answer). Phone line may be busy, or computer may be down.
00492	ILLEGAL INTRINSIC CALL	(Diagnostic message for the HP Service Representative.)
00495	COPY VOLUME NOT ALLOWED	The "to" volume has not been purged. Purge the volume to remove the volume name, then execute the Copy Volume command.
00496	FILE TYPES CLASH DURING COMPARE	You tried to compare an ASCII file with a binary file. (Files must be of the same type.)
00497	NO DISC IN DRIVE	Disc drive not loaded or disc in drive not accessible.
00498	NO MORE DISC SPACE	The volume is full. If the volume contains purged files, PAck the volume to make more space available; otherwise, replace it with another volume that has more space.
00499	FILE ALREADY OPEN	File may only be opened once. Tape Emulator files remain open until a REWIND, MARK FILE, FIND FILE, or READ (to end of file) is executed.
00500	BAD DISC FORMAT	Available volumes are either unformatted (nonfmt) or unlabeled (nolabl).
00501	UNEXPECTED END OF FILE	Incorrectly formatted disc file. Occurs if disc is removed while writing.

Table E-1. Terminal Error Codes (in numerical order) (Continued)

CODE	MESSAGE	EXPLANATION
01281	**AGL** AGL ERROR 1281	Syntax error.
01282	AGL ERROR 1282	Too many parameters.
01283	AGL ERROR 1283	Cannot default parameter.
01284	AGL ERROR 1284	Driver not present.
01285	AGL ERROR 1285	Parameter out of range.
01286	AGL ERROR 1286	Illegal action request.
01287	AGL ERROR 1287	Cannot open new device without closing old device.
01288	AGL ERROR 1288	No devices on.
01289	AGL ERROR 1289	Wrong number of parameters.
01290	AGL ERROR 1290	Null area specified.
01291	AGL ERROR 1291	Scaling values are equal.
01292	2647 DRIVER ERROR 1292	Attempt to plot beyond limits of logical address space.
01299	2647 DRIVER ERROR 1299	No graphics hardware installed.
01293	**PLOTTER** PLOTTER DRIVER ERROR 1293	Instruction not recognized.
01294	PLOTTER DRIVER ERROR 1294	Wrong number of parameters.
01295	PLOTTER DRIVER ERROR 1295	Bad parameter.
01296	PLOTTER DRIVER ERROR 1296	Illegal character.
01297 (0511)	PLOTTER DRIVER ERROR 1297	Unknown character set.
01298	PLOTTER DRIVER ERROR 1298	Position overflow.

Since error messages appear on the screen without the corresponding error code number, the following listing provides a quick alphabetic reference to the appropriate explanation for each error as well as a reference to the corresponding error number. The messages are grouped according to category (i.e., AGL, BASIC, DATACOMM, DISC, etc.) and these categories are themselves in alphabetical order.

Table E-2. Terminal Error Codes (in alphabetic order)

MESSAGE	CODE	EXPLANATION
AGL ERROR 1281	01281	Syntax error.
AGL ERROR 1282	01282	Too many parameters.
AGL ERROR 1283	01283	Cannot default parameter.
AGL ERROR 1284	01284	Driver not present.
AGL ERROR 1285	01285	Parameter out of range.
AGL ERROR 1286	01286	Illegal action request.
AGL ERROR 1287	01287	Cannot open new device without closing old device.
AGL ERROR 1288	01288	No devices on.
AGL ERROR 1289	01289	Wrong number of parameters.
AGL ERROR 1290	01290	Null area specified.
AGL ERROR 1291	01291	Scaling values are equal.
2647 DRIVER ERROR 1292	01292	Attempt to plot beyond limits of logical address space.
2647 DRIVER ERROR 1299	01299	No graphics hardware installed.
BASIC BUFFER NOT AVAILABLE	01057	Two different files are being accessed by READ # or INPUT statements which have not used all the contents of the last record read for each file and an attempt is made to access a third file through a READ # or INPUT statement. Restructure the file contents of the input files or change the program so that no more than two input files are actively using buffers at the same time.
BUFFER OVERFLOW	01058	The maximum record size of 255 characters in a PRINT # statement was exceeded. Change program logic to prevent records containing more than 255 characters.
CHECKSUM ERROR IN BASIC	01026	Something has caused the BASIC interpreter code to be altered. Reload the BASIC interpreter.

Table E-2. Terminal Error Codes (in alphabetical order) (Continued)

MESSAGE	CODE	EXPLANATION
DATA ERROR	01062	The data in a DATA statement or in an input buffer does not match the requirements of an item in the read list of a READ, READ #, or INPUT statement. Change the data or change the read list so that the data requirements match.
DATA INPUT MISSING	01061	A READ, READ # or INPUT statement encounters consecutive commas in the DATA statement or the input data or the DATA statement or input data record ends with a comma. Correct the DATA statement or the input data record so that they do not end in commas and that consecutive commas do not occur.
DIMENSION MISMATCH	01047	The number of dimensions in an array reference does not match the number of dimensions specified in the first reference to that array; or the number of dimensions in an array passed to a subprogram as an actual parameter does not match the number of dimensions for the corresponding formal parameter. Change any array references which specify an incorrect number of dimensions.
DISPLAY MEMORY FULL	01073	Display memory (12K) is locked and full.
DIVISION BY ZERO	01041	A division operation has a zero divisor. Change program logic to prevent zero divisors.
EXTRA IGNORED		Too many items entered in response to INPUT prompt (?). Entering too few items generates double prompt (??).
FEATURE NOT PRESENT	01068	A statement, command, or function is referenced and the current version of BASIC does not support that operation. This may occur when trying to run a program which was saved by one version of BASIC (e.g., a version of BASIC for which REMOVE STDX has not been specified) with a different version of BASIC (e.g., a version of BASIC for which REMOVE STDX has been specified) which does not support the statement, command, or function referenced. Load the appropriate version of BASIC prior to loading and or executing any BASIC operations.
FORMAT ERROR	01065	A format specification is non-existent, or an edit symbol is undefined, or a simple replicator is used with an invalid edit symbol, or a simple replicator exceeds 255 characters, or parentheses are mismatched, or parentheses contain no edit symbols, or a disallowed combination of edit symbols is used for one print list item. Check the edit symbol specifications in the manual and correct any misuses.

Table E-2. Terminal Error Codes (in alphabetical order) (Continued)

MESSAGE	CODE	EXPLANATION
FUNCTION USAGE ERROR	01056	A function invocation occurs in a disallowed context (e.g., as the first action in a statement, or a cursor movement function in a PRINT # statement, or a GETKBD function call prior to executing GETKBD ON. See the manual for the proper context for the function usage and correct the context.
LINE ALREADY EXISTS	01043	The AUTO command is attempting to overwrite a program line which already exists. Specify the AUTO command so that it does not overwrite already existent program text or delete the program lines which would be overwritten prior to initiating the AUTO command.
LOADER FORMAT ERROR	01059	A record which is supposed to be in the format for the terminal's binary loader fails to match that format. Recreate the file in the binary loader format.
LOOP VARIABLE CAN'T BE LONG	01028	A FOR statement must specify a simple variable of type INTEGER or SHORT. Declare the loop variable to be of type INTEGER or SHORT.
MISMATCHED ELSE	01037	More ELSE's occur in a statement than IF...THEN's. Correct the program logic.
MISMATCHED QUOTES	01036	A statement or input response or data record for READ # contains a quoted string which is not enclosed in matched quotes. Re-enter the statement or input response; or correct the data record.
MISSING OPERAND	01039	A function or subprogram call requires more parameters; or an expression terminates with an operator; or a READ # statement has no semicolon and/or variable list; or a PRINT # statement has a semicolon and no print list. Correct the syntax in the statement with the error.
MISSING SUB	01034	A SUBEND statement occurs prior to any SUB statement. Add the appropriate SUB statement.
MULTIPLE STATEMENTS	01067	The initial entry (i.e., not through GET or MERGE) of a line containing multiple statements is attempted after the SET SINGLE command has disallowed multiple statements per line. Enter the line as separate lines or specify SET MULTIPLE.
NESTING EXCEEDS LIMIT	01070	More than 255 levels of subprograms are currently active.

Table E-2. Terminal Error Codes (in alphabetical order) (Continued)

MESSAGE	CODE	EXPLANATION
NEXT WITHOUT FOR	01027	A NEXT statement is encountered and the corresponding FOR statement is missing or some intervening loop used the same loop variable. Add the appropriate FOR statement or change the loop variable in the intervening loop.
NO MESSAGE FOR ERROR	01069	The error message routine is invoked (probably by the ERROR statement) and no error message text exists for the specified error code. Issue the direct statement PRINT ERRN, ERRL to determine which error code and program line caused this error message to be displayed. Add custom error message text to the ERROR statement or change the error code which is specified.
NO RESUME	01032	The end of program text is encountered while executing the error handling routine (see ON ERROR). Add a RESUME statement.
NO RETURN	01030	The end of program text is encountered while executing a subroutine. Add a RETURN statement.
NONCONTIGUOUS STRING	01052	The character position preceding a substring (to which a value is being assigned by a LET statement or a function) is undefined. Prevent assignments to substrings which are not immediately preceded by defined characters.
NOT ALLOWED W/SECURE PGM	01071	Secured programs cannot be listed or saved without the SECURE option.
NOT IN PROGRAM BREAK	01066	GO is attempted when program execution was not halted by a STOP statement or a CONTROL break (default CONTROL-A) or when the program text was altered after being halted by a STOP statement or a CONTROL break. Restart the program.
NUMERIC OVERFLOW	01040	A hex or octal constant exceeds the integer value range; or the results of an arithmetic calculation exceed the allowed value range. Correct the hex or octal constant; or change the program logic to prevent exceeding the appropriate value range.
OUT OF DATA	01063	A READ is executed and all DATA statements have already been used; or a READ # statement is executed and end of file is encountered for a file for which no ON END statement was executed in the current program unit. Supply the appropriate DATA or ON END statement.
OUT OF FILE DESCRIPTORS	01074	Too many levels of files, expressions, or variables passed as parameters in Call statement.

Table E-2. Terminal Error Codes (in alphabetical order) (Continued)

MESSAGE	CODE	EXPLANATION
OUT OF MEMORY (2647A only)	01064	Insufficient memory exists to store the program and all variables. Use SET SIZE to obtain more BASIC workspace or shorten the program or change the program structure to require less variable storage.
OUT OF STACK (2647F only)	01075	Program contains too many nested FOR loops or too many GOSUBS. Check or alter program logic. Remove STD/STDX.
OUT OF WORKSPACE (2647F only)	01072	Too many variables or too much program text. Shorten strings or arrays or variable names. Use integer type numeric data.
REDECLARED VARIABLE	01048	A declaration statement (DIM, INTEGER, SHORT or LONG) is executed more than once; or a variable occurs in more than one declaration statement. Change program logic to avoid executing declaration statements more than once; or remove extraneous declarations or a variable.
REDO FROM START		Wrong type of data (numeric or string) was entered in response to INPUT prompt. Reenter all items in data list.
RESUME WITHOUT ERROR	01031	A RESUME statement is encountered while no error handling routine is being executed. Remove the RESUME statement.
RETURN WITHOUT GOSUB	01029	A RETURN statement is encountered and no GOSUB is currently active. Change the program to avoid accidentally falling into a subroutine or remove the offending RETURN.
STATEMENT MUST BE DIRECT	01045	Execution of a command which is disallowed in a program has been attempted. Remove the command from the program text.
STATEMENT MUST HAVE LINE	01044	A statement which can be executed only as part of a program has been attempted as an unnumbered command. Execute the statement only as part of a program.
STRING FORMULA TOO COMPLEX	01051	The evaluation of a string expression requires more intermediate results than the BASIC interpreter allows. Break the string expression evaluation down so that you provide your own intermediate results.
STRING TOO LONG	01050	A string expression results in a string longer than 255 characters. Change program logic.

Table E-2. Terminal Error Codes (in alphabetical order) (Continued)

MESSAGE	CODE	EXPLANATION
SUB WITHOUT SUBEND	01033	Two SUB statements exist with no intervening SUBEND statement; or a SUB statement exists with no SUBEND statement prior to the end of program text. Add an appropriate SUBEND statement.
SUBPROGRAM NOT FOUND	01035	A subprogram is referenced in a CALL statement, but no corresponding SUB statement is found. Change the CALL or supply the subprogram text.
SUBSCRIPT OUT OF RANGE	01046	A subscript value exceeds the limits of an array dimension (which were established upon first reference to that array). Explicitly declare the array with appropriate dimensions, or change the array declaration, or change program logic to avoid exceeding the dimension limits.
SUBSTRING DESIGNATOR ERROR	01053	The last character position in a substring specification precedes the first character position; or the last character position in a substring specification exceeds 255. Change the substring specification.
SYNTAX ERROR	01038	The statement syntax does not match the program text. Typically, the error occurs because a keyword is misspelled, a comma (or some other punctuation) is inappropriately used or omitted, parentheses are mismatched in an expression, a function call contains extra parameters, etc. Check the manual for the specific syntax for the statement.
TOO MANY OPEN FILES	01076	The maximum number of files which may be open at one time is 15.
TYPE MISMATCH	01049	An attempt is made to assign a string value to a numeric variable; or the value or variable passed to a function or subprogram is not of the required type. Change the variable specification; or change the program to pass the correct type of value or variable to a function or subprogram.
UNASSIGNED/DISALLOWED FILE NUMBER	01060	A READ #, LINPUT #, or PRINT # statement is attempted for a file which has not been specified in an ASSIGN statement; or the file number in an ASSIGN statement is less than 1 or greater than 255. Add an ASSIGN statement for the file number; or change the file number.
UNDEFINED LINE NUMBER	01042	A RESTORE, GOTO, GOSUB, or RESUME statement references a line number which does not exist in the current program unit. The GOTO or GOSUB may be invoked by the ON END, ON ERROR, or ON KEY statements. Supply the missing line number or correct an erroneous line number specification.

Table E-2. Terminal Error Codes (in alphabetical order) (Continued)

MESSAGE	CODE	EXPLANATION
VALUE OUT OF RANGE	01054	A value specified for a command or function parameter lies outside the permitted range, or an octal constant digit is greater than 7. The value may be a line number, a specific type of character (e.g., letter as versus punctuation mark), an integer (frequently positive and less than 256), have a required relationship to another value, etc. For example, the NUM function returns this error if the string argument is null; and the DELETE command returns this error if the last line in the line number range precedes the first line number. Correct the command or function parameter.
VARIABLE NAME REQUIRED	01055	The syntax requires a variable name (as versus a constant) in a function call; or a statement must start with a variable name if it does not start with some recognized statement initiator (i.e., an implied LET statement is the default statement type); or a subprogram name must follow CALL or SUB. Change the statement to provide variable or subprogram names as required.
COMMAND ACTIVE FILE TABLE FULL	00260	No more files can be opened until another file is closed first.
APPLICATION NOT FOUND	00279	Unable to find a running application program.
ASSIGN TABLE FULL	00273	Can't fit new assignment into assign table; delete another entry to make room.
CONFLICTING I/O	00264	A write to display from I/O buffer is occurring during a GET operation.
DIFF. IN BYTE x RECORD y FILE z	00266	A mismatch on a byte during a compare operation.
DIFF. IN RECORD TYPE	00267	Record types did not agree from source and destination devices.
DIFF. LENGTH RECORDS	00265	Record lengths on source and destination devices are different on a compare operation.
DUPLICATE DEVICES SPECIFIED	00261	More than one of the same device name in multiple device string specification; delete one.
EXCESSIVE NUMERIC PARAMETER IN COMMAND	00272	Numeric digit too large in command parameter list.
EXECUTE FILE NOT FOUND	00280	Unable to find an active execute file.
EXTRANEOUS PARAMETER IN	00269	Too many keywords in command parameter list.
ILLEGAL ASSIGN NAME	00276	Incorrect character used in assign name. Names are truncated to 11 characters.
ILLEGAL PARAMETER IN COMMAND	00268	Unrecognized keyword in command parameter list.

Table E-2. Terminal Error Codes (in alphabetical order) (Continued)

MESSAGE	CODE	EXPLANATION
INVALID DEVICE SPECIFIED	00257	Incorrectly spelling a device name; correct spelling.
INVALID FILE ACESS	00259	Reading a write only file or writing to a read only file.
INVALID FILE/ID	00258	File-id passed to F/S intrinsics is incorrect; program error.
MISSING PARAMETER IN COMMAND	00270	Expected keyword in command parameter list not found.
NAME NOT IN ASSIGN TABLE	00274	Name requested wasn't found in assign table.
NON-NUMERIC PARAMETER IN	00271	Illegal numeric digit found in command parameter list.
PROGRAM NOT FOUND	00277	Unable to find program with name specified in command channel.
RE-ASSIGNMENT NOT ALLOWED	00275	Devices cannot be reassigned in Edit Mode or Data Logging Mode. To reassign devices, disable Edit or Data Logging Mode, reassign, then enable the mode.
''SOURCE'' = ''DESTINATION''	00262	Trying to read and write to the same device; change one of them.
TOO MANY DEVICES SPECIFIED	00263	Can't have more than one source device; can't have more than one destination device for a compare operation.
UNRECOGNIZED COMMAND	00278	Unable to find command to match command specified in command channel.
VOLUME TABLE FULL	00281	Too many devices currently active. De-activate one of the devices.
DATACOMM		
DATACOMM ERROR	00353	Unspecific datacomm transmit or receive error.
TERMINAL NOT IN REMOTE	00352	Terminal remote switch not down.
DISC		
BAD BUFFER POINTER	00488	Diagnostic message for use by HP Service Representative.
BAD DISC DSJ	00480	Possible hardware failure; call your local HP Service Representative.
BAD DISC FORMAT	00500	Available volumes are either unformatted (nonfmt) or unlabeled (nolabl).
BAD DISC UNIT	00471	"DISC #" was not used or was misspelled, or no disc number given in the command sequence.

Table E-2. Terminal Error Codes (in alphabetical order) (Continued)

MESSAGE	CODE	EXPLANATION
COPY VOLUME NOT ALLOWED	00495	The "to" volume has not been purged. Purge the volume to remove the volume name, then execute the Copy Volume command.
CREATE VOLUME NOT ALLOWED	00462	Volume has not been purged; purge the volume, then execute the CReate Volume command.
DATA ON VOLUMES DID NOT COM-PARE <m>	00470	Data bytes are not the same on two volumes while executing the COmpare Volume command. The <m> parameter reports the number of the sector at which the compare ended. Possible faulty disc. Recopy volume data to a new disc.
DIAL NOT ALLOWED	00490	Attempted to execute DIAL command with the modem not connected, the incorrect datacomm PCA installed, or the multipoint ROMs installed.
DIRECTORY IS FULL	00459	Attempted to add more than 240 files to a volume. If the volume contains purged files, PAck the disc to gain file entry space.
DISC CONTROL BLOCK NOT AVAILABLE	00485	Diagnostic message for use by HP Service Representative.
DISC CONTROLLER TIME-OUT	00479	Disc drive not connected to terminal, or controller PCA not present in terminal. Check cable connections; check presence of controller in terminal.
DISC DATA DID NOT VERIFY	00469	The Enable Verify command (write-backspace-read mode) is active and the data did not write properly after four retries on the disc. The disc may be faulty; insert a new, formatted disc to continue. If problem continues, call your local HP Service Representative.
DISC FORMAT FAIL	00476	Disc may be faulty or hardware failure. Use new disc. If problem continues, call your local HP Service Representative.
DISC INITIALIZE FAIL	00478	The disc controller failed to mark a bad track. Probable hardware failure; call your local HP Service Representative.
DISC PACK NOT ALLOWED	00461	A file is open (i.e., PACK command was attempted while a file-read or a file-write was suspended.) Probably a CTU emulator file is still active. Execute the Rewind command, or exit Edit Mode, then execute the PAck command.
DISC READ FAIL	00475	Disc may be faulty or hardware failure. Use new disc; if problem continues, call your local HP Service Representative.
DISC SEEK FAIL	00473	Door may not be closed on drive; close door. Disc may be faulty or hardware failure. Use new disc. If problem continues, call your local HP Service Representative.

Table E-2. Terminal Error Codes (in alphabetical order) (Continued)

MESSAGE	CODE	EXPLANATION
DISC SELF TEST FAIL	00483 00484	Refer to *2647F Reference Manual*, Self-Test Chapter, for more details.
DISC VERIFY FAIL	00477	The track address data or the track CRC data on the volume did not verify while executing the VERify Volume command. Possible faulty disc. Data on the volume can be recovered only by a qualified HP Service Representative.
DISC WRITE FAIL	00474	Disc may be faulty or hardware failure. Use new disc. If problem continues, call your local HP Service Representative.
END OF DISC FILE	00468	Not enough space left in the file for data during an APpend operation. Copy the file data to a new file, then execute the APpend command.
FILE ALREADY OPEN	00499	File may only be opened once. Cartridge tape emulator files remain open until a REWIND, MARK FILE, FIND FILE, or READ (to end of data) command is executed.
FILE NAME ALREADY EXISTS	00450	Attempted to Copy, Transfer, or Record data to a file already present on the volume. Specify a new filename in the command, REName the existing file, or Purge the existing file and execute the command.
FILE NAME NOT FOUND	00449	File name does not exist on the volume, or it exists as a purged file. Use SHow Files to list the unpurged files on the LOG device; use the SHow Purged or SHow Purged <filename> commands to list the purged file(s) on the LOG device.
FILE SIZE TOO LARGE	00452	Attempted to CReate a file that is too large for the remaining space on the volume. PACK the disc if there are any purged files, use a new volume, or reduce the file size.
FILE TYPE CLASH DURING COMPARE	00496	You tried to compare an ASCII file with a binary file. Files must be of the same type.
FORMAT VOLUME NOT ALLOWED	00463	Attempted to format an unpurged volume; purge the volume, then execute the FORmat Volume command.
ILLEGAL DATE	00489	Attempted incorrect format for date; use <mm>/<dd>/<yy> in SET Date command.
ILLEGAL DISC ADDRESS	00481	Disc may be faulty or hardware failure. Use new formatted disc. If problem continues, call your local HP Service Representative.
ILLEGAL FILE NAME	00451	Attempted to use illegal characters in the file name; legal characters must be alpha, numeric, underscore (_), or apostrophe (').

Table E-2. Terminal Error Codes (in alphabetical order) (Continued)

MESSAGE	CODE	EXPLANATION
ILLEGAL INTRINSIC CALL	00492	Diagnostic message for use by HP Service Representative.
ILLEGAL UNIT NUMBER	00472	A character other than "1" or "2" was used with "DISC #"
ILLEGAL VOLUME NAME	00455	Attempted to use illegal characters in the volume name when using the CReate or REName commands. They must be either alpha or numeric.
IMPROPER DIRECTORY DATA	00460	The volume has been removed from the drive with a file open. The file that was left open is now being accessed.
NO DISC CONTROLLER	00482	Disc controller PCA not present in terminal.
NO DISC IN DRIVE	00497	Disc drive not loaded or disc in drive not accessible.
NO MORE DISC SPACE	00498	The volume is full. If the volume contains purged files, PAck the volume to make more space available; otherwise, replace it with another volume that has more space.
PHONE CONNECTION NOT ESTABLISHED	00491	Autodial was unsuccessful (i.e., computer modem did not answer). Phone line may be busy, or computer may be down.
PURGE FILE NOT ALLOWED	00466	The specified file is open (i.e., Purge File command was attempted while a file-read or a file-write was suspended. Probably, a CTU emulator file is still active; execute the Rewind command, or exit Edit Mode, then execute the Purge File command.)
PURGE VOLUME NOT ALLOWED	00467	A file is open (i.e., Purge Volume command was attempted while a file-read or file-write was suspended. Probably, a CTU emulator file is still active; execute the Rewind command, or exit Edit Mode, then execute the Purge Volume command.)
RENAME FILE NOT ALLOWED	00464	A file is open. (i.e., REName File command was attempted while a file-read or file-write was suspended. Probably, a CTU emulator file is still active; execute the Rewind command, or exit Edit Mode, then execute the REName File command.)
RENAME VOLUME NOT ALLOWED	00465	A file is open. (i.e., REName Volume command was attempted while a file-read or file-write was suspended. Probably, a CTU emulator file is still active. Execute the Rewind command, or exit Edit Mode, then execute the REName volume command.)

Table E-2. Terminal Error Codes (in alphabetical order) (Continued)

MESSAGE	CODE	EXPLANATION
UNEXPECTED END OF FILE	00501	Incorrectly formatted disc file. Occurs if disc is removed while writing.
VOLUME HAS CHANGED	00458	The volume was replaced with another volume while a file was open (i.e., while a file-read or file-write was suspended).
VOLUME IS WRITE-PROTECTED	00457	The write-enable notch on the flexible disc is covered. Remove the tab covering the notch.
VOLUME NAME ALREADY EXISTS	00454	Attempted to name a volume with the same name as the volume on the other drive.
VOLUME NAME NOT ALLOWED	00456	Attempted to specify a volume name or disc # as part of the new file name when executing the RE-Name File command. Omit the volume name when specifying the new file name.
VOLUME NAME NOT FOUND	00453	Volume specified in pattern parameter not found.
HP-IB HP-IB BUSY RETRY COMMAND WHEN FREE	00387	Shared HP-IB device currently busy; wait until free.
HP-IB TE#x TO TE#y TEST FAIL	00400	Transfer from terminal to terminal failed over HP-IB.
HP-IB TEST FAIL xx	00389	Self-test of HP-IB failed.
HP-IB TEST NOT ATTEMPTED ADDR=X, SYSCTL=YES or NO, CIC=YES or NO	00390	Self-test of HP-IB cannot be performed under current conditions or configuration.
HP-IB TIME-OUT	00384	Data transfer didn't complete before allotted time.
ILLEGAL HP-IB ADDR	00386	HP-IB address specified did not exist.
NO HP-IB CONTROLLER	00388	No terminal on the HP-IB has responded to a request for control.
NO HP-IB PCA	00385	No HP-IB PCA in terminal.
NOT HP-IB SYSTEM CONTROLLER	00392	HP-IB protocol violated; REN and IFC control lines may only be accessed by system controller.
NOT HP-IB TALKER IN BASIC	00391	Terminal not addressed to talk during a SENDBUS command.
PLOTTER PLOTTER DRIVER ERROR 1293	01293	Instruction not recognized.
PLOTTER DRIVER ERROR 1294	01294	Wrong number of parameters.
PLOTTER DRIVER ERROR 1295	01295	Bad parameter.

Table E-2. Terminal Error Codes (in alphabetical order) (Continued)

MESSAGE	CODE	EXPLANATION
PLOTTER DRIVER ERROR 1296	01296	Illegal character.
PLOTTER DRIVER ERROR 1297	01297	Unknown character set.
PLOTTER DRIVER ERROR 1298	01298	Position overflow.
PRINTER		
NO EXTERNAL PRINTER	00338	No printer connected, or no PCA board.
NO PAPER ON EXTERNAL PRINTER	00337	Paper out; resupply paper.
NO PP	00371	Printer did not respond in time.
NO READ FROM PRINTER	00370	Can't read from shared printer.
NO SHARED PRINTER	00368	No shared printer connected to HP-IB.
PRINT FAIL	00336	General external printer fail; unspecific.
PRINTER IS BUSY, RETRY	00369	Shared printer is currently being used; wait until free.
TAPE		
ABORTED ON LEFT DRIVE	00289	Tape operation aborted; remove tape.
ABORTED ON RIGHT TAPE	00305	Tape operation aborted; remove tape.
END OF DATA ON LEFT DRIVE	00290	End of valid data found on tape.
END OF DATA ON RIGHT DRIVE	00307	End of valid data found on tape.
END OF FILE ON LEFT DRIVE	00298	End of file is reached on tape during file compare operation.
END OF FILE ON RIGHT DRIVE	00314	End of file reached on tape during file compare operation.
END OF TAPE ON LEFT DRIVE	00290	No more room on tape.
END OF TAPE ON RIGHT DRIVE	00306	No more room on tape.
FAIL ON LEFT DRIVE	00297	Tape failed during a CTU test.
FAIL ON RIGHT DRIVE	00313	Tape failed during a CTU test.
FILE MISSING ON LEFT DRIVE	00299	Missing file mark found during a SKIP or FIND file command.
FILE MISSING ON RIGHT DRIVE	00315	Missing file mark found during a SKIP or FIND file command.
NO TAPE ON LEFT DRIVE	00293	Tape not inserted.
NO TAPE ON RIGHT DRIVE	00308	Tape not inserted.

Table E-2. Terminal Error Codes (in alphabetical order) (Continued)

MESSAGE	CODE	EXPLANATION
PROTECTED ON LEFT DRIVE	00292	Tape write protected; replace write enable tab.
PROTECTED ON RIGHT DRIVE	00308	Tape write protected; replace write enable tab.
READ FAIL ON LEFT DRIVE	00295	Unable to read record from tape after nine retries.
READ FAIL ON RIGHT DRIVE	00311	Unable to read record from tape after nine retries.
RUNOFF - ON LEFT DRIVE	00288	Tape ran off; remove tape and rethread it.
RUNOFF - ON RIGHT DRIVE	00304	Tape ran off; remove tape and rethread it.
STALL ON LEFT DRIVE	00294	Tape stalled; remove tape.
STALL ON RIGHT DRIVE	00310	Tape stalled; remove tape.
WRITE FAIL ON LEFT DRIVE	00296	Unable to write to tape while in verify mode.
WRITE FAIL ON RIGHT DRIVE	00312	Unable to write to tape while in verify mode.

Graphics Applications Programs_____F

Introduction

The following program is a driver program, written in BASIC/47, to enable the terminal to control an HP 9874A Digitizer.

HP 9874A Digitizer Driver Using Rubber Band Line

```
1000 REM ******************************************************************
1010 REM *                                                                *
1020 REM * 9874 DIGITIZER DRIVER USING RUBBER BAND LINE                   *
1030 REM *                                                                *
1040 REM * SETUP: 9874 MUST USE HP-IB ADDRESS 15                          *
1050 REM *                                                                *
1060 REM * OPERATION: MOVING DIGITIZER CURSOR WILL MOVE TERMINAL CURSOR   *
1070 REM *            PRESSING DIGITIZE SWITCH WILL                       *
1080 REM *              MOVE PEN TO CURSOR POSITION                        *
1090 REM *            PRESSING DIGITIZE SWITCH IN DRAW MODE WILL          *
1100 REM *              DRAW A LINE FROM PEN POSITION TO CURSOR           *
1110 REM *              POSITION AND THEN CHANGE PEN TO CURSOR            *
1120 REM *              POSITION                                          *
1130 REM *            SPECIAL FUNCTION KEYS:                              *
1140 REM *              fa = MOVE MODE                                    *
1150 REM *              fb = DRAW MODE                                    *
1160 REM *              fc = STOP PROGRAM                                 *
1170 REM *              fd,fe = NOT USED                                  *
1180 REM *            WORKS IN BOTH SINGLE AND CONTINUOUS MODES           *
1190 REM *                                                                *
1200 REM ******************************************************************
1210 REM
1220 INTEGER X,Y,Z,A,St,Pn
1230 ASSIGN "H#15" TO #1
1240 REM * INITIALIZE 9874; SET SINGLE MODE, SET AIR TOGGLE MODE,
1250 REM *   BEEP AND TURN ON fa LED
1260 PRINT #1;"IN;SG;AT;BP;SK; 1"
1270 REM * SET PEN CONTROL FOR MOVE MODE
1280 Pn=-2
1290 REM * INITIALIZE PLOTTING DEVICE (2647)
1300 PLOTR
1310 REM * SET ASPECT RATIO AND SCALE TO ACCOMMODATE 9874
1320 SETAR (1.29)
1330 SCALE (0,17400,0,13500)
1340 REM * CLEAR GRAPHICS MEMORY, TURN ON GRAPHICS CURSOR AND
1350 REM *   RUBBER BAND LINE
1360 PRINT CHR$(27)&"*dfakM"
1370 REM * REQUEST DIGITIZER STATUS AND READ IT
1380 PRINT #1;"OS"
1390 READ #1;St
1400 REM * HAS A KEY BEEN PRESSED?
1410 IF St AND 128 THEN 1540
1420 REM * IS A DATA POINT READY?
1430 IF St AND 4 THEN 1650
1440 REM * REQUEST 9874 CURSOR POSITION AND READ IT
1450 PRINT #1;"OC"
1460 READ #1;X,Y,Z,A
1470 REM * PRINT X AND Y VALUES ON 9874 DISPLAY
1480 PRINT #1 USING 1710;X,Y
```

```
1490 REM * MOVE 2647 CURSOR TO CORRESPOND TO 9874 CURSOR
1500 POINT (X,Y)
1510 REM * LOOK AT STATUS AGAIN
1520 GOTO 1380
1530 REM * REQUEST KEY CODE AND READ IT
1540 PRINT #1;"OK"
1550 READ #1;Ky
1560 REM * IF fa KEY CHANGE TO MOVE MODE
1570 IF Ky AND 1 THEN Pn=-2
1580 REM * IF fb KEY CHANGE TO DRAW MODE
1590 IF Ky AND 2 THEN Pn=-1
1600 REM * IF fc KEY STOP PROGRAM
1610 IF Ky AND 4 THEN STOP
1620 REM * GO UPDATE CURSOR
1630 GOTO 1450
1640 REM * REQUEST DATA POINT AND READ IT
1650 PRINT #1;"OD"
1660 READ #1;X,Y,Z,A
1670 REM * MOVE PEN OR DRAW LINE
1680 PLOT (X,Y,Pn)
1690 REM * GO UPDATE CURSOR
1700 GOTO 1450
1710 IMAGE "LB",XX,DDDDD,X,DDDDD,XX
1720 END
```

BASIC/47 includes as a standard feature an extension of AGL which allows you to send graphics output to the large screen, high resolution HP 1310B graphics display (instead of the 2647F display) using the HP 1351A graphics translator. All BASIC and AGL commands except the cursor control functions (POINT, CURSOR and DIGITIZE), LINE, and LDIR for PI/2 and 3PI/4, are supported with this configuration. The 19″ screen of the HP 1310B provides a high resolution, high quality display for graphics with 1000 × 1000 adressable resolution. The 2647F's shared peripheral interface, another standard feature, allows you to produce local hardcopy through a variety of HP peripherals, including the multicolor 8-pen and large format plotters.

To use this feature, simply type AUTOX following the asterisk in the command channel. Note that the AUTOPLOT/47 table which appears contains a line identifying this version as the 1351 DRIVER OPTION.