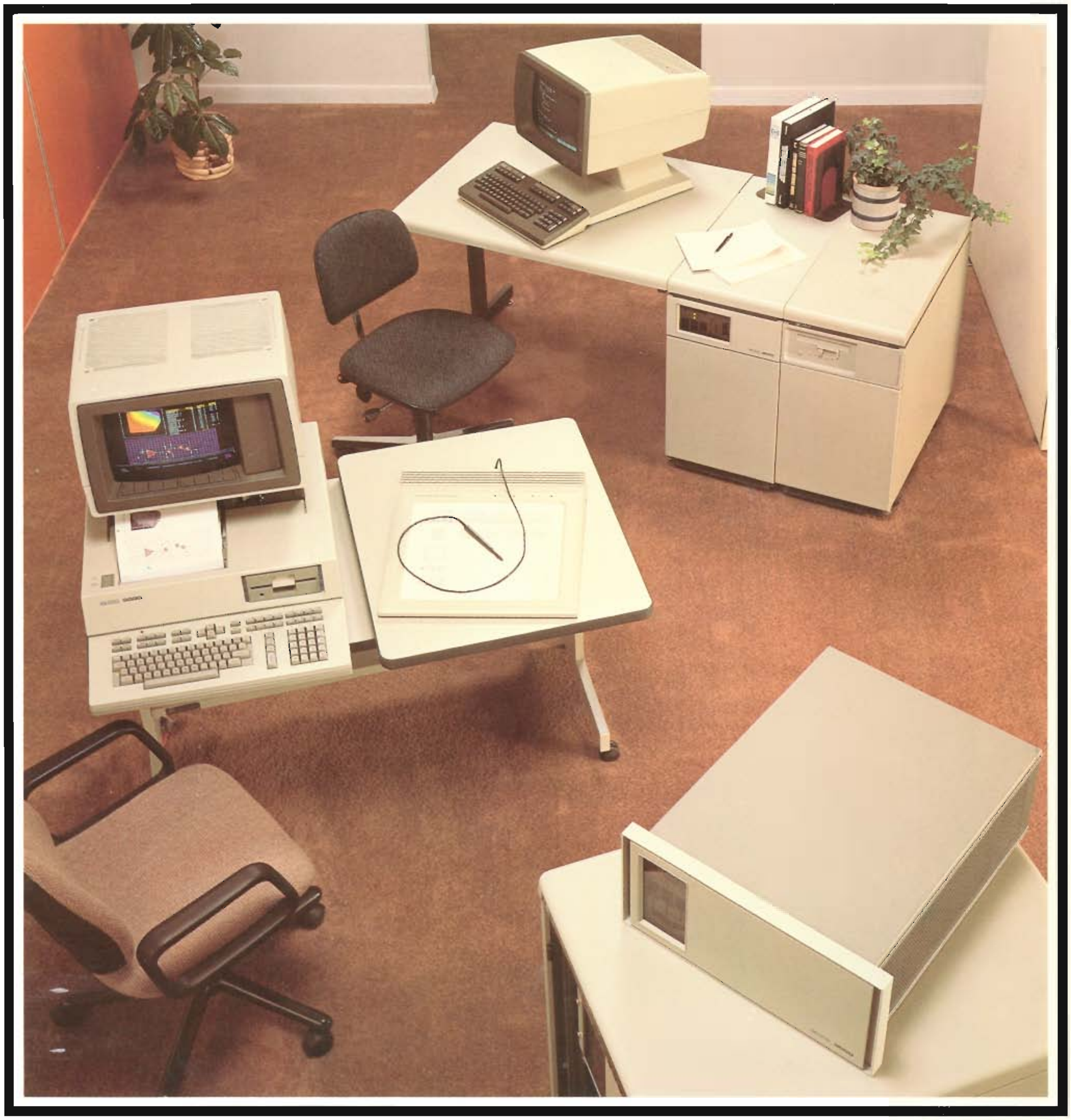
HP 9000 Series 500 Computers
Model 520

HEWLETT PACKARD

BASIC Programming Techniques



BASIC Programming Techniques

for the HP 9000 Series 500
Model 520 Computer

Manual Part No. 97050-90000

Hewlett-Packard Company
3404 East Harmony Road, Fort Collins, Colorado 80525

Printing History

New editions of this manual will incorporate all material updated since the previous edition. Update packages may be issued between editions and contain replacement and additional pages to be merged into the manual by the user. Each updated page will be indicated by a revision date at the bottom of the page. A vertical bar in the margin indicates the changes on each page. Note that pages which are rearranged due to changes on a previous page are not considered revised.

The manual printing date and part number indicate its current edition. The printing date changes when a new edition is printed. (Minor corrections and updates which are incorporated at reprint do not cause the date to change.) The manual part number changes when extensive technical changes are incorporated.

First Edition...December 1982
Update 1...February 1983
Update 2...May 1983
Reprint...June 1983
Second Edition...November 1983
Third Edition...March 1984
Update 3...August 1984: includes addition of Chapter 23, The HP-IB Interface.
Third Edition...January 1985

Note

The HP 9000 Series 500 Model 20 is now called the HP 9000 Model 520. There is no change to the product. Only the numbering convention has changed.

Preface

This manual presents HP 9000 Model 520 BASIC Programming Techniques. It is intended to familiarize you with your computer as well as the HP 97050 version of the BASIC language. Syntax details of the BASIC keywords are found in the BASIC Language Reference manual and occur in this manual only incidentally.

Documentation Map

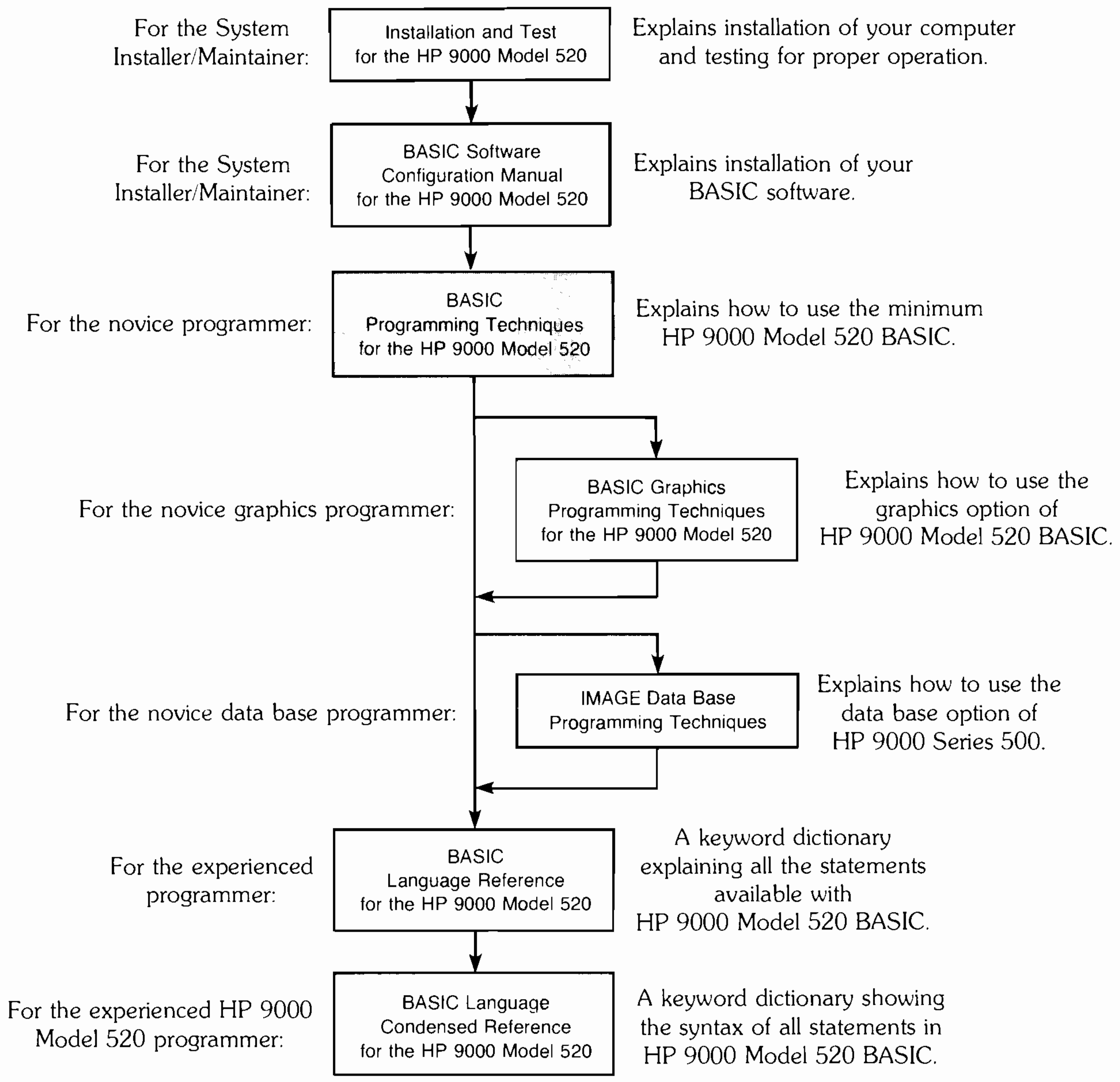

For the System Installer/Maintainer:
Installation and Test for the HP 9000 Model 520
Explains installation of your computer and testing for proper operation.
For the System Installer/Maintainer:
BASIC Software Configuration Manual for the HP 9000 Model 520
Explains installation of your BASIC software.
For the novice programmer:
BASIC Programming Techniques for the HP 9000 Model 520
Explains how to use the minimum HP 9000 Model 520 BASIC.
For the novice graphics programmer:
BASIC Graphics Programming Techniques for the HP 9000 Model 520
Explains how to use the graphics option of HP 9000 Model 520 BASIC.
For the novice data base programmer:
IMAGE Data Base Programming Techniques
Explains how to use the data base option of HP 9000 Series 500.
For the experienced programmer:
BASIC Language Reference for the HP 9000 Model 520
A keyword dictionary explaining all the statements available with HP 9000 Model 520 BASIC.
For the experienced HP 9000 Model 520 programmer:
BASIC Language Condensed Reference for the HP 9000 Model 520
A keyword dictionary showing the syntax of all statements in HP 9000 Model 520 BASIC.

Table of Contents

Chapter 1
Getting Acquainted with BASIC

Welcome to the first chapter of BASIC Programming Techniques. In this chapter, you'll learn about the BASIC Language System with simple, interactive examples that create and run programs, display the results on the CRT, save a program on the internal flexible disc drive, and finally load the program back into memory from the disc drive.

The **BASIC Language System** is a single-user operating system dedicated to one language, BASIC. The term "Language System" is derived from a combination of the terms "programming language" and "operating system".

First Time Power-up

First you'll learn how to start your system with the media supplied by Hewlett-Packard. However, you can store your BASIC Language System on a variety of mass storage media. It may be on an internal winchester disc drive, on an external hard disc drive, on an external flexible disc or on discs for the internal flexible disc drive. If someone else has set up your computer, they can best help you to learn the proper power-up procedure for your specific system. Quite often, it is as simple as turning off your machine and then turning it back on.

The BASIC Language System is supplied on two flexible discs. To load or "boot-up" this system, perform the following steps.

1. Power-down the computer.
2. Remove the two BASIC System Boot discs from the plastic disc library case and from their envelopes.
3. Insert "Disc 1 of 2-disc BASIC System Boot" into the internal flexible disc drive and set the power switch to ON. Three system messages are displayed.

   ```
   Loader REVxx.x
   ```
   (The loader number xx.x can vary.)
   ```
   BASIC LANGUAGE SYSTEM  REV xx.xx              Vol 1 of 2
   ```
   (REV xx.xx refers to the latest revision number of the system.)
   ```
   Please mount next volume.
   ```
   (This message takes a few seconds to appear.)

4. Now remove Disc 1 and insert the disc labeled "Disc 2 of 2-disc BASIC System Boot". Three messages appear on the display.

   ```
   BASIC LANGUAGE SYSTEM  REV XX.XX              Vol 2 of 2
   ```
   (Again REV xx.xx refers to the latest revision number.)
   ```
   (c) Copyright Hewlett-Packard Company, 1982,1983. All rights reserved.
   BASIC ready for use.
   ```
   (This message also takes a few seconds to be displayed.)

When a flashing underscore character '_' (the cursor) appears on the lower left of your CRT followed by "`BASIC ready for use.`", the "boot-up" process is finished.

Remove the second disc and put both discs away.

When you are booting your system, there are three conditions of which you need to be aware.

- After inserting a flexible disc, the door must then be closed, even if you want to remove the disc immediately.
- When changing flexible discs, wait 4-5 seconds before inserting your new disc.
- Approximately fifteen seconds of "wake-up" time is necessary before your internal hard disc is ready (if you have one). During this time the system is performing its self-tests and no messages are displayed on the CRT.

During the loading sequence all hardware is tested and then the computer searches for a system. If any hardware fails the self test a message is displayed near the top of the CRT display indicating the nature of the failure. For a description of the self-test errors and what to do about them, refer to the "Installation and Test" manual.

Sample Programs

You can now enter and run programs. Here is a short example to try:

1. Type:

   EDIT (EXECUTE).

   A "10" appears on the left of your CRT display to indicate the system is in the edit mode and therefore ready to receive program lines.

2. Enter the following program, remembering these details.

   - Use uppercase characters only. If the light to the left of (CAPS LOCK) is not on, press (CAPS LOCK). The keyboard is now in uppercase mode.
   - Press (RETURN) at the end of each line to enter that line of code as part of your program. If there is a typing mistake, your computer beeps and shows you where it thinks the error is.
   - If you make a mistake, move the cursor around with the arrow keys and input that part of the line or the whole line again.

   Here is the program:

   ```
   10  INPUT "WHAT IS YOUR NAME?",N$
   20  PRINT "HELLO, ";N$
   30  END
   ```

3. Press (RUN) to start your program. A flashing underscore character is on the lower left of your CRT and a (?) is on the lower right. This indicates that the system is waiting for you to input your name. Type in your name and press (RETURN). Use a short name, less than eighteen characters. The message "`HELLO,` *your name*" appears on your CRT display.

4. You can go back into the EDITOR to look at your program or to correct any mistakes by typing:

   ```
   EDIT (EXECUTE).
   ```

5. Next, prepare a flexible disc to store your program. This is done with the INITIALIZE statement. Get a blank 5-1/4" flexible disc for your internal flexible disc drive. Make sure that the disc does not contain any important data or programs. When a disc is initialized, all the data on it is lost.

   Check that the disc is not "write protected". Notice that the disc has a small, square notch on one side. When this notch is open, the computer is allowed to write data to the disc. If this notch is covered, say by a piece of tape, data may be read from the disc, but recording and initializing is not allowed.

   Insert the disc and type:

   ```
   INITIALIZE ":INTERNAL" (EXECUTE)
   ```

   `":INTERNAL"` is the name of the internal flexible disc drive.

   Wait for the asterisk in the corner of your CRT display to go out, indicating that the initialization process is complete. This takes a couple minutes. Your disc is now ready to store your program.

6. To store your program on the disc, type:

   ```
   STORE "HELLO:INTERNAL" (EXECUTE)
   ```

   This stores the program under the file name "HELLO" on the disc in drive ":INTERNAL".

7. The CAT (CATalog) command lets you see what is on your disc. Type:

   ```
   CAT ":INTERNAL" (EXECUTE)
   ```

   The "listing" that appears on your display shows the files on the disc in the drive ":INTERNAL".

   Save this disc as a practice disc on which to place other programs and data needed in other chapters.

8. To erase the current program from memory type:

   ```
   SCRATCH (EXECUTE)
   ```

9. To enter the next program go back into the edit mode with the commands previously described. When you are finished you can run, store and erase this program, too.

   ```
   10  FOR I=200 TO 10000 STEP 10
   20    PRINT I
   30    BEEP I,.01
   40  NEXT I
   50  END
   ```

   Try storing this program on your disc under the name "MUSIC". The proper command is:

   ```
   STORE "MUSIC:INTERNAL"
   ```

10. Now, for practice, try loading the first program back into memory by typing:

    ```
    LOAD "HELLO:INTERNAL" EXECUTE
    ```

    Now:

    ```
    EDIT EXECUTE
    ```

    to confirm that the program is in memory.

What to Do Next

The previous section is just a small sample of the things you can do with your computer. The rest of this manual explains the available commands and features in greater detail. You don't have to read every topic in every chapter and you don't have to read each tabbed section in the order they are presented. In general, though, this manual flows from easiest information in the front to hardest information in the back.

The first tabbed section, **Getting Started**, includes this chapter and another chapter which introduces some of the hardware of your computer.

The second tabbed section, **Program Structure**, explains how to use the program editor and describes a variety of programming statements available.

The third section, **Data Storage**, explains how your computer's memory stores the various data used for programs. It gives hints on how to make your programs and data fit in less memory and how to get your programs to execute faster.

I/O Topics explains simple input and output operations with peripheral devices.

Advanced I/O Topics explains more sophisticated I/O subjects.

Mass Storage Topics explains how to use mass storage devices such as disc drives and tape drives.

The appendices cover the various utility programs shipped with your software, and the differences between your computer and the HP 9835/45 or HP 9000 Series 200 computers.

We strongly suggest you sit near your computer when reading this book and that you try the examples as you go along. If you have any comments or suggestions for improving this manual, please fill out the Customer Comment card in the back. We will respond to you as quickly as possible.

HAPPY COMPUTING!

Chapter 2
Getting to Know Your System

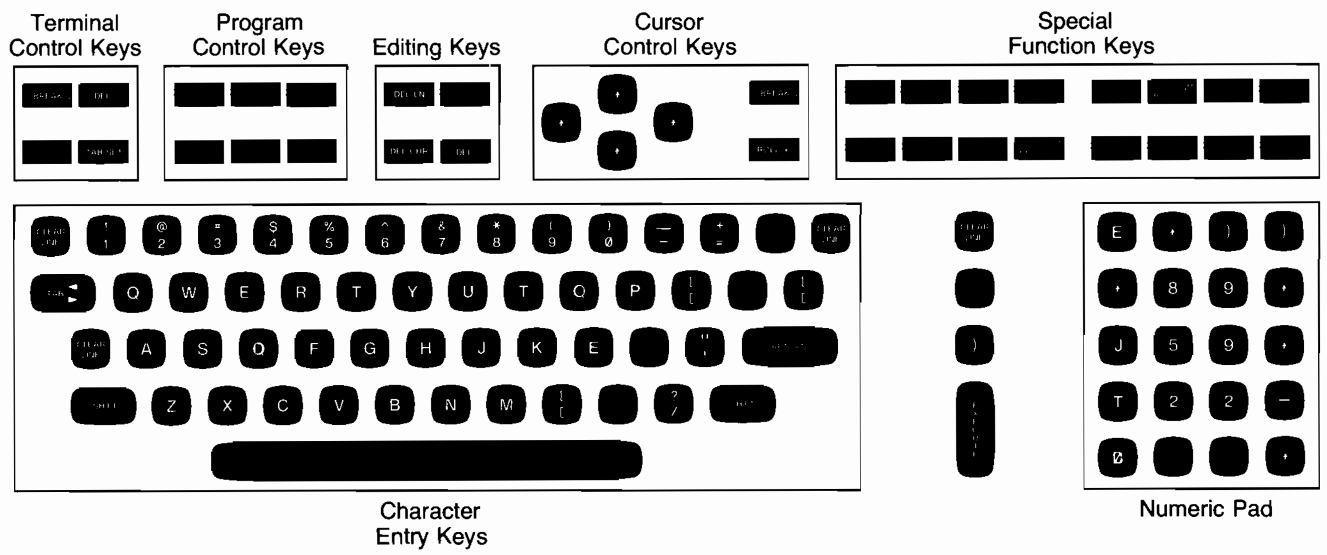
This chapter explains details regarding your computer's hardware and how it is affected by the BASIC Language System. This chapter is not intended to be a complete description of your computer's hardware. (See the Installation and Test manual and the Service manual for more information.)

Although some topics are covered fairly deeply, it is not necessary to understand everything in this chapter to appreciate the following chapters.

Topics covered include:

- the keyboard;
- the CRT display;
- the thermal printer;
- the real-time clock;
- the tone generator;
- non-volatile memory.

The Keyboard



The keyboard is arranged into logical groups, a Character Entry Keyboard, a Numeric Keypad, Cursor Control Keys, Special Function Keys, Editing Keys, Program Control Keys and Terminal Control Keys. In addition to these, there are a few miscellaneous keys, the (EXECUTE), (RECALL), (CLEAR LINE) and (CLEAR SCN) keys.

Character Entry Keyboard

These keys are basically the same as you find on any typewriter. When you press these keys, the system echoes them on a certain portion of the CRT display. The only two unusual keys are the CONTROL Key, (CTRL), and the RETURN key, (RETURN).

The (CTRL) Key

The (CTRL) key, when pressed at the same time as any other Character Entry Key, outputs a **control character**. A control character is a character which, when sent to a device, can be interpreted as an action such as a formfeed or carriage-return. This is different than **displayable characters** which are used as input data by the device. Control characters can also be output as displayable characters. Their interpretation is dependent upon whether DISPLAY FUNCTIONS is enabled or disabled. If DISPLAY FUNCTIONS is enabled, the character is output as a displayable character. If DISPLAY FUNCTIONS is disabled, the action specified by the control character is performed.

To disable DISPLAY FUNCTIONS, type:

```
    PRINT "EcZ"        (EXECUTE)
```

(To print the 'Ec' (escape) character, press (CTRL) and ([ ) simultaneously.)

To enable DISPLAY FUNCTIONS, type:

```
    PRINT "EcY"        (EXECUTE)
```

The ASCII standard control characters always are displayed when you input them on the command line of the CRT. They may or may not appear when output to a device. The extension control characters perform their specified action (inverse video and so on) on the command line, but they may or may not appear when output to a device.

For a complete list of control characters, see the "Useful Tables" section of the BASIC Language Reference.

The RETURN Key

The RETURN key is similar in function to that of a typewriter carriage-return mechanism in that it signals the end of a line. However, it also sends a signal to your computer which says: "Here is a piece of data." Your system then examines the line you typed (entered) and one of three things happens. First, it might recognize what you've typed in as some data for which it has been waiting. Second, it might recognize it as a statement for a program. In this case, it stores the program line so you can execute it later. Third, it might not understand the line at all in which case it gives you an error message.

The computer doesn't care what you type until you press the RETURN key. This is the signal to try to understand what has been typed in.

The EXECUTE Key

The EXECUTE key is similar to the RETURN key. It also signals the end of a line of input. However in this case, the computer examines the line you have typed in and tries to execute that command right away. If it can't execute the command, it gives you an error message.

It's important to remember the difference between RETURN and EXECUTE. RETURN signals that the item entered is a program line to be saved or a piece of data to be used by a waiting program. EXECUTE signals that the item entered is a command to be executed immediately or an equation to be solved.

The RECALL Key

Pressing the RECALL key redisplays the last thing you input. Pressing it again displays the thing you input before that and so on. There is a buffer inside the system which stores the last 500 characters that were input. This means the system can remember a few long input items or a lot of short items.

If you want to move forward through the list of input items, press SHIFT RECALL. For example, type:

```
PRINT "FIRST THING ENTERED." EXECUTE
PRINT "SECOND THING ENTERED." EXECUTE
PRINT "THIRD THING ENTERED." EXECUTE
PRINT "FOURTH THING ENTERED." EXECUTE
RECALL
RECALL
RECALL
RECALL
SHIFT RECALL
SHIFT RECALL
SHIFT RECALL EXECUTE
RECALL
```

The CLEAR LINE Key

The CLEAR LINE key deletes the input item you're currently entering. It is useful if you make a mistake while typing and want to start over. It also is useful when in Edit mode (explained in the "Editing" chapter).

SHIFT CLEAR LINE clears from the cursor to the end of the line.

The CLEAR SCN Key

The CLEAR SCN (CLEAR SCREEN) key clears the display of the internal CRT. It only affects alphanumeric characters; it does not clear graphics. To clear graphics, see the "BASIC Graphics Programming Techniques" manual.

CLEAR SCN clears private screens and SHIFT CLEAR SCN clears public screens.

The Numeric Keypad

This set of keys was designed to make number entry an easier task. It is the same basic format as that used by most calculators. The E key is used for scientific notation. The following are examples of typing in scientific notation.

If you type in:	You are executing:
1.45E3 + 12E4	1450 + 120000
7.89E2/56	789 ÷ 56

The ^ key is the exponential key. The following are examples of using this key.

If you type in:	You are executing:
2^3 * 4.5	$2^3 \times 4.5$
3^4/2.3E2	$3^4 \div 230$

Note that these keys send the same signals to the computer as their equivalent keys in the Character Entry Keys portion of the keyboard. In fact, you can use any combination of numeric keys from both portions of the keyboard to solve a math problem.

For more information on the way the system evaluates numeric expressions, see the "Numeric Computations" chapter of this manual.

Cursor Control Keys

The cursor control keys consist of two groups, the arrow keys and the roll keys. (↑), (↓), (→), (←) move the cursor around. Notice that there is a limited area in which the cursor can move, the whole width of the display but only two lines up and down. This is limited area is known as a screen. This particular screen is known as the KEYBOARD SCREEN. The area above the KEYBOARD SCREEN where some of the examples have been printed is known as the STANDARD SCREEN. Screens are discussed more thoroughly in "The CRT Display" section of this chapter. For now, keep in mind that there is an area where all of your commands are typed and that the cursor only moves within this area when you are inputting commands.

Pressing (SHIFT) (←) moves the cursor back to the first character of the KEYBOARD SCREEN.
Pressing (SHIFT) (↓) moves the cursor to the last characters of the KEYBOARD SCREEN.

The roll keys, (ROLL ↑) and (ROLL ↓), move the text in the STANDARD SCREEN. As an example, execute the following command and then roll the STANDARD SCREEN up and down.

```
    PRINT "A SIMPLE LINE OF TEXT."
```

Notice that the text can roll up all the way off the display. This is because the system maintains some extra memory for the screen's text which is not displayed. With this you can have more text on hand without having to have a huge display; you simply roll the text to the part you want to use.

Special Function Keys

The **Special Function Keys**, or SFKs, are the sixteen keys in the upper-right part of the keyboard with the keycaps (0 15) to (16 31).

These keys have two separate purposes.

- They can be typing aids. Each key can store a command, program line or other piece of data which can then be entered with a single press of the key. This is very useful when you expect to enter the same set of items over and over. Instead of typing in the same thing many times, you can define it once and then use it with one key press.
- You can also use the keys to cause a program branch. For example, if you have a program with a menu of choices, SFKs can be defined so that pressing one causes a branch to another part of a program.

There are 32 SFKs available, labelled 0 thru 31. The first sixteen are accessed by simply pressing the key with the corresponding number in its lower-left corner. The other sixteen are accessed by pressing (SHIFT) while pressing the key with the corresponding number in its upper-right corner.

When you power up your computer, SFKs eight thru eleven have definitions already assigned to them. Their definitions are shown directly below these keys and are:

EDIT FIND CHANGE SCRATCH

Notice that there are also three SFKs labelled:

AIDS MODES USER KEYS

These are for the HP-UX system and are not defined in the BASIC system. You can ignore the labels.

You can use these SFKs with their predefined functions or redefine them for your own use. SFKs can have up to 80 characters each. Try this sequence:

```
EDIT KEY 0 (EXECUTE)
```

"`Key 0`" appears in the upper-left corner of the display. This means that anything you type in becomes part of the definition of SFK 0. Now type:

```
PRINTER IS CRT (RETURN)
```

You have just defined SFK 0 to specify that the output of any PRINT statements are to be sent to your CRT display. Try the following.

```
EDIT KEY 1 (EXECUTE)
PRINTER IS PRT (RETURN)
```

You have now defined SFK 1 to specify that the output of any PRINT statements go the internal printer. Execute:

```
EDIT KEY 2 (EXECUTE)
PRINT "DO NOT TOUCH.  WILL BE BACK IN FIVE MINUTES." (RETURN)
```

If you now press SFK 0, your definition appears on the CRT and is executed when you press (EXECUTE). Now press SFK 2 followed by (EXECUTE). The CRT displays your message: "`DO NOT TOUCH.  WILL BE BACK IN FIVE MINUTES.`".

If you press SFK 1 (EXECUTE) and the SFK 2 (EXECUTE), the message is on the printer, if you have one.

You can add an automatic (EXECUTE) command to your definition of the SFK by including a (CTRL) (EXECUTE) sequence. It is also a good idea to start an SFK definition with a (CTRL) (CLR LN). This deletes any characters on the input line before printing the SFK's predefined command. For example, type:

```
EDIT KEY 2 (EXECUTE)
(CTRL) (CLR LN) (Press these at the same time.)
PRINT "DO NOT TOUCH.
(CTRL) (EXECUTE) (Press these at the same time.)
(RETURN)
```

Now press SFK 2.

When acting as input devices for program branches the first eight SFKs - zero thru seven - can have corresponding labels placed on the internal display. This enables the user to remember what each key is defined to do. As an example, type in the following program.

```
10  ON KEY 2 LABEL "  KEY 2           " GOTO 40
20  ON KEY 3 LABEL "  KEY 3           " GOTO 40
30  WAIT
40  BEEP
50  GOTO 30
60  END
```

Each label has space for twenty characters, ten characters on two lines. All the labels butt up against each other so, to keep the labels separated you usually don't want to use the tenth and twentieth spaces to keep the labels separated.

If you have an HP 98770B or HP 98780B internal CRT, you'll notice that there are eight keys on the CRT bezel which lie directly beneath the labels for the SFKs. These eight keys are duplicates of the eight keys on the keyboard. Thus, instead of looking at the CRT while pressing keys on the keyboard, you can match the key to its label.

If you look carefully, you'll see that some of the keyboard SFKs have a white label on the front side of the key cap. These are modes of printing you can assign to the current output device. For example, you can have the internal CRT print its characters in blinking inverse video or, if you have a color CRT, you can print characters in blue with a blue underline. This is enabled by pressing the (CTRL) key and the appropriate SFK simultaneously. To disable this mode of printing, press (CTRL) and the SFK again.

Program Control Keys

The **Program Control Keys** help you when running or debugging a program. The (RUN) key starts a program running and (STOP) stops it. The (PAUSE) key stops a program where it is at without destroying any data and so on. The (CONTINUE) key continues a paused program where it left off. The (STEP) key steps through program lines one at a time, showing what line is to be executed next. This helps you follow the flow of a program that would otherwise happen too fast to see. The (PRT ALL) or PRINT ALL key prints each program line as it is executed and all output on the current output device. This helps you record the flow of a program for later analysis. These keys are covered more completely in the "Editing" chapter of this manual.

When working with multiple partitions, you can start, stop, pause and continue them simultaneously using (SHIFT) (RUN), (SHIFT) (STOP), (SHIFT) (PAUSE) and (SHIFT) (CONTINUE) respectively.

Editing Keys

The **Editing Keys** help you edit programs when you are in edit mode. (INS LN) inserts a line in a program, (DEL LN) deletes a line. (INS CHR) inserts a character in a program line and (DEL CHR) deletes a character. These keys are also covered in the "Editing" chapter of this manual.

Terminal Control Keys

The **Terminal Control Keys**, BREAK, DEL (DELete) and ESC (ESCape) are used primarily for communication with other computers and with HP-UX systems.

The BREAK Key

BREAK is disabled when the BASIC Language System is running on your computer.

The DEL Key

DEL generates the ASCII "DEL" or delete character, just as it does on a terminal. This is also sometimes known as the "rubout" character.

The ESC Key

The ESC key generates the ASCII Escape character. This can be used in conjunction with other keys from the Character Entry section of the keyboard to output certain character codes in a program. For example in the discussion of the CTRL key, you are asked to type:

```
PRINT "ECZ"     EXECUTE
```

and

```
PRINT "ECY"     EXECUTE
```

Instead of generating the escape character using CTRL [, you can simply press ESC.

If you look at the escape sequences and control characters available with your machine, you find some overlap in their capabilities. The control characters provide compatibility with computers such as the HP 9845, HP 9826 and HP 9836. The escape codes are compatible with a number of printers.

Foreign Keyboards

If you have a Katakana keyboard (used for Japanese output), you can enter Katakana mode by pressing CTRL <. To return to Roman characters, press CTRL >.

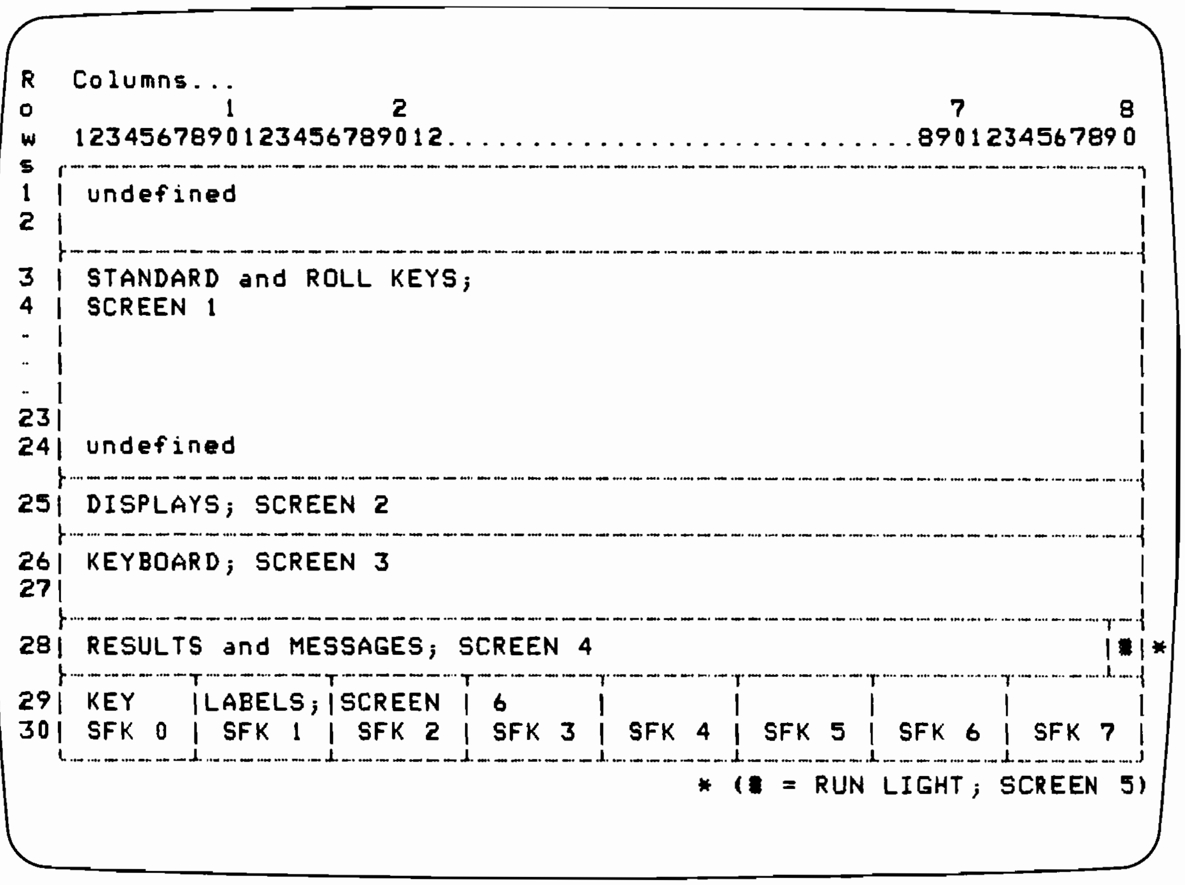
The CRT Display

This section describes the alpha numeric capabilities of your computer's CRT. To learn more about its graphics capabilities, see the BASIC Graphics Programming Techniques manual.

Display Organization

The CRT on your computer can display 26 or 30 80-character lines of text, depending upon the type of display you have. This area is normally divided into several smaller display areas called screens. One screen, for example, is reserved for entering and executing keyboard commands. You can print to a screen, assign buffer space, assign special functions and read from a screen. You can reorganize the display as needed by using BASIC statements.

This diagram shows the default screen assignments.



```
R  Columns...
o          1         2                                                    7         8
w  12345678901234567890 12............................8901234567890
s ┌───────────────────────────────────────────────────────────────────────┐
1 | undefined                                                             |
2 |                                                                       |
  ├───────────────────────────────────────────────────────────────────────┤
3 | STANDARD and ROLL KEYS;                                               |
4 | SCREEN 1                                                              |
- |                                                                       |
- |                                                                       |
- |                                                                       |
23|                                                                       |
24| undefined                                                             |
  ├───────────────────────────────────────────────────────────────────────┤
25| DISPLAYS; SCREEN 2                                                    |
  ├───────────────────────────────────────────────────────────────────────┤
26| KEYBOARD; SCREEN 3                                                    |
27|                                                                       |
  ├─────────────────────────────────────────────────────────────────────┬─┤
28| RESULTS and MESSAGES; SCREEN 4                                      |▮|*
  ├────────┬────────┬────────┬────────┬────────┬────────┬────────┬──────┴─┤
29| KEY    |LABELS; |SCREEN  | 6      |        |        |        |        |
30| SFK 0  | SFK 1  | SFK 2  | SFK 3  | SFK 4  | SFK 5  | SFK 6  | SFK 7  |
  └────────┴────────┴────────┴────────┴────────┴────────┴────────┴────────┘
                                             * (▮ = RUN LIGHT; SCREEN 5)
```

At power-up, there are six screens. These screens have system functions assigned to them. As you use the system, you'll probably notice that SFK labels appear at the bottom of the display, the run light is on line 28 and PRINT statements start on line 3. These functions and a few others are assigned to particular screens at power-up and when the RESET SCREENS statement is executed. The six screens created and their functions are shown in the following table.

Screen Number	Starting Row,Column	Number of lines	Number of Chars/Line	Buffer Size (Lines)	Default Functions
1	3,1	20	80	25	STANDARD SCREEN (STD)
2	26,1	1	80	0	DISPLAYS
3	28,1	2	80	0	KEYBOARD
4	24,1	1	78	0	RESULTS and MESSAGES
5	28,80	1	1	0	RUN LIGHT
6	29,1	2	80	0	KEY LABELS

The **STANDARD** or STD SCREEN holds 45 lines of information, although only 20 lines appear on the display. Program output appears in this area. When the 20 line area is filled, the top line scrolls off into a buffer, or holding area, of memory as each new line is output to the display. Use the cursor control keys to scroll thru the page and view these lines. When the entire output area is filled, each new line output to the display causes a line to be lost from the top of the buffer.

DISPLAYS is used to display instructions or prompts from a program to the operator.

KEYBOARD is the portion of the CRT which displays information typed from the computer keyboard. If a program prompts you to enter information, as you type it on the keyboard, it is displayed in the keyboard area of the CRT display. Up to 160 characters, or two full character lines, can be entered in the keyboard area of the display.

RESULTS and MESSAGES displays the results of most keyboard operations. For example, if you type:

```
483 + 239 (EXECUTE)
```

this information appears in the keyboard area of the display as you type until you press the (EXECUTE) key, when the sum is calculated and displayed in the message/results area of the display, and the original problem disappears.

```
722
```

RUN LIGHT identifies the current state of the system. A summary table identifying the meaning of each run light follows.

Run Light	Meaning
▒	running a program, program waiting
–	paused
(blank)	stopped, no program running
?	waiting for keyboard input
*	executing keyboard command

KEY LABELS displays labels which appear when one or more of the lower eight special function keys, 0 thru 7, are given those names. These keys are duplicates of the CRT softkeys on the display. They are placed on the CRT to allow easy association of a 98770B or 98780B CRT softkey with a label. The softkey labels are only displayed while a program is running; they are blanked out when the program pauses or stops.

User-Defined Screens

You can create new screens and delete or redefine existing screens - up to one hundred screens, numbered 0 thru 99 can be defined. In a program you might want to use one area of the CRT to display data on an external process and another area of the CRT to display the results of many calculations. If you use a DISP statement, the two displays overlap each other. If you use PRINT statements, you have to worry about where the cursor is located so you don't overwrite what is on the CRT. Or you can create two screens which are independent of each other and each routine of the program can write to a different one. For example:

```
 50  CREATE SCREEN 10,Row,Column,Height,Width
 60  CREATE SCREEN 20,15,12,5,80
  .
  .
  .
 90  PRINTER IS SCREEN (10)
100  PRINT Message$
110  PRINTER IS SCREEN (20)
120  PRINT Results
```

If the screen already exists, you don't get an error. Instead the screen is redefined to the new size and location specified in the CREATE SCREEN statement. If there is data in the screen, it is cleared.

A screen is independent of the progam that creates it. When the program stops, the screens created in the program still exist.

To delete a screen, use DEL SCREEN.

```
300  DEL SCREEN 10
310  DEL SCREEN 20
```

In order to delete a screen, no system functions can be assigned to that screen. The system functions are shown in the Default Functions column of the Default Screens table.

When working with multiple partitions you can specify that a screen is to be public or private. A public screen can be written to by any partition. A private screen can only be written to by the partition that created it. For more information, see the "Partitions and Events" chapter of this manual.

Stacking Screens
You can create screens of any size, limited only by the size of the physical CRT and available memory. There may be times when screens overlap. When this occurs, the screens are stacked one on top of the other in a manner similar to stacking pieces of paper.

When a screen is accessed, either written to or read from, that screen is brought to the top of the stack. You can see the entire screen. If the screen overlaps other screens, only those portions of the other screens which are not covered by the most recently accessed screen are visible.

Moving Screens
You use the MOVE SCREEN statement to move a screen around the CRT. The size of the screen, the screen's position and the data in the screen remain unchanged.

Reading from and Writing to Screens
For more information on this, see the "Partitions and Events" chapter of this manual.

Display Functions

You can access a number of unique display functions on your CRT using control characters or escape sequences. For example, you can use inverse-video characters by pressing (CTRL) (0 15).

For more information, see the "Character Codes" table and the "Escape Sequence Effects" table in the BASIC Language Reference.

Degaussing a Color CRT

After prolonged use, the color mask inside a color CRT becomes magnetized. This leads to wavering and poor display quality. To correct this, see the Degauss Utility in the "Utilities" appendix of this manual.

Converging a Color Display

A color CRT has three electron guns which sometimes get out of alignment. This leads to a blurry picture and poor color registration. To correct this, see the Convergence Utility in the "Utilities" appendix of this manual.

The Thermal Printer

The thermal printer can also interpret a number of the escape sequences and control codes. For more information on which are supported and what their effects are, see the "Useful Tables" in the BASIC Language Reference.

The Real-Time Clock

The internal clock provides the date, time of day and interval timing. A program can use the clock to schedule itself to execute a routine at a particular time or time and date, after a delay, repeatedly, or to wait quietly for a certain length of time. The system also uses the clock to set the random number generator seed. The clock also sets the time/date stamp on a file when it is altered.

Julian Time

The internal clock keeps track of both the time of day and the day, month and year.

Time of day is the number of seconds past midnight. Midnight time is zero.

The date is the number of seconds past a specific base point date. The base point is midnight on the morning of 24 November −4713. (At this date in history, an unusual astronomical event occurred and astronomers have been using it as a base point ever since.)

The AD/BC convention is not used with the Julian date. Instead, negative years are used. −4712 is the same as 4713 BC because there is no year 0 in the AD/BC convention.

Setting the Clock

The SET TIMEDATE statement sets both the software and hardware clock. SET TIME is a subset of the SET TIMEDATE statement. SET TIME sets the time within the current date. Both statements use seconds rather than days, hours, minutes and seconds to set the clock.

So that you do not have to count the seconds since midnight or the base point of the Julian date, there are two system functions to do this for you: TIME and DATE.

For example, to set the internal clock to 10:30 PM, use the following statement:

```
    SET TIME TIME("22:30")
```

Notice that a 24 hour clock is used, not the AM/PM convention.

Note

When you set the time, the software clock is set to the exact time you specify in the command. The hardware clock is set, but any seconds specified in the command are ignored. If you set the clock on the minute boundary, you avoid an inconsistency.

To set the time and date, use SET TIMEDATE.

```
    SET TIMEDATE TIME("8:00")+DATE("21 NOV 1984")
```

You do not have to set the clock each time you turn on the computer. The clock runs for at least two weeks when the computer is turned off. The actual period varies due to temperature and humidity.

When you power up the computer, the system reads the hardware clock and sets the software clock. If the hardware clock ran out of power when the computer was off, the system sends a message to the CRT:

```
TIME AND DATE NOT SET
```

While the system runs without you resetting the clock, programs which use the clock may get incorrect information.

You should be cautious when setting the clock. If programs are running which use the clock related statements, you could affect their execution. Programs in all partitions are affected.

Reading the Clock

The TIMEDATE function reads the software clock. It returns the number of seconds since the base point. Two other functions convert seconds to standard notation: TIME$ and DATE$.

```
PRINT TIME$(TIMEDATE);DATE$(TIMEDATE)
```

TIME$ returns only the time. The time is specified as HH:MM:SS.

DATE$ returns only the date. The date is specified as DD MMM YYYY. For example, 25 Dec 1983.

You can use the TIMEDATE function to time segments of your program. For example:

```
100  Start_time=TIMEDATE
 .
 .                                     ! Timed Segment
 .
780  Elapsed=TIMEDATE-Start_time
```

Scheduling with the Clock

You can schedule a branch to be taken at a specific time, after a number of seconds or periodically. You can also schedule the program to wait for a number of seconds.

WAIT (seconds) puts the program in the wait state for the specified number of seconds. It is interruptable by keyboard commands but not ON condition interrupts. If the wait is interrupted, the wait time begins over after the keyboard command is executed.

The ON condition statements specify a branch to be taken after a condition has been met. The ON condition statements which use the clock are ON TIME, ON TIMEDATE, ON CYCLE, ON DELAY and ON TIMEOUT. These statements are described in the "ON Conditions" chapter.

The Tone Generator

Your computer has a programmable tone generator, or beeper. The BEEP statement causes an audible tone of the specified frequency (pitch) and duration.

You can also generate a brief tone by sending an ASCII BEL (CHR$(7)) character to the display in an output statement. The tone causes by the BEL character has the frequency and duration of the simple BEEP statement: `BEEP 500,0.1`

Two typical uses of BEEP are:

- alerting the user to errors or unexpected conditions (for example, the BASIC system executes a simple BEEP when an error occurs);
- sending messages to a user whose attention may not be directed to the CRT. You can use different frequencies or combinations of frequencies to signify different computer responses.

The following program waits for you to press SFK 1 or SFK 2 and gives you an audible indication of which SFK you pressed:

```
 10  ON KEY 1 LABEL "PHASE 1" CALL Phase_1
 20  ON KEY 2 LABEL "PHASE 2" CALL Phase_2
 30 Wait:WAIT
 40  GOTO Wait
 50  !
 60  SUB Phase_1
 70    BEEP 2000,.09
 80    BEEP 555,.1
 90    SUBEXIT
100  !
110  SUB Phase_2
120    BEEP 2000,.09
130    BEEP 200,.02
140    BEEP 2000,.09
150    SUBEXIT
160  !
170  END
```

Although a conversion table of frequencies and notes of the chromatic scale is in the "Useful Tables" section of the BASIC Language Reference, the tone generator is not designed to produce music. It generates a single square-wave frequency at a time and has no amplitude (volume) control. The duration of each cycle is an integer multiple of 1/50 000 of a second. Nonetheless, you are invited to try the following example.

```
 10   A=440
 20   Duration=.2
 30   FOR Note=0 TO 12
 40     Sharp_flat=(Note=1 OR Note=3 OR Note=6 OR Note=8 OR Note=10)
 50     IF NOT Sharp_flat THEN
 60       Frequency=A*2^(Note/12)
 70       BEEP Frequency,Duration
 80       WAIT .05
 90     END IF
100   NEXT Note
110   END
```

Non-Volatile Memory

Non-Volatile Memory is a memory area which contains information needed when the system is powered up. The information is not erased when the system is powered down.

The following table shows the location of elements within Non-Volatile Memory (NVM).

Register	Contents	
(1 Byte Each)		
1	Clock Check Byte	
3	NVM Stack Size	
0	Seconds	Time and Date
2	Minutes	Time and Date
4	Hours	Time and Date
7	Day	Time and Date
8	Month	Time and Date
9	Year	Time and Date
5	Century	Time and Date
6	Weekday (unused)	Time and Date
10	CRTL A	Clock Control Registers
11	CRTL B	Clock Control Registers
12	CRTL C	Clock Control Registers
13	CRTL D	Clock Control Registers
14	Timer resolution	
15	I/O Timeout, scan interval	
16	Alpha Color, Keyboard repeat rate	
17-24	Reserved	
25-44	MSUS address	
45	System IO buffer size	
SERVICE ONLY		
46-47	Checksum	
48-53	MC Failures	
54-57	Double-bit errors	
58-60	On time	
61	Number of overheat cycles	
62-63	Number of power up cycles	
64-2047	Reserved	

Reading and Writing to NVM

You can read each NVM register using the STATUS statement. The device selector is 603. The return value is a numeric.

```
STATUS 603,Register;Return
```

You can read all the registers, but only a few can be written to directly (registers 14,15,16 and 45). You can change registers 0, 2 thru 9, and 25 thru 44 indirectly. You cannot change the Service Only registers, 46 thru 2047. These registers contain data on machine performance.

To change the value of the register use the CONTROL statement.

```
CONTROL 603,Register;New_value
```

When a value is changed directly, the register receives the value, but it does not affect software until a SCRATCH A or power up. When the value is changed indirectly, through another statement or utility, the software is also affected.

Clock Check Byte

This register indicate whether or not the clock has a valid time. A valid time is seconds between 0 and 59, minutes between 0 and 59, etc. If the time is not valid, the following message is displayed at power up.

```
TIME AND DATE NOT SET
```

The register can be set to valid only by setting the clock with the SET TIMEDATE statement.

Stack Size

This register indicates the number of finstrates found by the system at powerup. The value is two times the number of finstrates.

If finstrate configuration is found to be a larger value than found in this byte, the value is updated to the current number of finstrates. If the value is smaller an error message is displayed at powerup.

```
FEWER FINSTRATES THAN EXPECTED
```

The message indicates that either finstrates have been removed or that one or more finstrates have failed to report after self-test.

You cannot change the value directly. The utility program SET_NON_VOL_MEM reads the current number of finstrates and updates this register. The SET_NON_VOL_MEM utility is described in the "Utilities" appendix of this manual.

Time

The seconds thru century bytes contain the time and date. At power up, the system clock is set using these values.

You set these bytes and the system clock using the SET TIMEDATE statement.

Control Registers

The control registers, 10 thru 13, are used by the clock. You cannot set them.

Timer Resolution and I/O Timeout/Scan Interval

There are two timers: Timer Resolution and IO Timeout.

The **Timer Resolution** timer specifies how often to update the clock and how frequently to allow timed interrupts. The default value of this timer is 10 milliseconds.

The **IO Timeout** timer specifies how frequently to check for I/O timeouts, to give equal priority partitions CPU time and to perform internal tests. The default value of this time is 10 milliseconds.

The default values of these times are set to provide you with a balance of quick response time and low CPU overhead. At each tick of the timers, the system uses the CPU. The greater the resolution of the timers (smaller time between updates), the more CPU time is used. The default values result in less than one percent system overhead. A timer resolution of one millisecond results in about ten percent overhead.

The values have a range from 500 microseconds to 1/2 second. The Time Resolution must be smaller than the IO Timeout.

The values are contained in one byte expressed in binary as:

NNNNNEEE

where NNNNN is the mantissa with a trailing binary point and EEE is the decimal exponent field.

The resulting number is NNNNN * 10^EEE, microseconds.

For example,

1 millisecond = 1000 microseconds
    $1000 = 1*10^3$
    NNNNN = 00001 binary
    EEE = 011 binary
00001011 binary = 11 decimal

To set the value of the Timer Resolution to 1 millisecond in NVM, use the following command.

```
CONTROL 603,14;11
```

Note

The IOP has its own interval timer which has a non-changeable resolution of 28.888 microseconds. (This is based on the 18 megahertz quartz crystal.) The effective value of the Timer Resolution is always rounded down to a multiple of 28.888.

Color and Keyboard Repeat Rate

The upper three bits of this register contain the alpha color. The lower five bits contain the keyboard repeat rate.

Since this register contain two elements, you must calculate the value as follows:

```
Value = Repeat_rate + (32 * Color)
```

Color

The color portion of this register contains the default color of characters sent to the **alpha** display. It does not affect the graphics display. Literals within an output statement (for example, PRINT or DISP) appear in the color in which they were entered; not necessarily the alpha color.

The default color depends upon the type of display.

HP 98760A	GREEN	(low cost color CRT)
HP 98770B	WHITE	(high performance color CRT)
HP 98780B	WHITE	(high performance monochromatic CRT)

You specify a color as is an integer which corresponds to one of the following colors.

0	BLACK
1	RED
2	GREEN
3	YELLOW
4	BLUE
5	MAGENTA
6	CYAN
7	WHITE

You can also change the alpha color for an individual partition. When you do this, NVM is not changed. The CONTOL statement for changing a partitions color is:

```
CONTROL CRT,5;Color
```

The Color parameter is an integer between 136 and 143.

136	WHITE
137	RED
138	YELLOW
139	GREEN
140	CYAN
141	BLUE
142	MAGENTA
143	BLACK

Keyboard Repeat Rate

When you press an alphanumeric key on the keyboard and hold it down for a certain length of time, the character output is repeated. This is equivalent to repeatedly pressing the key.

The value in this register is an integer from 0 thru 31 which corresponds to the delay and rate in the following table. Delay is the length of time in milliseconds which the key must be held down before the repeat begins. The rate is the length of time in milliseconds between each repeat of the key.

register value	delay	rate	register value	delay	rate	register value	delay	rate	register value	delay	rate
0	500	20	8	400	60	16	600	20	24	700	80
1	300	20	9	400	80	17	600	40	25	700	100
2	300	40	10	400	100	18	600	60	26	20	20
3	300	60	11	500	20	19	600	80	27	40	40
4	300	80	12	500	40	20	600	100	28	100	100
5	300	100	13	500	60	21	700	20	29	500	500
6	400	20	14	500	80	22	700	40	30	1000	1000
7	400	40	15	500	100	23	700	60	31	5100	5100

A delay rate of 20 or 40 milliseconds (values 26 and 27) is so short that you cannot enter a single character. At power up or SCRATCH A, the rate is reset to the value 0 (delay 500, rate 20).

msus

This is the select code of the default mass storage device. The default value of this register is ":CS80,7,1"; the internal flexible disc.

You set it with the SET_NON_VOL_MEM utility.

System IO Buffer

This register contains the size of the system IO buffer in number of sectors (1 sector = 256 bytes). The default value is 16 sectors which is equal to one track of a flexible disc.

The system uses this buffer when you access a program file (i.e. GET, LOAD, SAVE or STORE). The most efficient size of this buffer is equal to one track of the disc; 16 sectors for a flexible disc and 64 sectors for an HP 7912.

Service Only Registers

These registers contain information on the system's usage and error rates. They can be read but not changed by the user.

Checksum

These registers are a check on the validity of the values read in NVM.

MC Failure

This area contains an array of 4 bit counts of the number of times the healer CAM for a given memory controller filled up with single bit memory errors.

Double-bit Errors
This register contains information on the last double-bit error. The first byte contains the memory controller number. The last three bytes contain the address of the failure.

On Time
This register contains the number of minutes the compter has been on divided by ten.

Number of Overheat Cycles
This register contains the number of times the computer has powered itself down due to overheat.

Number of Power Up Cycles
This register contains the number of times you have powered up the computer.

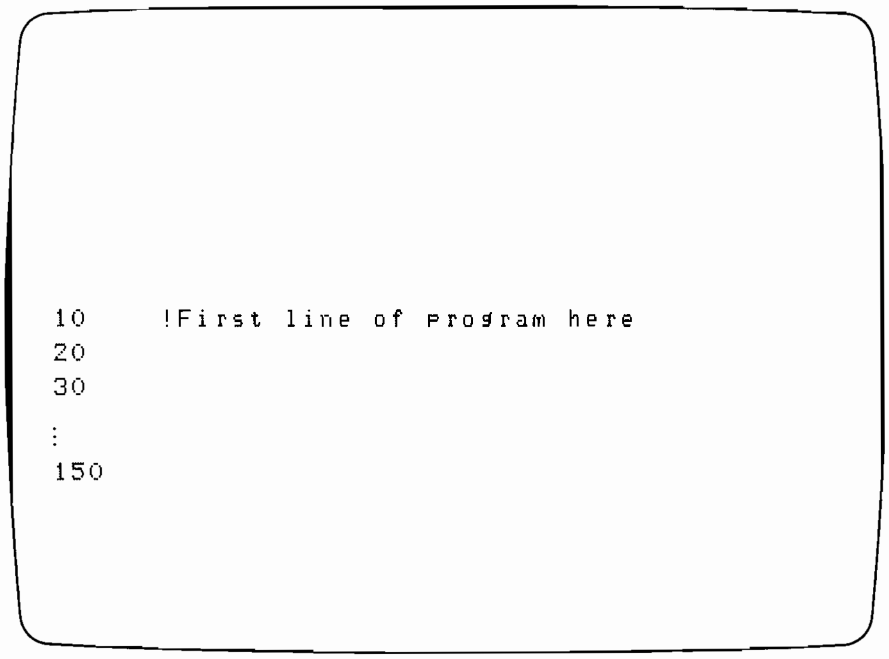
Chapter 3
Editing

The Editor is a tool designed to simplify program entry, debugging and program modification. The Editor is a full screen editor with syntax checking. When you are in the Editor, you simply type over the line you want changed, press RETURN, and the program line has been changed. You can use any of the editing keys and special function keys on the keyboard for editing. There are also several commands used for editing. These are shown in the following table.

To perform this operation:	Use this command:
Enter the Editor.	EDIT
Find and change one literal with another. (Also enters the Editor.)	CHANGE
Copy program lines from one location to another.	COPYLINES
Delete program lines.	DEL
Find a literal. (Also enters the Editor.)	FIND
Indent the program once and exit the Editor.	INDENT
Move program lines from one location to another.	MOVELINES
Renumber the program.	REN
Indicate the importance of spaces when checking the syntax.	SPACE DEPENDENT ON/OFF
Exit the Editor.	Any other command

How to Enter the Editor

To get into the EDIT mode, either press EDIT or type the word "EDIT" and then press EXECUTE. As a result, the current contents of the screen are removed. These contents reappear when you exit the Editor. The program currently in memory is displayed beginning in the middle of the CRT. If there is no program in memory, line number 10 is put in the middle of the CRT so that you can begin entering program lines.



```
10     !First line of program here
20
30
:
150
```

Memory Requirements

In most cases, the Editor uses the entire CRT. This requires approximately 5K bytes of memory. If there is insufficient memory available, the Editor uses a smaller portion of the CRT. If there is insufficient memory to create a screen of at least 5 lines, you cannot enter the Editor. You can free memory to use the Editor by stopping a running program or executing SCRATCH V.

Line Identifiers

The EDIT command has two optional parameters. The first is a line identifier, and the second is the increment between line numbers. For example, the command

```
EDIT 140,20
```

tells the computer to place the program on the CRT so that line 140 is in the center of the screen. Any lines that are added to the program get a line number 20 greater than the previous line.

If the increment parameter is not specified, the computer assumes a value of 10. Thus, the command

```
EDIT 1000
```

puts the line 1000 in the center of the CRT, and added lines get a line number of 10 greater than the previous line.

When the line identifier is not supplied, the computer has several ways of choosing a line number. If this is the first EDIT after a power-up, SCRATCH, SCRATCH A or LOAD, the line number is 10. If EDIT is performed immediately after a program has paused because of an error, the number of the line that generated the error is chosen. If EDIT is performed after a program is paused, the next line to be executed is chosen. If the program is stopped, the first line of the program is chosen.

The line identifier can also be a line label. For example, assume that you want to edit a sorting routine that begins with a line labeled "Go_sort". Simply type:

```
EDIT Go_sort
```

and press (EXECUTE). The line labeled "Go_sort" is placed in the center of the screen, the next fifteen lines (approximately) of the routine are displayed below it, and the previous fifteen lines are displayed above it.

If you execute EDIT while a program is running, the program PAUSEs and enters the Editor with the next line of the program to be executed in the middle of the CRT. When you exit the Editor with the CONT command, the program continues execution. While PAUSEd, however, there are some lines of the program which cannot be changed. For example, if you pause while in a FOR...NEXT loop, you cannot change the FOR statement.

Using CHANGE and FIND to Enter the Editor

CHANGE and FIND are two Editor commands which, when executed outside the Editor, cause the Editor to be entered. When you execute either of these commands, the Editor is entered and then the command is executed. These commands as well as the other Editor commands are discussed later.

Deleting an Old Program

You may want to enter a new program when there is an old program already in memory. Rather than deleting the program line-by-line, there is an easy method for deleting the entire program. The command SCRATCH clears the program from memory. You can type in the command or use the SCRATCH key and then press (EXECUTE).

If you don't want to disturb the current program, you can create a partition and run the Editor there. Partitions are discussed in the "Partitions and Events" chapter.

Editing the Text

The keyboard and its use for typing and editing on the input line was described in the "Getting to Know Your System" chapter. All of these normal editing features are available in the Editor plus some additional actions specific to programming.

You can edit any line of the program regardless of where it is on the CRT. The line you are editing is indicated by the cursor. After editing the line, press RETURN to store the line. If you do not want to store the line, use the arrow or roll keys to move the cursor to another line.

Editing Keys

There are several keys on the keyboard which you may use for editing. These keys are shown in the following table.

Key	Action
INS CHR	Used to toggle between the normal cursor and the insert cursor.
←	Moves the cursor one character to the left. If the cursor is at the first character of the line, it does nothing.
SHIFT ←	Moves the cursor to the beginning of the line.
→	Moves the cursor one character to the right.
SHIFT →	Moves the cursor to the end of the line.
BACK SPACE	Moves the cursor one character to the left. If the cursor is at the first character on the line, it does nothing.
DEL CHR	Deletes the character above the cursor.
CLEAR LINE	Clears the line on which the cursor is located
SHIFT CLEAR LINE	Deletes all the characters from the cursor to the end of the line.
TAB SET	Sets the tab.
TAB CLEAR	(SHIFT TAB SET) Clears the tab.
TAB	Moves the cursor to the next tab position.
SHIFT TAB	Moves the cursor to the previous tab position.

Scrolling the Program

The position of the program on the screen can be changed by using the ROLL ↑ and ROLL ↓ scrolling keys.

Key	Action
ROLL ↑	Moves the program up one line.
ROLL ↓	Moves the program down one line.
SHIFT ROLL ↑	Moves the program up approximately fifteen lines.
SHIFT ROLL ↓	Moves the program down approximately fifteen lines.

The (↑) and (↓) keys scroll the program up or down one line when the cursor is at top (for (↑)) or bottom (for (↓)) of the CRT.

By repeating the EDIT command with a line number, you can jump from one section of your program to another without having to scroll through the program.

Entering Program Lines

You enter program lines by typing them after the line number and pressing (RETURN). Before storing the line, the computer checks for syntax errors and converts alpha characters to upper or lowercase as required for names.

Line Numbers and Line Labels

Although the Editor supplies a line number automatically, you may change it. To change the line number, simply back up the cursor and type in the line number you want to use. Here are some points to keep in mind when changing the line number supplied by the Editor. Changing the line number of an existing line causes a copy operation, not a move. The line still exists in its original location. Entering the line number of an existing line replaces that line. Be careful that you don't accidently replace a line because of a typing mistake in the line number.

Line numbers have a range from 1 to 999 999.

Each line can have a label associated with it. A line label appears immediately after the line number and is followed by a colon. Line labels can be up to 15 characters long. This allows you to give names to sections of your program that are easy to remember and that have meaning. Some example line labels are listed below.

```
Start
Print_heading
Input_name
Error_routine
```

Line labels come after the line number and are followed by a colon (:). The first character is an upper case letter. Other characters can be lowercase letters, numbers, or the underscore character. A line label must have a statement following it. A line label cannot be on a line by itself.

Inserting Lines

Lines can be easily inserted into a program. As an example, assume that you want to insert some lines between line 90 and line 100 in your program. Place the cursor at line 100 and press the INS LN key. The program display "opens" and a new line number appears between line 90 and line 100. Type and store the inserted line in the normal manner. Another line number is displayed below the one you just inserted. The insert mode can be cancelled by moving the cursor to another line.

```
90
100 _!Watch this line
105
```

INS LN

```
90
95   _
100  !Watch this line
105
```

While inserting lines, the Editor maintains the established interval between line numbers if possible. If the interval between lines in the preceding example was 5, the first line number appearing would be 95. When the normal interval between lines can no longer be maintained, an interval of 1 is used. Thus, after line 95 is stored, the next line number supplied is 96. When there are no line numbers available between the line just inserted and the next line, enough of the program below the current line is renumbered to allow the insert operation to continue. In the example, this would happen after line 99 is stored. The original line 100 is renumbered to 101, and the number 100 appears after line 99.

```
90
95
96
97
98
99
100  _
101  !Watch this line
105
```

Automatic Indentation

Indenting the program improves its readability. If a FOR NEXT loop is indented, you are able to see that your FOR loop is ended with a single NEXT statement. You can indent entire subroutines to make it easy to find their beginning and end.

The Editor does some automatic indentation. As you enter program lines, the cursor is placed in the same column position as the beginning of the last line entered. Thus, if you begin line 100 in column 10, when line number 110 is displayed, the cursor is moved to column 10. You can begin typing the new statement, or you can move the cursor to a different column position. If you move the cursor, say to column 15, when you store the line and the Editor gives you the next line number, 120, the cursor is in column 15.

```
100   !Start of line
110   _
```

```
100   !Start of line
110       !Indent more
120       _
```

A line label is ignored when determining where the cursor is placed on the next line.

```
900 Label1: !Text
910         _
```

The INDENT Command

There is also a command which does indentation for you. INDENT looks for structured programming statements and indents their constructs. (The structured programming statements are described in the "Structured Programming" chapter of this manual.)

For example, enter the following lines:

```
10   !Example Indent
20   FOR I=1 TO 10
30   A=A+1
40   NEXT I
```

`INDENT` **EXECUTE**

Since the INDENT command exits the Editor, you need to execute `EDIT`. The program listing looks as follows:

```
10  !Example Indent
20  FOR I=1 TO 10
30    A=A+1
40  NEXT I
```

Syntax Checking

Syntax is a term used to describe the elements that compose a statement and their order in the statement. Immediate syntax checking is one advantage of interactive programming. If the syntax of the line is correct, the line is stored and the next line number appears in front of the cursor. If the system detects an error in the input line, it displays an error message immediately below the line and places the cursor at the location it blames for the error.

```
100  PRNIT A$

Improper Statement
```

Keep in mind that there is an endless variety of mistakes that can occur, and the system is acting on erroneous input. As a result, you might not always agree with its diagnosis of the exact error or the error's location. An error message, however, is proof that something needs to be corrected.

Spaces and Upper and Lowercase Characters

When you are typing statements, you do not always have to be concerned with upper and lowercase characters. Keywords, however, must always be typed in upper case.

The command SPACE DEPENDENT ON sets the mode which indicates that spaces are important. After a power-up, SCRATCH or SCRATCH A, SPACE DEPENDENT ON is set. There must be a space between keywords and other parts of the statement. Variables can be typed with the first character upper case and following characters either upper or lowercase letters, numbers or the underscore character. If a variable name is the same as a keyword, at least one of the characters in the variable name must be typed in lowercase.

SPACE DEPENDENT OFF sets the mode in which spaces are no longer important. Now variables must be typed with the first character upper case and all following characters lowercase letters, numbers or the underscore character. Spaces can appear anywhere, including in the middle of a variable name. Spaces are not necessary between keywords and other parts of the statement. When the statement is stored, the system puts in spaces where necessary and takes out unnecessary spaces.

When SPACE DEPENDENT is ON, the following results:

Entered	Parsed As
`ABCD=SIN(D)`	`Abcd=SIN(D)`
`ABCD = SIN ( D )`	`Abcd=SIN(D)`
`ABCD = SIN D`	`Abcd=SIN(D)`
`AB CD = SIN(D)`	`Error`
`A $ = ""`	`Error`

When SPACE DEPENDENT is OFF, the following results:

Entered	Parsed As
`ABCD=SIN(D)`	`Error`
`Abcd = SIN ( D )`	`Abcd=SIN(D)`
`A b c d = S I N ( D )`	`Abcd=SIN(D)`
`A b c d = S I N D`	`Abcd=SIN(D)`
`A $ = ""`	`A$=""`

Deleting and Recalling Lines

You can delete lines one at a time or a block of lines. The DEL LN key is used to delete the current line. If DEL LN is pressed by mistake, the line can be recovered by pressing RECALL then RETURN. The computer has a recall buffer that holds approximately 500 characters. So, if you have lines that are about 80 characters long, you can recall the last 6 lines which were deleted.

Try deleting a few lines and then pressing RECALL. The last line deleted is the first line recalled. You must press RETURN to store the line. Note that it does not matter where the cursor is placed when you do a RECALL. The line being recalled is stored in its proper location as indicated by its line number.

After you have recalled and stored a line, the next line you recall is the one you just stored. The recall buffer stores any line entered or deleted, not just the lines deleted.

The DEL Command

You can also use the DEL command to delete lines. When the keyword DEL is followed by a single line identifier, only a single line is deleted. The line identifier can be a line number or a line label. Unlike the DEL LN key the DEL command does not save the line(s) in the buffer.

You can delete blocks of program lines by using one line number or label to identify the start of the block to be deleted and another line number or label to identify the end of the block to be deleted. The line identifiers must appear in the same order as they do in the program.

For example:

```
DEL 100,200
```

deletes lines 100 thru 200 inclusively.

```
DEL Block2,999999
```

deletes all the lines from the one labeled "Block2" to the end of the program.

```
DEL 250,10
```

causes an error.

There is a DEL key on the keyboard. This key is not a typing aid. It generates the DEL character (CHR$ 127).

Editor Commands

There are several commands which make it easier to edit a program. One of these is the DEL command which you have just seen. Other commands are CHANGE, FIND, COPYLINES, MOVE-LINES, and REN (renumber).

Renumbering a Program

Renumbering is done by the REN command. You can renumber the entire program or portions of it. You specify the starting line number, the increment and the range of the renumbering operation. The starting line number is the number that the first line in the range becomes after the renumber. The increment is the interval between lines.

For example,

```
REN 100,5
```

renumbers all the lines in the program. The first line becomes line 100, the next line is 105, etc.

```
REN 200,5 IN 100,300
```

renumbers the lines between lines 100 and 300. The first line becomes line 200, the next line is 205, etc.

The renumber operation **cannot** move a segment of the program from one location to another. The order of the program lines remains unchanged with the REN command.

Line References

There are several commands which change the line numbers in the program. Whenever you use one of these commands, the Editor changes any references to a line which changed its line number.

For example:

```
45  IF A<B THEN GOTO 75
55  B=B+1
65  A=B
75  !Continue

REN

10  IF A<B THEN GOTO 40
20  B=B+1
30  A=B
40  !Continue
```

The reference in line 45 to line 75 is changed when the REN command is executed. Line 75 becomes line 40 so the reference to that line is now to line 40.

Moving Program Lines

If you have a section of your program that you want to move from one location to another, use the MOVELINES statement. For example,

```
MOVELINES 100,200 TO 300
```

moves the lines from line 100 to line 200 to the area of the program beginning with line 300. The lines after line 300 may have to be renumbered to make room for the lines being moved. Also, if there is a current line 300, it is renumbered to be after the lines moved.

After a MOVELINES operation, the lines exist in the new location and not in the old location.

```
90
100   !This is line 100
110   !This is line 110
 :
200   !This is line 200
210   !This is line 210
 :
290
300   !This is line 300
310   !This is line 310

MOVELINES 100,200 TO 300

90
210   !This is line 210
 :
300   !This is line 100
301   !This is line 110
 :
310   !This is line 200
311   !This is line 300
312   !This is line 310
```

Line number references to the moved lines and within the moved lines are updated.

Copying Lines

If you have a set of lines which are used in several places in the program, you can use COPYLINES to copy the lines. You specify the first and last lines to be copied and the target location.

For example:

```
COPYLINES 30,50 TO 100
```

copies lines 30 through 50 to location 100. Lines after 100 (and line 100 if any) may have to be renumbered to make room for the copied lines.

If there is a reference to the copied lines within the copied lines, the reference is updated to the new line numbers. References outside the copied lines are **not** changed. For example, assume the following lines are copied:

```
 1   IF A THEN 30
10   INPUT "NEXT";A$
20   IF A$=" " THEN 10
30   GOTO 200

COPY LINES 10,30 TO 75

 1    IF A THEN 30       ←——— This reference is not changed
10    INPUT "NEXT";A$
20    IF A$=" " THEN 10 ←——— This reference is not changed
30    GOTO 200
:
75    INPUT "NEXT";A$
85    IF A$=" " THEN 75 ←——— This reference is changed
95    GOTO 200
```

Finding a String

There are times when you want to find a string in the program. The FIND command allows you to look for a string through either the entire program or just a segment. For example,

```
FIND "Myname"
```

looks through the program beginning at the first line for the string "Myname". When the string is found, the line in which it occurs is placed in the center of the CRT and the cursor is displayed under the first character of the string. You can find the next occurrence of the string by pressing (STEP).

The line number at the beginning of each line is not searched for the string. You can, however, search for line numbers in a statement.

You can specify a section of the program to search, rather than searching the entire program.

```
FIND  "A STRING" IN 500, 700
```

searches lines 500 thru 700 for the string.

For example, enter the following lines:

```
500   !TEST      TEST                TEST
510   !    TEST            TEST
520   !TESTING    PROTEST     TESTIFY
```

Execute `FIND "TEST" IN 500,520`. Now press (STEP) repeatedly.

The cursor stops every time it finds "TEST" as a string or substring. After you press (STEP) the last time in line 520, the message "STRING NOT FOUND" is displayed.

Changing a String

When you want not only to find a string, but also to change it, the CHANGE statement is useful. You specify the string you want to find and the string to which you want to change it. You can look through the entire program or just a portion of the program. For example,

```
CHANGE "Myname" TO "Wendy" IN 100,200
```

looks through the lines beginning with 100 and ending with 200. If the string "Myname" is found, the string "Wendy" replaces it, but the line is not stored. You are given the chance to accept the change, by pressing RETURN, or to reject the change, by pressing STEP. In either case, the Editor searches for the next occurrence of the string. If you press any line editing key other than STEP or RETURN, the CHANGE mode is canceled.

For example, enter the following lines:

```
200 T=T+1
210 IF T>10 THEN GOTO Done
220 STOP
```

Now execute `CHANGE "T" TO "TEST"`.

All T's will be replaced with TEST line-by-line. In line 200, this is what you want. Press RETURN. In line 210, you do not want all T's changed. Delete the characters added to correct the line and press RETURN. In line 220 you do not want the change. Note there is no error because the parser assumes "Stestop" is a subprogram call. Press STEP to reject the change in line 220.

This example shows one problem with using a single character as a variable name, even if you plan to change the name later. You can use up to 15 characters to name a variable.

You can also search for and replace all occurrences of a string without having to press RETURN each time. Adding `;ALL` to the CHANGE command does this for you. For example,

```
CHANGE "A$" TO "A_real_name$";ALL
```

searches the entire program for the string "A$" and replaces it with "A_real_name$". If a syntax error occurs, the line in error is displayed so you can correct it. After you correct the error, press RETURN to store the line. The CHANGE command then continues, but without the ALL option. You can use the RECALL key and re-execute the original command if desired.

You can do a search and replace in just a portion of the program.

```
CHANGE "One" TO "Two" IN Second_routine,Third_routine
```

searches the portion of the program which lies between the line labels Second_routine and Third_routine.

The STEP Key

You use the STEP key in both the FIND and CHANGE commands. If you are currently changing text, STEP rejects a change and continues with the CHANGE command. In any other case, STEP re-executes either the last CHANGE or FIND command you executed. The most recently executed command is the one re-executed. For example, if you do a FIND, then a CHANGE and then some editing, when you press STEP, the Editor looks for the string in the CHANGE command.

If you execute a FIND or CHANGE and then do some editing, the next time you press STEP, the command continues but sometimes with different line limits. If the cursor is on a line past the last line specified in the command, the new limits are from the current line to the end of the program. If the cursor is on or above the last line specified in the command, the new limits are from the current line to the end limit specified in the command.

Saving Your Program

When you have finished editing your program, you want to save it in a file. There are several ways to do this. Only one method is discussed in this chapter. All methods are discussed in the "Working with Files" chapter in this manual.

To save your program, first choose a name. The name should be unique, ie. it does not exist as a file now. Next, execute the SAVE command using the name you have chosen.

```
SAVE "Myprogram:INTERNAL"
```

This saves your program on the disc that is in the internal disc drive. The program is still in the computer's memory too.

Note that a disc must be in the drive for this command to work. If you get an error, or you want to save you program on another media, refer to the "Working with Files" and "Mass Storage Organization" chapters.

To retrieve your program at a later time, put the disc in the disc drive (if it was removed) and execute:

```
GET "Myprogram:INTERNAL"
```

You can also use the STORE and LOAD commands to save and retrieve your program.

Listing A Program

All or part of a program can be displayed or printed by executing the LIST command. The LIST command allows parameters that specify both the range of lines to be listed and the device to which the listing should be sent. If the keyword LIST is executed without any parameters, the entire program is listed on the default printer. After a power-up, SCRATCH or SCRATCH A, the default printer is the CRT. (The default printer is defined with the PRINTER IS statement.)

You can specify starting and ending line numbers in the LIST statement. For example,

```
    LIST 100,200
```

lists lines 100 thru 200 inclusively.

```
    LIST 1850
```

lists the last portion of the program from line 1850 to the end.

You can list the program on the internal printer with the following command:

```
    LIST TO PRT
```

To list parts of the program to a printer use:

```
    LIST First_line, Last_line TO PRT
```

Directing the listing to other devices is easy, but involves concepts that have not been introduced yet. Refer to the "I/O Output" chapter for a complete discussion of listing programs.

Exiting the EDIT

There are many ways to terminate the EDIT mode. Your choice depends upon what you want to do next. Executing any command, other than an Editor command, exits the Editor. Editor commands are those described in this chapter.

(RUN), (STOP), (CONTINUE) and (PAUSE) exit the Editor and then perform their normal action on the program in memory.

Pressing (CLEAR SCN) exits the Editor without changing the state of the program in memory.

Editing in Color

You can change the color of the program lines if you have a color display. There are many uses for color in the program listing: such as putting comments in one color and other program lines in another color, putting passwords in black so they do not print, and putting literals in a PRINT statement in color rather that using control characters.

There are two ways to change the color. Using the CONTROL statement you change the color for all alpha displays. Using the color keys on the keyboard you can change the color for keyboard input.

The Color Keys

The color keys are the special function keys (8 24) thru (15 31). You press (CTRL) and the SFK at the same time to get the color change. The color name is printed on the SFKs.

(CTRL)	(8 24)	white
(CTRL)	(9 25)	red
(CTRL)	(10 26)	yellow
(CTRL)	(11 27)	green
(CTRL)	(12 28)	cyan
(CTRL)	(13 29)	blue
(CTRL)	(14 30)	magenta
(CTRL)	(15 31)	black

Commands and statements must appear in the color set for the alpha display. Only literals may be in an alternate color.

For example:

```
100    PRINT "RED"
```

↑ press (CTRL) (8 24) (or the default color) then type "

↑ press (CTRL) (9 25) then type the literal RED

The CONTROL Statement

The CONTROL statement sets the color for the alpha display.

```
CONTROL CRT,5;color
```

The color parameter is an integer which corresponds to one of the following colors.

136	white
137	red
138	yellow
139	green
140	cyan
141	blue
142	magenta
143	black

When you change the alpha color and then list a program, you may find that not every line has changed color as you expected.

In a line such as the previous PRINT example, the last quote mark ('') is listed in the previous alpha color. The Editor does not ''turn on'' the color red and then ''turn off'' the color red. It ''turns on'' red and then ''turns on'' white. The program line runs correctly, however, if you make a change to that line, you must change the last quote mark ('') to the new alpha color before storing the line.

A second PRINT statement using no specified color outputs in the old alpha color after the alpha color is changed.

For example:

```
CONTROL CRT,5;136  (set alpha color to white)
```

```
100    PRINT "(CTRL) (9  25) RED(CTRL) (8  24)"
110    PRINT "should be default color"
120    END
```

The output is:

```
RED                     (in red)
should be default color (in white)
```

```
CONTROL CRT,5;139 (set alpha color to green)
```

The output is:

```
RED                     (in red)
should be default color (in white)
```

To correct the color change, edit the program and end each line in the new default color.

Chapter 4
Program Execution

You run a program by either pressing (RUN) or executing the RUN command. The system performs a prerun on the program in memory and then executes the statements in the program. When a STOP or END statement is executed, the program stops running.

You do not have to run a program from its first statement to its last statement. The RUN command has a optional parameter with which you specify the first line on the program to be executed. The following examples show several RUN commands.

```
RUN 100         !Begin execution at line 100.
RUN Label       !Begin execution at the line labeled Label.
RUN             !Begin execution at the first line of the program.
```

You can stop a program at any time by pressing (STOP).

Prerun

When you run the program, the system performs a prerun on the MAIN context. Prerun for multi-line functions and subprograms is performed the first time the function or subprogram context is called during program execution.

Prerun consists of two primary operations: static matching and storage allocation.

Static Matching

The system scans the context (MAIN program, multi-line function or subprogram) for proper structure of static program constructs such as IF...THEN, FOR...NEXT, LOOP...END LOOP, etc.

An error is reported if the system detects an improper construct. This error is reported when the context containing the improperly matched construct is first entered. Compiled contexts do not require prerun static matching, as this is done during execution of the COMPILE statement.

Storage Allocation

The system scans the context for static storage allocation statements, such as DIM, COM, REAL and DOUBLE, and allocates the variables' space if needed.

Variables declared in any statement other than COM or ALLOCATE are always allocated space during prerun.

COM statements do not always require that new space be allocated. If the common block does not exist, the system allocates space for it. If the common block already exists, the system may change its size or leave it alone depending on rules explained in the "Storage Allocation" chapter.

The system does nothing with ALLOCATE statements at this time. Space for this statement is allocated dynamically when the statement is executed.

All static allocation statements are included in prerun regardless of where they appear in the context. If an allocation statement appears in a conditional construct, such as an IF...THEN...ELSE, the system allocates storage for it. The truth of the IF conditional expression is not considered. Similarly, when the program is running, static allocation statements are **not** executed.

For example,

```
10   DIM Array$(5,5)[80]        !This statement is executed at prerun
20   INPUT A
30   IF A=1 THEN
40      REAL C,E                !This statement is executed at prerun
50   ELSE
60      DOUBLE C,E              !This statement is executed at prerun
70   END IF                     !It causes ERROR 12 , attempt to redeclare variable
80
```

Program States

When you run a program, the program is in the RUN state. This state can change due to a statement within the program or due to a key being pressed on the keyboard. The different states in which a program can be are:

STOP - The program is stopped. Only RUN can start execution.

RUN - The program is running and executing statements.

PAUSE - The program was running but has paused due to a PAUSE statement, (PAUSE), or the occurrence of an error that was not trapped. The program may be continued with the CONT statement, (CONTINUE), or it may be stopped or rerun.

INPUT - The program is waiting for input (INPUT, LINPUT, EDIT or other input statements).

WAIT - The program is waiting. A variety of statements cause this state.

Each state is indicated in the RUN LIGHT screen. The default location of this screen is on the lower right portion of the CRT.

State	Run Light Indicator
STOP	(no character shown)
RUN	▒
PAUSE	–
INPUT	?
WAIT	▒

Program Waiting

When a program is waiting, it is not using the CPU. The program enters the WAIT state when it executes one of the following statements: WAIT (time), WAIT, WAIT FOR EOT, WAIT FOR EOR and WAIT FOR INTR. WAIT FOR EVENT, DBLOCK and LOCK may also cause a wait.

WAIT (time) — The program enters the WAIT state for the specified number of seconds. When that time has elapsed, the program continues execution on the next line.

WAIT — The program enters the WAIT state until an ON condition occurs. If the ON branch is a GOSUB or CALL, the return point for the user ON routine is the statement immediately following the WAIT statment. ON conditions are described in the "ON...Conditions" chapter of this manual.

WAIT is useful when waiting for external events without using the CPU resource(s). For example, a program that waits for the operator to press a SFK could use the following statements:

```
100   ON KEY 1,3 GOSUB Handle_Key_1
110   GOTO 110
  .
  .
  .
```

The problem with this approach is that, while the program is "waiting" for the user to press key 1, the CPU is very busy doing nothing. Because the Series 500 can execute several programs simultaneously, this is wasteful of a valuable resource.

A better approach is to use the WAIT statement.

```
100   ON KEY 1,3 GOSUB Handle_Key_1
110   WAIT
120   GOTO 110   ! Wait for another key press
  .
  .
  .
```

WAIT FOR EOT/EOR — The program enters the WAIT state until an end of transfer or an end of record occurs.

WAIT FOR INTR — The program enters the WAIT state until an interrupt from the specified select code.

WAIT FOR EVENT — The program enters the WAIT state if the event is less than one. (Events are described in the "Partitions and Events" chapter.) There is a conditional parameter in this statement. If the program specifies the parameter, it does not wait. The program exits the WAIT state when the event level becomes greater than zero.

DBLOCK — The program enters the WAIT state if it cannot lock the data base or data set. It exits the WAIT state when it can get the lock.

LOCK — The program enters the WAIT state if it cannot lock the file resource. It exits the WAIT state when it can lock the file.

Forms of the Program

There are three forms or types of code that the program can have: source code, intermediate code or compiled code.

When you write the program, you use **source code**. The code is in characters which you can read. This form of the program is never actually stored in the memory of the computer. The Editor displays source code. The Editor reads the intermediate code in memory and translates it to source code for display.

As you type in a line of the program, the system checks the line for correct syntax, and then changes the line from source code to **intermediate code**. The intermediate code is a compacted representation of the source code.

The system produces **compiled code** from intermediate code as it executes the program. In this case both the intermediate and compiled codes are present in memory. You can also change intermediate code to compiled code by using the COMPILE statement.

File Storage

The different forms of code can be stored in a file.

SAVE — translates intermediate code into source code and saves it in a DATA file. (The file type can also be BDAT, ASCII or BCD.)

GET — gets source code from the file, changes it to intermediate code and stores it in memory.

STORE — stores the intermediate code and/or compiled code in a PROG file. The compiled code which is stored consists only of the program contexts compiled with the COMPILE command. All other contexts are stored as intermediate code.

LOAD — loads the intermediate code and/or compiled code from the PROG file to memory.

Since the source code never appears in the computer's memory, you may wonder why you should save it. One reason is for transportability. Another reason is that when you compile a program, there is no way to translate the compiled code to source code. Thus, when you want to update the program, you must have the source code or intermediate code stored somewhere.

Program Execution

When you run a program, the system executes each statement line-by-line. Before a statement can be executed, it must be in compiled code. If you compiled the program, the system does not have to recompile the statements during execution. If the program is in intermediate code, then the system must compile each statement once.

Before executing the statement, the system looks at the statement. If it is compiled, then the system executes the statement. If the statement is in intermediate code, the system compiles the statement and then executes it. The compiled form of the statement and its intermediate code both remain in memory. The second time the statement is executed, the system does not have to recompile it.

When you re-run a program, the compiled code is still in memory. The program runs faster when re-run because the system does not have to recompile each statement.

Start of Line Check

Before the system executes a line it performs several checks.

The system looks for keyboard commands and other key presses. If the priority of the keyboard is equal or higher than the priority of the program, the command is executed.

The system also looks for ON conditions which may have occurred. For example, if an ON EVENT statement was executed earlier in the program and a CAUSE EVENT was executed by another program, then the branch specified in the ON EVENT statement is taken. There are many rules for when the branch is taken and when it is not. These are described in the "ON...Conditions" chapter.

Start of line check takes less than a microsecond if no events have occurred.

Busy Lines

There are times when the compiled code is thrown away. If you edit any line in a context, all the compiled code for that context is thrown away.

There are some lines in the program which you cannot edit while the program is running. These are called busy lines.

When the following program pauses at line 220, you could edit the MAIN program. This would cause the system to throw away the compiled code for the MAIN program. When the function FNA returns to the MAIN, the system recompiles lines 30 and 40 and resumes execution in the middle of the line. For this to work without error, the compiled code for lines 30 and 40 must not change when the recompilation occurs.

```
  1 !Main
 10    DEF FNX(A)=A+FNY(A)
 20    DEF FNY(B)=B+2
 30    DEF FNZ(C)=FNX(2)+FNA(C)
 40    Q=FNZ(5)
  :  More of the main program
190 END
200 DEF FNA(P)
210    DIM T$[1000]
220    PAUSE
230    RETURN P*3
240 FNEND
```

There are other lines in this program which are also busy.

- Lines 30 and 40 are busy because the lines have not completely executed yet.
- Line 200 is busy because it defines an active multi-line function.
- Line 210 is busy because all variable allocate statements in active subprograms are busy. (This is necessary to support recursion.)
- Line 10 is "implied busy" because modifying it could affect the reproducibility of the compiled code for a busy line (in this case, line 30).

Memory Management

The compiled code which the system created is thrown away if there is not enough memory to keep both the compiled code and the intermediate code.

During program execution, if the system runs out of memory, the system recovers memory in several ways.

1. The system attempts to extend the program segment to make room for new compiled code. This is done by using memory that other partitions are not currently using.
2. If the system cannot extend the program segment, it compacts all user program contexts and tries to execute the current line again.
3. If 2 fails, the system throws away all compiled code except for the current context (MAIN, subprogram or multi-line function). The system compacts all user program contexts again, reclaims excess space and tries to execute the current line again.
4. If 3 fails, the system throws away the compiled code for the current context, recompiles the current line and tries to execute the current line again.
5. If 4 fails, the system reports an ERROR 2, memory overflow.

Chapter 5
Structured Programming

The advantages of structured programming include top-down organization, efficiency in developing a program, ease of understanding and maintaining the program.

Specificly these advantages are:

- logical organization facilitates documentation, modification and maintenance;
- easy to learn and use;
- faster to debug than unstructured programs;
- powerful syntax;
- smooth program flow;
- reduced program development time.

These benefits come from using only a few programming constructs. A construct is a group of logically related BASIC statements. BASIC has three constructs for decision making and four constructs for loop control.

To perform this operation:	Use this construct:
Decision Making Constructs	
Execute one of two segments.	IF...THEN...ELSE...END IF
Branch to one of several segments.	ON...GOTO/GOSUB
Execute one of many segments.	SELECT...CASE...CASE ELSE...END SELECT
Loop Control Constructs	
Repeat specific number of times.	FOR...TO...STEP...NEXT
Repeat at least once and until a condition is true.	REPEAT...UNTIL
Repeat while a condition is true.	WHILE...END WHILE
Repeat a segment until one of many conditions is true.	LOOP...EXIT IF...END LOOP

In addition to these constructs, the INDENT command aids the readability of the program and helps in debugging.

Linear Flow

The simplest form of program flow is linear. Although linear flow is not glamorous, it has a very important purpose. Most operations required of the computer are too complex to perform using one line of BASIC. Linear flow allows many program lines to be grouped together to perform a specific task in a predictable manner. This form of flow requires little explanation, but keep the following characteristics in mind:

- Linear flow involves **no** decision making. Unless there is an error condition, the program lines involved in this type of flow always execute in exactly the same order, regardless of the results of or arguments in any expression.
- Linear flow is the default mode of program execution. Unless you include a statement that stops or alters program flow, the computer always executes the next higher-numbered line after finishing the line it is currently executing.

Branching

Branching involves executing lines in an order other than sequential order. An **unconditional** branch simply redirects the sequential flow of the program. **Conditional** branching involves decision making during program execution.

Unconditional Branching

GOTO, GOSUB, CALL and FN are the statements which redirect the program flow unconditionally.

CALL and FN invoke new contexts in addition to their branching action. This is covered in a separate chapter, "Subprograms and Functions".

GOTO

GOTO directs the program flow to a line other than the next higher line number. When execution is redirected by a GOTO, the program flow does not automatically return to the line following the GOTO. If you want a return, use the GOSUB statement.

GOSUB

Use the GOSUB statement to tranfer program execution to a subroutine.

You should know the differences between **subroutines** and **subprograms**. These differences are completely discussed in "Subprograms and Functions". The basic differences are mentioned here.

Calling a subprogram invokes a new context. A context is similar to a separate program, but you can return to the calling line. Subprograms can declare formal parameters and local variables.

A subroutine is simply a segment of a program that is entered with a GOSUB and exited with a RETURN. Subroutines are always in the same context as the program line that invokes them. There are no parameters passed.

The following example shows a subroutine:

```
 300  R=R+2
 310  Area=PI*R^2
 320  GOSUB Subroutl
 330  Width=Width+1
 340  Length=Length+1
  .
  .
  .
1000 Subroutl: PRINT Area;"square in."
1010  Cent=Area*6.4516
1020  PRINT Cent;"square cm."
1030  PRINT
1040  RETURN
```

Selection

Selection constructs provide a way to select a segment of the program to execute. The segment executed depends upon the results of a conditional test. Selection can be 1) whether or not to execute a program segment or 2) which one of several segments to execute.

There are three selection constructs, IF...THEN, ON...GOTO/GOSUB and SELECT...CASE.

Conditional Execution of One Segment

The basic decision of whether or not to execute a program segment is made by the IF...THEN statement. This statement includes a boolean expression that is evaluated as being either true or false. If **true** (non-zero), the conditional segment is executed. If **false** (zero), the conditional segment is bypassed.

The conditional segment can be either a single BASIC statement or a program segment containing any number of statements. First let's look at a single statement.

```
100  IF Ph>7.7 THEN PRINT "Value =",Value
```

When the computer executes this line, it evaluates the expression Ph>7.7. If the value contained in the variable Ph is 7.7 or less, the expression evaluates to 0 (false), and the line is exited. If the value contained in the variable Ph is greater than 7.7, the expression evaluates as 1 (true), and the PRINT statement is executed. If you do not understand logical and relational operators, refer to the "Numeric Computations" chapter.

The same variable can be on both sides of an IF...THEN statement. For example:

```
IF Number>9 THEN Number=9
```

When the computer executes this statement, it checks the initial value of "Number". If the variable contains a value less than or equal to nine, that value is left unchanged, and the statement is exited. If the value is greater than nine, the conditional assignment is performed, replacing the original value of Number with the value nine.

Prohibited Statements

Certain statements are not allowed as the conditional statement in a single-line IF...THEN. The disallowed statements are those identified during pre-run. These statements declare variables, define context boundaries, define program constructs or identify lines that are literals.

The BASIC Language Reference manual includes a line at the beginning of each keyword page which reads "In an IF...THEN". This identifies which statements can and cannot be used in a single-line IF...THEN statement.

The following statements cannot be used in a single-line IF...THEN:

COM	REM	SUB	FOR...TO...STEP	REPEAT
DOUBLE	DATA	SUB END	NEXT	UNTIL
INTEGER	IMAGE	DEF FN	LOOP	SELECT
REAL	PACKFMT	FNEND	EXIT IF	CASE
SHORT		END	END LOOP	CASE ELSE
OPTION BASE			IF...THEN	END SELECT
DIM			ELSE	WHILE
			END IF	END WHILE

Conditional Branching

In the previous section on branching the statements GOTO, GOSUB, CALL and FN were described as unconditional branching statements. They can also be specified in the IF...THEN statement which makes them conditional branches.

For example:

```
110   IF Free_space>100 THEN GOSUB Expand_file
```

One important feature of this structure is that the program flow is essentially linear except for the conditional "side trip" to a subroutine and back.

The conditional GOTO has a special syntax. Assuming that line 200 is labeled "Start", the following statements all cause a branch to line 200 if X is equal to 3.

```
IF X=3 THEN GOTO 200
IF X=3 THEN GOTO Start
IF X=3 THEN 200
IF X=3 THEN Start
```

When a line number or line label is specified immediately after THEN, the computer assumes a GOTO statement for that line. This is not the same as an identifier which appears at the beginning of a line. An identifier at the beginning of the line is an implied CALL.

For example:

```
350   Start
```

does a **CALL** to the subprogram Start, not a GOTO to the line Start.

Multiple-line Conditional Segments

If the conditional program segment requires more than one statement, you use a slightly different form of IF...THEN.

```
200  IF A=B THEN
210     PRINT "A=B=";A
220     GOSUB Equal
230  END IF
```

Any number of programs lines can be placed between a THEN and an END IF.

When using multiple-line IF...THEN structures, remember to mark the end of the structure with an END IF statement and do not put any of the statements on the **same line** as the IF...THEN.

The following example shows the **WRONG** way.

```
30  IF A THEN GOSUB True
40     PRINT "True"
50  END IF
```

The **RIGHT** way is:

```
30  IF A THEN
40     GOSUB True
50     PRINT "True"
60  END IF
```

The conditional segment can contain any statement except one which is used to set context boundaries (such as SUB or DEF FN). If a construct is used within a multiple-line IF...THEN, the entire structure must be contained in one conditional segment. This is called **nested** constructs. The following example shows some properly nested constructs. Notice that the use of indenting improves the readability of the code.

```
1000  IF Flag THEN
1005    Lines=0
1010    IF End_of_page THEN
1020      FOR I=1 TO Skip_length
1030        PRINT                      } FOR NEXT } IF THEN } IF THEN
1040        Lines=Lines+1
1050      NEXT I
1060    END IF
1070  END IF
```

The INDENT command indents a program, or you can indent as you input lines. The INDENT command is described later in this chapter.

Choosing One of Two Segments

Often you want a program to flow through only one of two paths depending upon a condition. You can do this using the IF...THEN...ELSE construct.

In the following example, the operator inputs a value representing interest. If the value is greater than one, the program divides it by 100 to get a percentage. If the value is less than or equal to one, then the value is already a percentage.

```
290  INPUT "Interest",Int
300  IF Int>1 THEN
310    Interest=Int/100
320  ELSE
330    Interest=Int
340  END IF
```

Notice that this structure is similar to the multiple-line IF...THEN shown previously. The only difference is the addition of the keyword ELSE. There can only be one ELSE in an IF...THEN construct. Like the previous example, the structure is terminated by END IF, and the proper nesting of other constructs is allowed.

If a construct is nested within the IF...THEN...ELSE, the entire construct must be in one of the segments. For example,

```
10 IF A THEN
20   FOR I=1 TO A
30     CALL Sub1
40   NEXT I
50 ELSE
60   CALL Sub2
70 END IF
```

Choosing One of Many Segments

The IF...THEN construct uses a boolean expression to determine which of two segments to execute. If the expression is true, the THEN segment is executed. If the expression if false, the ELSE segment, if any, is executed. There are times, however, when you need an expression which is not boolean and when you need more than two choices.

There are two constructs you can use: the ON...GOTO/GOSUB statement or the SELECT construct.

Using the ON Statement

There are two classes of statements in BASIC which begin with ON. One of these classes, referred to as ON conditions, name a condition such as ON DELAY or ON EVENT. These are described in the "ON conditions" chapter.

The other class is the ON...GOTO/GOSUB construct. This construct is a decision branch and is described here.

Assume you have a function which has either one, two or three parameters passed to it. The system function NPAR returns the number of parameters. You could use multiple IF...THEN statements to execute the program segment which handles each case, or you could use the structured ON statement.

```
770  DEF FNExample(A,B,C)
780  ON NPAR GOTO One,Two,Three
790 One:                    !Routine for one parameter
800      Value=A
810      RETURN Value       !Return to the calling program
820 Two:                    !Routine for two parameters
830      Value2=A+B
840      RETURN Value2       !Return to the calling program
850 Three:                   !Routine for three parameters
860      Value3=A+B+C
870      RETURN Value3       !Return to the calling program
880  FNEND
```

The value after the ON must be a positive integer and there must be enough line numbers or line labels after the GOTO or GOSUB to equal the highest value the integer can take. So in the previous example, if NPAR returns a four, an error occurs.

The ON statement is useful when you are expecting one of a sequence of numbers, such as the number of parameters passed to a subprogram. If the number becomes large, however, the ON statement becomes awkward to use.

Using SELECT

The SELECT construct also directs the program flow to one of several segments. It is easier to use than the ON statement because you do not have to use line labels or line numbers. The program automatically goes to the right segment by matching the SELECT expression with the CASE expression.

Unlike the ON statement, the SELECT construct jumps to the end of the construct when one of the segments has completed.

Using SELECT in the previous example results in the following:

```
780  SELECT NPAR
790  CASE 1 !Routine for one parameter
800    Value=A
810  CASE 2 !Routine for two parameters
820    Value=A+B
830  CASE 3  !Routine for three parameters
840    Value=A+B+C
850  CASE ELSE !Routine for anything else
860    Value=0
870  END SELECT
880  RETURN Value
```

Notice that the SELECT construct starts with a SELECT statement specifying the expression to be tested and ends with an END SELECT statement. The anticipated values are in CASE statements. Although this examples shows a numeric function being tested, the SELECT statement can contain numeric or string variables or expressions. The CASE statements can contain constants, variables, expressions, comparison operators or a range specification. The anticipated values, or **match items**, must be of the same type (numeric or string) as the tested expression.

The CASE ELSE statement is optional. It defines a program segment that is executed if the tested variable does not match any of the cases. If CASE ELSE is not included and no match is found, program execution simply continues with the line following END SELECT.

The following example shows a string variable tested with comparison operators and a range specifier.

```
 40  SELECT A$
 50  CASE <"A"          !If the value of A$ is less than "A",
       .                 this segment is executed.
       :
 90  CASE "A","B","C"   !If the value of A$ is "A","B" or "C",
       .                 this segment is executed.
       :
130  CASE "a" TO "z"    !If the value of A$ is between "a" and "z",
       .                 this segment is executed.
       :
200  CASE "D","E","F","O" TO "R"    !If the value of A$ is "D","E",
       .                             "F", or between "O" and "R",
       :                             this segment is executed. .
260  CASE Match$        !If the value of A$ is the same as the value
       .                 of Match$, this segment is executed.
       :
340  END SELECT
```

A common error when using the SELECT construct is to put a boolean value in the CASE statement. In the following example, the program segment following the CASE statements are never executed.

```
 20 SELECT A
 30 CASE A=2
      .
      :
 80 CASE A=3
      .
      :
130 END SELECT
```

If A equals 2, the expression A = 2 evaluates as a boolean and returns a 1. Since 1 does not equal 2, the program segment after CASE A = 2 is not exectued.

The proper method is to leave out "A=".

```
 20 SELECT A
 30 CASE 2
      :
 80 CASE 3
      :
130 END SELECT
```

Matching CASE Expressions
If two CASE expression match or one expression is included in another, the first CASE statement is used. For example:

```
CASE "A"
:
CASE "A" TO "Z"
```

If the SELECT expression is "A" then the segment following the first CASE statement is executed. The segment following the second CASE statement is not executed.

Complex SELECT Expressions
When the SELECT statement is encountered, all of the expressions in the construct (both the SELECT expression and all CASE expressions) are put in an object code buffer. This buffer can contain 2000 bytes of code. If the expressions don't fit, the "Statement too complex" error occurs at the SELECT statement. Any errors found in the expressions appear at the SELECT statement even when the error is actually caused by a CASE statement.

If your SELECT construct is too complex, you can split the large construct into several smaller constructs.

Repeating a Program Segment

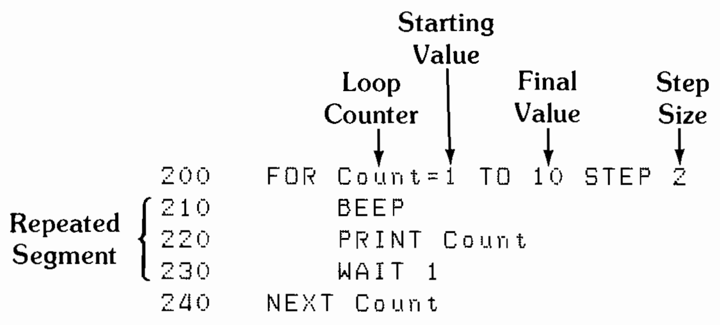
When you want to repeat a segment of your program several times, you don't want to duplicate that segment. To avoid this, you can put it in a construct. There are four constructs which do this.

Construct	Reason for Use
FOR...NEXT	Repeating a loop a specific number of times
REPEAT...UNTIL	Tests the condition **after** executing the loop
WHILE...END WHILE	Tests the condition **before** executing the loop
LOOP...EXIT IF...END LOOP	When there are multiple conditions which can result in the termination of the loop, or when the test occurs in the middle of the loop.

Repeating a Specific Number of Times

If you want to repeat the segment a specific number of times, you could put an IF...THEN statement in front of the segment, a GOTO at the end and increment a counter within the segment. Using the FOR...NEXT construct is easier.

With this construct, a program segment is executed a predetermined number of times. The FOR statement marks the beginning of the repeated segment and establishes the number of repetitions. The NEXT statement marks the end of the repeated segment. This construct uses a numeric variable as a **loop counter**. This variable is available for use within the loop, if desired. The following drawing shows the basic elements of a FOR..NEXT loop.



```
200   FOR Count=1 TO 10 STEP 2
210       BEEP
220       PRINT Count
230       WAIT 1
240   NEXT Count
```

The number of loop iterations is determined by the FOR statement. This statement identifies the loop counter, assigns a starting value to it, specifies the desired final value and determines the step size that is used to take the loop counter from the starting value to the final value. The counter is compared against the final value. If the counter is greater than the final value, the loop is not executed. If the STEP value is negative, the counter is tested to see if it is less than the final value.

The NEXT statement performs an increment to the counter and then compares the counter with the final value. In the preceeding example, the loop counter is given the value 1 when the FOR statement is encountered. The loop is then executed. When NEXT is encountered, the step size, 2, is added to the loop counter and the result is tested against the final value, 10. If the loop counter is less than or equal to the final value, the loop is repeated. If the loop counter is greater than the final value, the statement following the NEXT statement is executed.

The STEP value is an optional parameter. If it is not specified, a value of one is used.

Binary Arithmetic

The loop counter can be a REAL number, with REAL quantities for the step size, starting or final values. In some cases, using a REAL number causes the number of iterations to be different than what you would expect. This is because there is no way to exactly represent certain REAL numbers in binary arithmetic. You should use integer numbers for a predictable number of iterations.

In the following example, the variable I is just a little bit larger than 1 the last time though the loop. Rather than executing 101 times the loop is executed only 100 times.

```
10   FOR I=0 TO 1 STEP .01
20     PRINT I,
30   NEXT I
40   PRINT
50   FLOAT 16
60   PRINT "FINAL VALUE OF I=";I
70   END
```

```
0          .01        .02        .03
.04        .05        .06        .07
.08        .09        .1         .11
.12        .13        .14        .15
.16        .17        .18        .19
.2         .21        .22        .23
.24        .25        .26        .27
.28        .29        .3         .31
.32        .33        .34        .35
.36        .37        .38        .39
.4         .41        .42        .43
.44        .45        .46        .47
.48        .49        .5         .51
.52        .53        .54        .55
.56        .57        .58        .59
.6         .61        .62        .63
.64        .65        .66        .67
.68        .69        .7         .71
.72        .73        .74        .75
.76        .77        .78        .79
.8         .81        .82        .83
.84        .85        .86        .87
.88        .89        .9         .91
.92        .93        .94        .95
.96        .97        .98        .99

FINAL VALUE OF I= 1.000000000000006E+00
```

Variables in the FOR Statement

The starting value, final value and step size can all be variables or expressions. Since the FOR statement is executed only once, any variables in the FOR statement are evaluated only once.

For example, the following FOR...NEXT loops execute five times.

```
 50   A=1
 60   B=10
 70   C=2
 80   FOR I=A TO B STEP C
 90       C=C+1
100       PRINT I
110   NEXT I
```

```
 50   A=1
 60   B=10
 70   C=2
 80   FOR I=A TO B STEP C
 90       B=B-1
100       PRINT I
110   NEXT I
```

In both cases, the results are: `1  3  5  7  9`.

Even though no error is generated by this, it is not good programming practice to change the variable specifying the initial, final and step values.

Conditional Number of Iterations

The FOR...NEXT loop produces a fixed number of iterations established by the FOR statement before the loop is executed. Some applications need a loop that is executed until a certain condition is true without specifically stating the number of iterations involved.

The following segment asks the operator to input a positive number. A looping structure is used to repeat the entry operation if an improper value is given. Notice that it is not important **how many times** the loop is executed. If it only loops once, that is fine, or it can take ten or more times through the loop. It always executes at least once. What is important is that a **specific condition** is met. In this example, the condition is that a value be non-negative. As soon as that condition has been satisfied, the loop is exited.

```
800   REPEAT
810     INPUT "Enter a positive number",Number
820   UNTIL Number>=0
```

The loop is entered with the REPEAT statement, and the loop action is performed. When the UNTIL statement is executed, the condition is tested. In this way, a REPEAT...UNTIL loop is always executed at least once. If the condition is false, the loop is repeated; if the condition is true, the loop is exited.

The WHILE loop is used in the same manner as the REPEAT loop. The difference is the location of the conditional test. The WHILE loop has the test at the top of the loop; therefore, it is possible for the loop to be skipped entirely. In the WHILE construct, if the condition is true, the loop continues, and if the condition is false, the loop is exited.

For example:

```
720  WHILE Number>10
730     Number=Number DIV 10
740     Magnitude=Magnitude+1
750  END WHILE
```

Multiple Exit Points

There are times when more than one exit point is desirable. The REPEAT and WHILE loops allow only one exit point and one conditional test. The LOOP construct has multiple exit points and each one can have its own test.

For example:

```
100  A=0
110  B=12
120  LOOP
130    C=B-A
140    PRINT A,B,C
150  EXIT IF C=7
160    B=B-1
170    A=A+1
180  EXIT IF A>B
190  END LOOP
```

In the example, there are two exit points, line 150 and line 180. If the first test, C = 7, is true, the loop is exited. If it is false, then the next lines are executed. If the second test, A>B, is true, the loop is exited. If it is false, then the loop is executed again.

Nesting Constructs

Any construct can be nested within another construct. If you do this, the entire construct must be within one program segment of the other construct.

For example:

```
 30  IF A THEN
 40     WHILE B<C
 50       GOSUB Routine1
 60     END WHILE
 70  ELSE
 80     FOR I=1 TO 10
 90       SELECT B
100       CASE 1
110          GOSUB Routine2
120       CASE 2
130          REPEAT
140            B=FN(B)
150          UNTIL B>C
160       CASE ELSE
170          GOSUB Routine3
180       END SELECT
190     NEXT I
200  END IF
```

The INDENT Command

The INDENT command uses the defined structures described in this chapter to decide where to indent. If you refer back to the examples, you can see that the lines between a REPEAT and an UNTIL statement are indented. The lines between each CASE of the SELECT construct are indented.

You can use the INDENT command to make your program listing easier to read. This in turn makes it easier to debug. Since the lines between the boundaries of a construct are indented, it is easy to see if you included the boundary statements. If you do forget a boundary, the program cannot run. An error is displayed when you execute RUN.

The INDENT command indents the current program listing. It does not do automatic indentation as you add lines to the program.

Look at the following two listings and decide which one is easier to read.

```
 10  FOR Count=1 TO 68            10  FOR Count=1 TO 68
 20  IF Count<Lines THEN          20    IF Count<Lines THEN
 30  READ #1;A$                   30      READ #1;A$
 40  PRINT A$                     40      PRINT A$
 50  Lines=Lines+1                50      Lines=Lines+1
 60  ELSE                         60    ELSE
 70  SELECT Last$                 70      SELECT Last$
 80  CASE "None"                  80      CASE "None"
 90  FOR I=1 to Lines             90        FOR I=1 to Lines
100  PRINT                       100          PRINT
110  NEXT I                      110        NEXT I
120  CASE "Page"                 120      CASE "Page"
130  PRINT                       130        PRINT
140  PRINT "Page";Page           140        PRINT "Page";Page
150  PRINT                       150        PRINT
160  CASE "Foot"                 160      CASE "Foot"
170  PRINT "1.";Foot$            170        PRINT "1.";Foot$
180  PRINT                       180        PRINT
190  END SELECT                  190      END SELECT
200  PRINT PAGE                  200      PRINT PAGE
210  END IF                      210    END IF
220  NEXT Count                  220  NEXT Count
```

The INDENT command has two optional parameters. The first specifies which column to start the first line in each context. The second specifies how far to indent.

For example,

```
INDENT 10,5
```

indents the program beginning in column 10 and each indent is 5 columns away from the previous line's indentation.

```
INDENT
```

the start column is column 9 and the incremental indent is 2 columns.

Chapter 6
Subprograms and Functions

Subprograms and user-defined functions are powerful structured programming tools. A subprogram can do everything a MAIN program can do; however, it must be invoked or called before it is executed, whereas a MAIN program can be executed by pressing (RUN).

There are several benefits available with subprograms.

- The subprogram allows you to take advantage of the top-down method of designing programs. In this technique, the problem to be solved is broken up into a set of smaller and more easily solvable problems.
- By separating all the details of performing the subtasks from the overall logic flow of the MAIN program, the program is much easier to read.
- Finding bugs in a program is time consuming. By using subprograms and testing each one independently, it is easier to locate and fix problems.
- Often, you may want to perform the same task from several different areas of the program. One subprogram can be executed for all cases. Each time the task is performed, a call is made to the subprogram.
- Libraries of commonly used subprograms can be assembled for widespread use. Many different users solving diverse types of problems may require some identical subprograms.
- Subprograms can be compiled separately from the MAIN program.
- Subprograms can be loaded and deleted during program execution, saving memory.

Statement Summary

The following table shows the statements discussed in this chapter.

To perform this operation:	Use this statement:
Mark the beginning of a subprogram.	SUB
Mark the beginning of a function.	DEF FN
Mark the end of a subprogram.	SUBEND or SUBEXIT
Mark the end of a function.	FNEND
Exit from a function.	RETURN
Invoke a subprogram.	CALL
Invoke a function.	FN
Define variables in common.	COM
Store a subprogram or function.	STORESUB
Load a subprogram or function.	LOADSUB
Delete a subprogram or function.	DELSUB
Compile a subprogram or function.	COMPILE

Some Startup Details

Subprograms are located after the body of the MAIN program. A subprogram logically begins with a SUB statement and ends with a SUBEND statement. A multi-line function begins with a DEF FN statement and ends with a FNEND statement. When compiled, a subprogram begins with the SUB statement and ends with the last statement before the next SUB or FN statement. Single-line functions only have a DEF FN statement, and they can be defined anywhere. Subprograms or multi-line functions may not be nested within other subprograms or multi-line functions.

A subprogram is invoked with a CALL statement. It is exited with a SUBEXIT or SUBEND statement.

A function is invoked with a FN statement and is exited with a RETURN statement.

Subprograms and multi-line functions can be invoked from the MAIN program or any subprogram or multi-line function. Single-line functions can be invoked only from the context in which they are defined.

The term subprogram is used throughout this chapter. The term refers to both subprograms and multi-line functions.

Naming

A subprogram has a name which may be up to fifteen characters long. The first character of the name is an uppercase letter; following characters are lowercase letters, numbers or the underscore character. National and line drawing characters, CHR$(161) through CHR$(254) can be used in both the initial and following characters of the name. String functions have a $ following their names.

For example,

First$
Initialize
Sort_array
Read_2

Example

The following is an example of a program which calls subprograms.

```
100   !Main
110      COM #1,Date$
120      INPUT "File to be printed",File$
130      ASSIGN #1 TO File$
140      PRINTER IS PRT
150      CALL Print_header
160      CALL Print_file
170      CALL Print_trailer
180      ASSIGN #1 TO *
190   END
200   SUB Print_header
210      COM #1,Date$
220      Date$=DATE$(TIMEDATE)
230      PRINT TAB 40,Date$           !The printer is still set
240      ASSIGN #2 TO Header$         ! to PRT
250      FOR Line=1 TO 10
260         READ #2,Line$
270         PRINT Line$
280      NEXT I                       !File #2 is closed upon exiting
290   SUBEND                          !from the subprogram
300   SUB Print_file
305      DIM Line$[80]
310      COM #1
320      ON END #1 GOTO 370
330      LOOP
340         READ #1,Line$             !#1 was assigned in the MAIN
350         PRINT Line$
360      END LOOP
370      !end of file
380      PRINT
390   SUBEND
400   SUB Print_trailer
410      COM #1,Date$
420      ASSIGN #2 TO Trail
430      PRINT TAB 15,Date$           !The value of Date$ is in common
440      FOR I=1 TO 3
450         READ #2,Trail$
460         PRINT Trail$
470      NEXT I
480   SUBEND
```

Program Context

A **context** represents a particular invocation of a subprogram. It includes the execution state of the subprogram itself and the binding between memory locations and variable names resulting from a particular call to that subprogram. Different subprograms represent different contexts. Furthermore, a given subprogram may represent several contexts at a given time because of recursion. A local variable in that subprogram may have a different value for each level of recursion. In a given context, the variable has only one value.

Recursion is supported by the BASIC Language system and occurs when a subprogram calls itself. For example:

```
160 SUB Two
170   INPUT "Next Character";A$
180   IF A$="S" THEN
190     SUBEXIT
200   ELSE
210     CALL Two
220   END IF
230     PRINT A$;
240 SUBEND
```

This subprogram reads characters from the keyboard. If the character is an S the subprogram exits. If the character is anything else, the subprogram calls itself. Assume the operator inputs A, B, D and S in that order. The subprogram prints the following:

```
D   B   A
```

The value of A$ is retained for each invocation of the subprogram.

Assume a Main program declares local variables A, B, C, and D, and subprogram One declares local variables A, B, C and D. The variables A, B, C, D in the Main program are independent of the variables A, B, C, D in subprogram One.

```
 10 !Main
 20 Start:   A=1
 30          B=2
 40          C=3
 50          D=4
 60          CALL One
 70          PRINT A;B;C;D
 80 END
 90 SUB One
100 Start:   A=5                !Line labels in the MAIN program
110          B=6                !can also appears in subprograms.
120          C=7
130          D=8
140          PRINT A;B;C;D
150 SUBEND
```

This program prints two lines. The first line printed is the values of the variables in the subprogram. The second line is the values of the variables in the MAIN program.

```
5  6  7  8
1  2  3  4
```

When a context is called, either through a CALL or through a RUN command, the context is put on a stack within memory. The following example shows a program on the left of the page and a figure on the right. The figure represents the stack in memory as the program is running. The stack contains the location to which to return when the context has finished executing, the context's local variables and its execution stack.

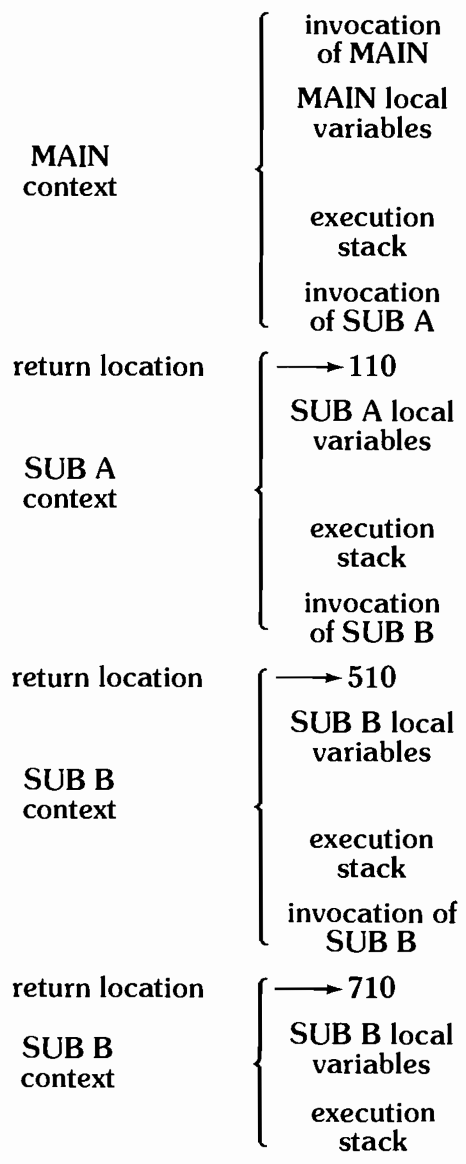
1. When the MAIN program is run, the first context is put on the stack. Since there is no line number that called the MAIN, there is no return pointer.
2. Line 100 calls subprogram A. The first portion of the stack that contains the MAIN context remains on the stack. The context for SUB A is added to the stack. Since line 100 called SUB A, when SUB A exits, execution continues with line 110.
3. Line 500 in SUB A calls SUB B. The context for SUB B is added to the stack.
4. Line 700 in SUB B calls SUB B. A new context for SUB B is added to the stack.

At this time, the stack appears as follows:

PROGRAM

```
 10 !MAIN
 20     INTEGER A,B,C
  ⋮
100     CALL SUB A
110     !
  ⋮
200     CALL SUB B
210     !
  ⋮
400 END
410 SUB A
420     INTEGER D,A,B
  ⋮
500     CALL SUB B
510     !
  ⋮
580     SUBEXIT
  ⋮
630 SUBEND
640 SUB B
650     REAL K,I,L
  ⋮
700     IF A THEN CALL SUB B
710     !
  ⋮
800     SUBEXIT
  ⋮
860 SUBEND
```

STACK



5. When SUB B exits, the return pointer is used to find the next line to execute. SUB B's context is removed from the stack.
6. SUB B exits. Its context data is removed from the stack.
7. SUB A exits. Its context data is removed from the stack.
8. The MAIN exits. Its context data is **not** removed from the stack. The MAIN program's local variables are still accessable until you execute a SCRATCH command, GET, LOAD, or RUN.

Functions

You can use the FN invocation in a numeric or string expression the same way you use a constant or variable. A function's purpose is to return a single value (either a number or a string).

For example,

```
50 A=SIN(X)+FNUser_defined(Y)
```

There are several functions built into the BASIC language such as SIN, SQR, RPT$ and CHR$. These are referred to as **system functions**. The system functions available in BASIC are described in the "Numeric Computations" and "String Operations" chapters.

There are two types of user-defined functions, **single-line** and **multi-line** functions. A single-line function is completely defined in one BASIC program line. A multi-line function uses more than one line. You invoke them in the same way. The difference between them is that a single-line function does not define a new context and a multi-line function does.

You can only invoke a single-line function from the context in which it is located. You can invoke a multi-line function from any context, including itself.

The following example shows a program which uses both multi-line and single-line functions.

```
100  DEF FNParity(A)=A MODULO 2
110  FOR I=1 TO 10
120      PRINT I,FNParity(I),FNFactorial(I)
130  NEXT I
140  END
150  DEF FNFactorial(A)
160    IF A=0 THEN RETURN 1
170    RETURN A*FNFactorial(A-1)
180  FNEND
```

Subprogram or Subroutine

Do not confuse subprograms with subroutines. A **subroutine** is part of the context from which it was invoked. It does not have its own context. A subroutine shares data and line labels with the rest of the context; if the subroutine changes the value of a variable, the variable is changed in the rest of the context also. A subroutine is invoked with a GOSUB statement and is exited with a RETURN. It can only be invoked from the context in which it is located.

A **subprogram** defines a new context. It can share data with other contexts and can have its own local variables. It has local line labels. It is invoked with a CALL statement and exited with a SUBEXIT or SUBEND. It can be invoked from any context.

Sharing Data Between Contexts

There are two primary ways for a subprogram to share information with the MAIN program or with other subprograms: parameter lists and common variables.

Parameter Lists

The formal parameter list is part of the subprogram's definition, as is the subprogram's name. The formal parameter list tells how many values may be passed to a subprogram, the types of those values (string, INTEGER, DOUBLE, REAL, SHORT, I/O path name, file number) and the names the subprogram uses to refer to those values. The calling context provides a pass parameter list which corresponds with the formal parameter list provided by the called subprogram. The pass parameter list provides the values for those parameters required by the subprogram and also provides the storage for any output values. It is perfectly legal for both the formal and pass parameter lists to be null or empty.

Here is a sample formal parameter list showing the types of each parameter:

```
    SUB Read_dvm(@Dvm,A(*),INTEGER Reg,#1,Status$,Errflag)
```

@Dvm is an I/O path name which may refer to either an I/O device, a mass storage file or a BUFFER.

A(*) is a REAL array. Its size is declared by the calling context. The entire array is passed. If only a specific portion of the array is going to be used by the subprogram, that portion has to be described in another parameter, in this case, Reg. The subprogram has the responsibility of using only the specified portion.

Reg is an INTEGER. When the calling program invokes this subprogram, it must supply either an INTEGER variable or a numeric expression which the system converts to an integer.

#1 is a file number. The calling context may assign a file to this number. The number in the calling context does not have to match the number in the subprogram.

Status$ is a simple string.

Errflag is a REAL number. All numeric variables after a string, file number or path name are REAL unless otherwise specified.

Pass by Reference or Value

There are two ways for the calling context to send values — **pass by value** and **pass by reference**. Using pass by value, the calling context supplies a value. If the subprogram changes the value, the calling context's variables are not affected. Using pass by reference, the calling context actually gives the subprogram access to the calling context's variable. The distinction is that a subprogram cannot alter the value of data in the calling context if the data is passed by value, while the subprogram can alter the value of data in the calling context if the data is passed by reference.

The subprogram has no control over or knowledge of whether its parameters are sent using pass by value or pass by reference. That is determined by the calling context's pass parameter list.

In order for a parameter to be passed by reference, the pass parameter list (in the calling context) must use a variable for that parameter.

In order for a parameter to be passed by value, the pass parameter list must use an expression which is not a simple variable or array element. A simple variable or array element is passed by value by placing parentheses around the variable or array element.

Using pass by value, it is possible to pass an integer expression to a REAL formal parameter (the expression is coerced to REAL) without causing a mismatch error. Likewise, it is possible to pass a real expression to an INTEGER formal parameter; although, the coercion may cause an overflow error. Using pass by reference, the parameters must match exactly.

Substrings (eg. A$[5,10]) are always passed by value since they are expressions.

Arrays are always passed by reference.

The CALL corresponding to the previous example appears as follows:

```
    CALL Read_dvm(@Volt,Readings(*),(Register),#8,Stat$,Err)
```

@Volt is the pass parameter which matches the formal parameter @Dvm in the subprogram. I/O path names are always passed by reference, which means the subprogram can close the I/O path or assign it to a different file or device.

Readings(*) matches the array A(*) in the subprogram's formal parameter list. Arrays are always passed by reference.

Register is passed by value to the formal parameter Reg. If the value of Reg is changed in the subprogram, the value of Register is not affected.

#8 is the file assigned to number 8. In the subprogram, #1 is used to reference this file. File numbers are always passed by reference, which means the subprogram can close the file or assign it to a different file.

Stat$ is passed by reference here. If this variable is used to pass status information back to the MAIN program, it would be useless to pass the variable by value.

Err is passed by reference.

OPTIONAL Parameters

An important feature of formal parameter lists is the OPTIONAL keyword. Any formal parameter list (the one in the subprogram defining line) may contain the keyword OPTIONAL, although it isn't required. The OPTIONAL keyword indicates that any parameters that follow it are **not required** in the pass parameter list of a calling context. On the other hand, all parameters preceding the OPTIONAL keyword are required. If no OPTIONAL appears in the subprogram's parameter list, then all the parameters must be specified, or an error occurs.

When the MAIN program or a subprogram is compiled, there may be a CALL to a subprogram which is not currently in memory. When this occurs, there cannot be any buffer parameters or optional parameters in the formal parameter list (of the subprogram not in memory) and all formal parameters associated with numeric parameters passed by value must be of type REAL.

The system function NPAR returns the number of parameters actually passed to the current subprogram. NPAR returns a 0 if used inside the MAIN program or if no parameters were passed to a subprogram.

For example,

```
220   CALL Write(#6,Stet,Bul,Dot,Indent)
 .
 .
 .
700   SUB Write(#1,A,B,C,OPTIONAL D,Ret$)
710   Parameters=NPAR

Parameters=5
```

Common Blocks

The other method a subprogram has for communicating with the MAIN program or with other subprograms is with COM blocks. Common is not used to pass values between partitions.

There are two types of COM blocks, labeled and blank (unlabeled) common. Blank COM is a special case of labeled COM; it has no label, and it must be declared in the MAIN program. Both types of COM blocks declare blocks of data which are accessible to any context having matching COM declarations. Access to blank common is faster than access to labeled common.

When declaring a COM block, the context specifies each variable and the variable's type and size. For example:

```
COM  Conditions(15),INTEGER Cin,Caz,@Plane,Zkz$[35]
```

The COM block can be defined using multiple COM statements if necessary. Each COM block must be completely defined before you begin defining the next COM block.

Each COM block is accessible after one context has declared it. Thus, common declared in a MAIN program is accessible to all subprograms. When a subprogram declares a COM block, after the subprogram has executed once, that COM block is available to all other contexts.

When a context accesses a COM block, it can either specify part of the block or the entire block. Matching is done by position and type, not by name. For example:

```
!MAIN
COM /First/A$,B$,INTEGER Next(12,12),Totals,REAL Variance,Results(5,5)
COM /Second/D,E,F,G,H,I
:
SUB Use
COM /First/Use1$,Use2$,INTEGER Arr(12,12),A,REAL Var1,Results(*)
COM /Second/B,C
:
```

Note in the subprogram Use the array Arr has a stated size of (12,12) and Results has no stated size. When a size is stated, it must be the same as the size specified in the COM statement which first declared the array. When the size is not stated, (*) is used, then the array size matches any size specified in the declaring COM statement. If the size is not stated in the declaring COM statement, an error occurs.

There are several characteristics of COM blocks which distinguish them from parameter lists as a means of communications between contexts.

COM survives pre-run. In general, any numeric variable is set to 0, strings are set to the null string and I/O path names and file numbers are set to undefined when that COM block is defined. After the COM block variables are defined, they retain their values until:

1. SCRATCH A, SCRATCH C or SCRATCH P is executed;
2. you modify a statement declaring the COM block and then re-run the program;
3. a new program is brought into memory using a GET or LOAD command which doesn't declare that COM block.

COM can be used to communicate between programs which overlay each other using LOAD or GET statements.

1. COM blocks which match each other exactly between the two programs are preserved intact. **Matching** requires that the COM blocks are named identically, that corresponding blocks have exactly the same number of variables declared and that the types and sizes of these variables match.

   For example;

   ```
   COM /Matching/ A,B,String$[25],Array(10,10)
   COM /Matching/ Numb1, Numb2, Other$[25], And(10,10)
   ```

2. Any COM blocks existing in the old program which are not declared in the new program (the one being brought in with the LOAD or GET) are lost.

3. Any COM blocks which are named identically, but which do not match variables and types identically, are defined to match the definition of the new program. The data area is not reinitialized. Thus, the variables in the new COM statement may have values other than zero.

   For example;

   ```
   COM /Identical/ A$,B$,Next(15,20)
   COM /Identical/ A,B,String$
   ```

   When the second COM statement is executed, the variables A, B and String$ have values.

4. Any new COM blocks declared by the new program (including those mentioned above in #3) are initialized implicitly. Numeric variables and arrays are set to zero, strings are set to the null string and I/O path names and file numbers are set to undefined (closed).

The declaration of a COM block can take as many statements as necessary. COM statements can be located anywhere in the program. One limitation on parameter lists (both pass and formal parameter lists) is that they must fit into a single program line along with the line's number, possibly a label and the subprogram header. Depending upon the situation, this can impose a restriction on the size of your parameter lists.

COM blocks can be used for communication between contexts that do not invoke each other. Information such as modes and states can be an integral part of communicating between contexts, even though those contexts don't explicitly call each other.

COM blocks can be used to communicate between subprograms that are not in memory simultaneously. Similar to the case above, subprograms can communicate with each other through COM blocks even though combinations of LOADSUB/DELSUB may preclude their simultaneous presence in memory. Data to be shared between the subprograms is stored in common.

COM blocks can be used to retain the value of "local" variables between subprogram calls. In general, the variables used by a subprogram are discarded when the subprogram is exited; however, there are situations where it might be useful for a subprogram to "remember" a value. These values are saved in common.

System Subexit Support

The BASIC system performs many operations at the exit of a subprogram. Many modes are reset to the calling context. (The Master Reset table in the BASIC Language Reference manual shows the conditions which are reset at subprogram exit.) Any variables ALLOCATEd in the context are implicitly DEALLOCATEd. Also, if the subprogram allocated any I/O path names or file numbers, the system unlocks and closes them if the context failed to do so. For example, if a subprogram opens a file and writes to it, the system posts buffers associated with that file and closes the file if needed. Note that if a context references an I/O path name or file number and passes it to another context to be opened, that I/O path name or file number belongs to the context that first referenced it.

For example, in the following program sections, @FILE is closed by the system when the SUB Owner exits in line 1030.

```
        .
        .
        .
1000  SUB Owner
1010    CALL Open_file(@File)
1020    OUTPUT @File;"THIS FILE IS MINE"
1030  SUBEND
1040  SUB Open_file(@My_file)
1050    ASSIGN @My_file to "DATAFILE";FORMAT ON
1060  SUBEND
        .
        .
        .
```

If an error occurs during the system subexit process, the behavior of the system depends on the nature of the subexit. There are two classes of subexits: normal and abortive.

A **normal subexit** is caused by RETURN(), SUBEXIT, and SUBEND. If an error occurs during a normal subexit, the system reports the error and pauses at the statement that caused the subexit, unless an active ON condition is present to handle the exception.

For example, suppose there is a file @A allocated and opened in a subprogram. When the SUBEND statement is encountered, the system attempts to post @A's buffers and finds that the disc is no longer in the disc drive. An ERROR 80 occurs and the program is paused at the SUBEND statement. You may mount the correct disc and press CONTINUE. The system again attempts to post @A's buffers, this time successfully. Note that this is exactly the behavior you would expect if the statement was ASSIGN @A TO *.

An **abortive subexit** is caused by pressing STOP, executing an END or STOP statement in the subprogram or by the program taking an ON...RECOVER branch.

The ON...RECOVER construct definition is that the branch should occur regardless of where the program is currently executing, and that the branch **must** be taken if the current priority is less than the priority specified in the ON condition statement. This implies that the branch cannot tolerate errors causing a state change or another branch. (STOP) is also abortive: the system does not cause your program to branch because of an error being detected while processing (STOP). The STOP statement and the END (in a subprogram) behave exactly as (STOP) key does.

When an error occurs during an ON ERROR RECOVER abortive subexit, the error is displayed on the screen. It does not effect ERRM$, ERRN, or ERRL. Otherwise, the system would be destroying the reason that the ON ERROR RECOVER branch was taken.

The STOP and ON...RECOVER subprogram exit are performed on ALL contexts between the current program execution location and the outermost level where the cause occurred. This means that all contexts are exited in this manner for a (STOP), and all contexts between the defining ON...RECOVER and the current program execution location for an ON...RECOVER branch.

Subprogram Libraries

You can store subprograms, the MAIN program or the entire program. Another program can then load the subprograms it needs. When a program is finished with a subprogram, it can delete the subprogram from memory to save memory space.

Storing Subprograms

There are two commands you can use to store subprograms. STORESUB and COMPILE. Compiling subprograms is discussed in a later section. STORESUB stores a compiled subprogram, but it does not do any compiling itself.

You store one subprogram at a time with the STORESUB statement.

```
STORESUB Subprog TO Library$
```

If the file already exists and there is sufficient room in the file to add the subprogram, the subprogram is added to the file. If the subprogram name already exists in the file, the old subprogram is deleted, and the new subprogram replaces it. The old subprogram is deleted only after the new subprogram is stored, so there must be room for both versions in the file.

If the file does not exist, the system creates a PROG file and stores the subprogram.

You can also store the MAIN program or all subprograms and the MAIN with this statement.

```
STORESUB MAIN TO File$
STORESUB ALL TO Sub_file$
```

When you store a subprogram, all statements beginning with the SUB statement and ending one line before the next SUB statement are stored. For example:

```
100   !Main
 :
210   END
220   !Comments
230   SUB First
 :
280   SUBEND
290   !Comments
300   !Comments
310   SUB Second
 :
500   SUBEND
510   !Comments
```

`STORESUB MAIN TO File$`	Lines 100 through 220 are stored. This is the first line of the program to one line before the first SUB statement.
`STORESUB First TO File$`	Lines 230 through 300 are stored. This is the SUB statement defining the First subprogram to one line before the next SUB statement.
`STORESUB Second TO File$`	Lines 310 through 510 are stored. This is the SUB statement defining the Second subprogram to one line before the next SUB statement.
`STORESUB ALL TO File$`	Lines 100 through 510 are stored. This is the first line of the program to the last line of the program.

Refer to mass storage chapters for a discussion of file names, files, mass storage devices and storage techniques.

Loading Subprograms

Once the subprograms are stored with either STORESUB or COMPILE, any program can load the subprogram and then CALL it.

For example, to load the subprogram Timer from the file Sublibrary put the following statement in the program.

```
LOADSUB Timer FROM Sublibrary$
```

To load all the subprograms from the file Sublibrary use the following statement.

```
LOADSUB ALL FROM Sublibrary$
```

If a MAIN program is in the file, it is **not** loaded. Only subprograms and multi-line functions are loaded.

LOADSUB adds the subprograms to the end of the program. The subprogram line numbers are not used, so you do not have to consider the line numbers when you store the subprogram. Once the subprogram is loaded, the program uses a CALL to execute the subprogram.

Deleting Subprograms

You may need to delete a subprogram from memory once you have finished with it. To do this, use the DEL SUB statement.

```
DEL SUB Timer, FNFunction, First
```

Note that the DEL SUB statement does not affect stored files.

`DEL SUB Sub TO END` deletes all subprograms from Sub to the end of the program.

Compiling Subprograms

You cannot modify a compiled program unless you go back to the source code. While this may seem to be a drawback of compiling, it is also an advantage. It is a security feature. No one else can change your code. Compiled programs also cannot be listed; only the first line of the program and subprograms are listed. Thus, no one can "steal" your code.

Compiled programs also run faster than non-compiled programs. Referring back to the "Program Execution" chapter, a source program must be compiled by the system before it can execute. A program that is already compiled does not have to go through another compilation.

The COMPILE command is similar to the STORESUB statement. COMPILE, however, is only keyboard executable.

```
COMPILE Subprogram1 TO Sub_library$
COMPILE ALL TO Library$
COMPILE MAIN TO Name$
```

If the file name does not exist, the compiler creates a PROG file. If the file already exists, the compiler adds the subprograms to it if the file is big enough. If the subprogram name exists, the new subprogram is added and then the old subprogram is deleted.

When ALL is specified, all subprograms, including ones already compiled, are stored in the file.

Restrictions

When running a compiled MAIN program, you cannot specify a line number in the RUN command.

When you compile a subprogram and then load it, you do not have access to the intermediate or source code. (Refer to "Program Execution".) As such, you cannot list or edit the lines in the subprogram.

Tracing with a compiled subprogram gives subprogram names instead of line numbers. Trace statements cannot appear in the subprogram which is being compiled.

GET statements also cannot appear in the subprogram being compiled.

When a CALL is made from the subprogram being compiled to a subprogram which does not exist in memory, the compiler makes assumptions about the missing SUB statement. The SUB statement can not include any BUFFER parameters and all pass parameters associated with numeric parameters passed by value are of type REAL. The number of parameters in the SUB statement must match the number of parameters in the CALL. That is, if there is an optional parameter in the SUB statement, the CALL **must** pass that parameter.

For example:

```
CALL Sub_a(1,2,3,4)
SUB Sub_a(A,B,OPTIONAL C,D)
```

The previous pair of statements is correct. The following pair of statements would cause an error when the CALL statement is executed.

```
CALL Sub_b(1,2,3,4)
SUB Sub_b(A,B,OPTIONAL C,D,E)
```

To avoid this problem, you can include the SUB statement in memory (type in the SUB statement if it isn't already there). You do not have to compile the SUB statement when you compile the context with the CALL.

Chapter 7
ON... Conditions

This chapter discusses a class of branching and program control statements which have unique properties not available with the simple branching and looping statements previously introduced. With these statements you can:

- execute a program branch when a condition occurs, not just at the next IF or SELECT statement;
- detect a condition which cannot be expressed as a Boolean, numeric or string value, such as an ERROR;
- suspend your program until a condition occurs, without consuming CPU time in a "wait" loop;
- branch on the completion of an I/O operation without waiting for that operation to complete.

This chapter discusses the following classes of statements:

- ON... statements, which specify a branch-on-condition;
- OFF... statements, which deactivate one or more ON... statements;
- CAUSE... statements, which simulate a condition specified in one or more ON... or WAIT FOR... statements;
- WAIT and WAIT FOR... statements, which suspend your program until an ON... condition occurs.

The Conditions

A program branch can occur on any combination of the conditions summarized in the following tables.

Error Conditions

When you want to detect:	Test this condition:
Any BASIC run-time error,	ERROR
An end-of-file (EOF) ERROR 59 or 60,	END
The failure of an I/O operation to complete in a specified time period (ERROR 168).	TIMEOUT

Chronological Conditions

When you want to branch:	Test this condition:
At a specific time of day within 24 hours,	TIME
At a specific date and time,	TIMEDATE
Once, after an elapsed time,	DELAY
Repeatedly, after elapsed time intervals,	CYCLE

Inter-task Communication Conditions

To branch on or suspend until:	Test this condition:
A signal from within your own program,	SIGNAL, EVENT
A signal from another program,	EVENT

I/O Related Conditions

To branch on or suspend until:	Test this condition:
An end of record (EOR) during a TRANSFER,	EOR
The end of a TRANSFER operation,	EOT
The pressing of a button or key on a graphics input device,	GKEY
An interface card interrupt,	INTR
The pressing of any key on the keyboard,	KBD
The pressing of a Special Function Key (SFK),	KEY

Start Of Line Check

When one or more ON... conditions are active in your program's current context, the BASIC Language System performs a "start of line check" (SOLC) at the beginning of each program line. The SOLC consists of checking for any condition which may have occurred during the execution of the previous statement.

To see the effect of condition servicing, consider the following program lines. In this example, a critical subroutine ("Service") must be performed at a specific time, denoted by the variable "When". Without ON... branching, you would have to write it like this:

```
30  IF TIMEDATE>=When THEN  GOSUB Service
31  FOR Index=1 TO Double
40  IF TIMEDATE>=When THEN  GOSUB Service
41    Array(Index,1)=SQR(SIN(LOG(Array(Index,2))))
50    IF TIMEDATE>=When THEN  GOSUB Service
51  NEXT Index
```

With ON... branching, the roughly equivalent program looks like this:

```
20  ON TIMEDATE When GOSUB Service
30  FOR Index=1 TO Double
40    Array(Index,1)=SQR(SIN(LOG(Array(Index,2))))
50  NEXT Index
```

Not only is a single ON... statement easier to type in than numerous IF statements, but also the SOLC consumes much less (CPU) time than an IF statement.

An ON... branch can occur several times. When an ON... branch occurs, the fact that it occurred is cleared by the BASIC system, but the ON... statement is not deactivated. The branch occurs again when the condition occurs again. Of course, in the above example the TIMEDATE condition cannot occur again unless you re-execute the ON TIMEDATE statement for a different value of "`When`".

Most ON...GOSUB and ON...CALL conditions which occur at line *n* are detected just before line *n + 1* and return to line *n + 1* from the subroutine or subprogram service routine. Because SOLC is performed at a line boundary, the fact that a condition occurred during a statement is recorded (logged), but the branch may not occur until the beginning of the next statement. Particularly time-consuming statements or I/O input statements such as

```
PRINT Array(*)
```

and

```
INPUT "Enter the quantity",Quantity
```

can "hold-off" ON... branching for considerable periods of time.

Not all ON... conditions are held off during statement execution. An exception to the SOLC check is the ON ERROR... statement. Errors caused by the currently executing statement are detected when they occur and not at the start of the next line. In ON ERROR...GOSUB and ON ERROR...CALL branches, the subroutine or subprogram returns to the beginning of the same line which caused the error. If your service routine does not correct the cause of the error, it can occur again, and your program may loop indefinitely.

Errors vs Asynchronous Conditions

All ON... conditions fall into one of two categories: **synchronous**, referred to as error conditions, and **asynchronous** conditions.

The Error Conditions

The occurrence or simulation of an **error** condition is called synchronous because it must be serviced by an ON... statement when it occurs; that is, the servicing must be in "sync" with the occurrence. These conditions are also called error conditions because they represent either program or I/O errors. If the condition cannot be serviced immediately, your program pauses and displays an error message. The following table summarizes the error conditions.

Condition	Related Statements
END	CAUSE END, ON/OFF END, WAIT
ERROR	ON/OFF ERROR, WAIT
TIMEOUT	ON/OFF TIMEOUT,WAIT

The Asynchronous Conditions

The occurrence or simulation of an **asynchronous** condition is so called because it does not need to be serviced by an ON... statement when it occurs; that is, the servicing can be out of "sync" with the occurrence. The condition is not serviced immediately, but the fact that it occurred can be recorded by the system in a "log bit". See the "Logging of ON... Conditions" section in this chapter. The following table summarizes the asynchronous conditions.

Condition	Related Statements
CYCLE	ON/OFF CYCLE, WAIT
DELAY	ON/OFF DELAY, WAIT
EOR	CAUSE EOR, ON/OFF EOR, TRANSFER, WAIT FOR EOR, WAIT
EOT	CAUSE EOT, ON/OFF EOT, TRANSFER, WAIT FOR EOT, WAIT
EVENT	CAUSE EVENT, CREATE EVENT, DEL EVENT, EVENT LEVEL, ON/OFF EVENT, WAIT FOR EVENT, WAIT
GKEY	GKEY, GRAPHICS INPUT IS, ON/OFF GKEY, WAIT
INTR	CAUSE INTR, DISABLE INTR, ENABLE INTR, ON/OFF INTR, WAIT FOR INTR, WAIT
KBD	KBD$, ON/OFF KBD, RESUME/SUSPEND INTERACTIVE, WAIT
KEY	ON/OFF KEY, RESUME/SUSPEND INTERACTIVE, WAIT
SIGNAL	ON/OFF SIGNAL, SIGNAL, WAIT
TIME	ON/OFF TIME, ON/OFF TIMEDATE, WAIT
TIMEDATE	ON/OFF TIME, ON/OFF TIMEDATE, WAIT

Identifying the ON... Condition

The syntax of some ON... statements (see the BASIC Language Reference manual) can specify a **condition identifier**, such as ON SIGNAL 3... This statement branches only upon the occurrence of SIGNAL 3 and not on SIGNAL 4.

Other ON... statements do not specify a condition identifier, such as ON ERROR. This statement branches upon the occurrence of any error (unless it is an END or TIMEOUT error handled by an ON END or ON TIMEOUT statement).

Types of ON... Branches

An ON... statement specifies one of four types of branches: GOTO, GOSUB, CALL and RECOVER. They offer either a non-returnable branch to a program line or a branch from which the service routine can return to the line at which the condition was detected.

There are other differences between the branches. The following table summarizes these.

	GOTO	RECOVER	GOSUB	CALL
Return to line which detected condition	NO	NO	YES	YES
Branch enabled in nested contexts	NO	YES	NO	YES

Skipping Program Lines - ON...GOTO

A typical use for ON...GOTO is to skip program lines which are expected to occasionally cause an error, or exit a loop on an expected error. For example:

```
   .
   .
   .
  310  DISP "Enter name or number"   !Prompt user
  320  LINPUT Input$                 !Read response as characters
  330  Input$=TRIM$(Input$)          !Remove leading/trailing blanks
  340  ON ERROR GOTO Is_a_name       !Is a name if VAL causes an error
  350  Number=VAL(Input$)            !Number input? (may cause error)
  360  GOTO Is_a_number              !   Yes, number input
   .
   .
   .
```

or

```
   .
   .
   .
  920  ASSIGN #4 TO "Data_file:INTERNAL"         !Open the file
  930  ON END #4 GOTO Close_file                 !Branch on EOF
  940 !
  950 Read_record:LOOP   !Loop to read records   !Until EOF "trap"...
  960    READ #4;Record$                           !Read a record
  970    CALL Process_record                       !Work with it
  980  END LOOP                                  !Get next record
  990 !
 1000 Close_file:     !End of file detected      !EOF Detected
 1010  OFF END #4                                !Cancel the branch
 1020  ASSIGN #4 TO *                            !Close the file
```

The ON...GOTO branch specifies a program **line number** or **line label**. The program line must be in the same context as the ON...GOTO statement. When an ON... condition triggers the branch, program execution goes directly to the specified program line. ON...GOTO provides no method for returning to the line at which the condition was detected.

An ON...GOTO branch is held off while the program is executing in a nested context other than the one which executed the ON...GOTO. See the "Deferred Branches" section of this chapter.

Abort Processing - ON...RECOVER

ON...RECOVER is similar to ON...GOTO except that ON...RECOVER is not disabled when the program changes contexts. A typical use for ON...RECOVER is abort processing. In the following example, the program re-starts at line "Start_prog" when you press (8 24) (SFK 8).

```
380  ON KEY 8 LABEL "ABORT prog",15 RECOVER Start_prog
```

The ON...RECOVER branch specifies a program **line number** or **line label**. The program line must be in the same context as the ON...RECOVER statement, but the program need not be executing in that context when the ON... condition occurs. When the ON... condition triggers the branch, program execution goes directly to the specified program line.

If the program is executing in a different context when the ON... condition occurs, BASIC performs as many SUBEXIT's as necessary to return to the ON...RECOVER context. This process closes files and I/O path names opened in each exited context. If ON... conditions or errors result from these operations, they are **ignored**. ON...RECOVER provides no method for returning to the context and line at which the ON... condition was detected.

Local Branch and Return - ON...GOSUB

A typical use for ON...GOSUB is to perform a task unrelated to the currently executing code. The following ON CYCLE statement "interrupts" the program every ten seconds to execute a subroutine which takes a reading from an instrument:

```
110  ON CYCLE 10 GOSUB Take_reading    !Read every 10 seconds
```

The ON...GOSUB branch specifies a program **line number** or **line label**. The program line must be in the same context as the ON...GOSUB statement. When an ON... condition triggers the branch, program execution goes directly to the specified line. When the subroutine executes a RETURN statement, program execution returns to the line at which the condition was detected or to the same line which caused an ON ERROR branch.

An ON...GOSUB branch is held off while the program is executing in a context other than the one which executed the ON...GOSUB. See the "Deferred Branches" section of this chapter.

Global Branch and Return - ON...CALL

ON...CALL is similar to ON...GOSUB. The difference is that ON...CALL is not held off when the program changes contexts. In the following example the ON SIGNAL branch can be taken during subroutine "Compute" but the ON CYCLE branch cannot.

```
1070  ON SIGNAL 3 CALL Trigger_scope
1080  ON CYCLE 10 GOSUB Take_reading
1090  CALL Compute(P1,P2,X3)
1100  Search=SQR(Range)
```

The ON...CALL branch specifies a **subprogram name**. The program need not be executing in the same context that defines the ON...CALL branch. When an ON... condition triggers the branch, program execution goes to the specified subprogram. When the subroutine executes a SUBEXIT or SUBEND statement, program execution returns to the context line at which the condition was detected. Note that unlike a CALL statement, you must explicitly specify the keyword CALL and cannot specify any parameters.

Deferred Branches

Simply executing an ON... statement once in your program does not guarantee that the occurrence of that ON... condition always results in an immediate branch. The occurrence can also be lost, cause an error or result in a delayed branch.

When or whether the branch occurs depends on the **state** of the ON... statement. The state can be: **inactive**, **active but disabled** or **active and enabled**. The state is affected by the type of ON... condition, the context and system priority. These topics are discussed in this section. The effect of the state is summarized in the following table.

	Condition Category	
ON... state	**Error**	**Asynchronous**
Inactive	Error occurs	Ignored becomes enabled
Active but disabled	Error occurs	Logged until statement becomes enabled
Active and Enabled	Immediate branch	Immediate branch

The table does not apply to conditions which occur during overlapped I/O. These are discussed in the "Branches Deferred Due to Overlapped I/O" section of this chapter.

The Inactive State

At system power-up, ON... processing for all conditions is **inactive**. The error conditions cause errors, and the asynchronous conditions are ignored.

When an ON... statement is executed in your program, the ON... processing state for the specified condition is activated. That ON... statement is said to be **active**. You can return it to the inactive state by an OFF... statement specifying the same condition or by exiting the ON... statement context.

An ON...CALL or ON...RECOVER can be temporarily disabled in a subordinate context by an **OFF**... statement executed in that context. The disabled ON... condition is re-activated upon exiting (SUBEXIT/SUBEND, RETURN...) that context.

ON... conditions made active by an ON... statement in a SUB or function are returned to the inactive state upon exiting that context. If you have not de-activated any ON... statements with OFF... statements, the SUBEXIT, SUBEND or RETURN... does it for you.

```
100   ON KEY 7 LABEL "EXIT" GOTO Close    !ON KEY is active
 :
230   OFF KEY                             !ON KEY is inactive
240   ON SIGNAL 4 CALL Fixit              !ON SIGNAL 4 is active
 :
370   CALL New_context
 :
500   SUB New_context
510      OFF SIGNAL                      !ON SIGNALs disabled
520      ASSIGN #4 TO "Data"
530      ON END #4 GOTO Close_file       !ON END is active
 :
720      SUBEXIT                         !ON END is inactive
730   SUBEND
```

There are potential side effects. Context exit statements also close or de-allocate resources opened or allocated in the subprogram context. This closure occurs before the implied OFF.... If, for example, a TRANSFER was in progress for an I/O path name assigned in the subprogram, and you had an ON EOT... statement active and enabled in the subprogram, the closing of the I/O path name causes an EOT. This EOT occurs **before** the ON EOT is de-activated. The ON EOT... branch is taken. If it is an ON EOT...GOSUB... or ON EOT...CALL...., it returns to the SUBEXIT or RETURN... and you exit the subprogram. If it is an ON EOT...RECOVER..., you may never return to the SUBEXIT or RETURN... unless the RECOVER specifies a line in the ON...'s context.

ON... conditions are also de-activated by STOP and END statements in the MAIN program. Since the STOP and END statements also release resources, similar side effects are possible. These side effects do not occur when you execute a STOP or END **within** a subprogram or function, nor when you press (STOP) at any time.

The Active State

When you execute an ON... statement, you **activate** that ON... condition. This is not fully sufficient to ensure that the branch can occur; an **active** ON... statement can be **disabled**. ON... statements are disabled by any of the following.

- The DISABLE statement disables all **asynchronous** ON... conditions in all contexts of your program. It does not affect error conditions; that is, DISABLE does not disable ON ERROR, ON END or ON TIMEOUT conditions.
- If the ON... statement specified a GOTO or GOSUB branch, it is disabled whenever the program is executing in any other **context**. This does not affect ON...CALL and ON...RECOVER except that no ON... statement executed in a subprogram or multi-line function is ever **active** in a higher context, such as its caller or the MAIN program. RETURN... and SUBEXIT/SUBEND deactivate all ON... statements executed in the subprogram or function.
- If the **priority** specified in the (asynchronous) ON... statement is less than the **system priority** when the condition occurs, the ON... statement is disabled. Priority does not affect **error** conditions, which have an implied priority of 16, always higher than system priority.

For example:

```
300  ON SIGNAL 1 GOTO Start
310  ON SIGNAL 5,10 CALL Fixit
 :
440  DISABLE                     !disables ALL ON conditions
450  !
460  !  Code you don't want interrupted
470  !
580  ENABLE                      !Enables all previously disabled
 :
810  SUB Sub_context
820                               !ON conditions with GOTO disabled
830      SYSTEM PRIORITY 15       !ON conditions with lower priority
840                               ! disabled
850      ON KBD GOSUB Key_in
 :
980      SUBEXIT                  !ON conditions defined in this context
990   SUBEND                      ! are inactive
```

The Enabled State

An **active** ON... statement is **enabled** when it is not otherwise disabled by the DISABLE statement, context or priority.

System Priority

Each ON... statement has an implied or specified **priority**. The ON... statement is disabled unless this priority is greater than the system priority. An ON... statement naming an error condition, such as ON END..., has an implied priority of 16. An ON... statement naming an asynchronous condition can specify its priority, such as `ON SIGNAL 3,Priority`..., or it can use the default priority.

There are two types of priority in the BASIC Language System: **partition** priority and **system** priority. Each priority is independent of the other. Partition priority is the priority of your entire program with respect to programs executing in other partitions. Partition priority is discussed in the "Partitions and Events" chapter.

System priority applies only within a partition and is the priority at which your program's current context is executing. Pre-run sets your program's system priority to zero. You can change system priority with a SYSTEM PRIORITY statement. The system priority changes automatically when an ON...GOSUB or ON...CALL branch occurs. You can determine the current system priority with the SYSTEM$("SYSTEM PRIORITY") function.

As discussed in a previous section, one of the conditions required for an ON... statement to be **enabled** is that the priority it specifies must exceed the current system priority. For ON... statements specifying asynchronous conditions you can specify a priority in the range 1 thru 15. ON... statements specifying error conditions have an implied priority of 16 and thus are never disabled due to system priority.

When the condition occurs, and the enabled ON... branch is taken, the system priority changes. This change is not permanent. Its duration depends on the type of branch.

For ON...GOTO and ON...RECOVER branches the program line specified in the ON... statement executes at an **effective** priority of 16. The system priority does not actually change. This first line of the service routine cannot be interrupted by another ON... condition. This gives you an opportunity to execute a SYSTEM PRIORITY statement to change the priority, or a DISABLE statement to hold off other ON... conditions. After executing that line, the system priority is reduced to its value at the time the ON... condition occurred (unless the line is a SYSTEM PRIORITY statement). The priority specified in the ON... statement only affects whether the ON... is enabled or disabled.

For ON...GOSUB and ON...CALL branches the system priority is changed to the ON... priority during the time the subroutine or subprogram service routine is executing. In order to allow error conditions to interrupt one another, the ON... priority of ON ERROR, ON END and ON TIMEOUT is reduced from 16 to 15 within the service routine. When RETURN, SUBEXIT or SUBEND is executed, the system priority is reset to its value at the time the ON... condition occurred.

For example:

```
100    ON ERROR CALL Fixit
110    ON KEY 1 LABEL "PRINT" GOTO Print_it
120    ON KEY 2 LABEL "NEW PROGRAM",12 CALL New_prog
 :
550 Print_it:    DISABLE        !This statement is uninterruptable
                                !Priority is previous system priority
 :
700    SUB Fixit                !Priority is 16
                                !Priority is 15
 :
740    SUBEND                   !Priority returns to previous
                                !System Priority
 :
900    SUB New_prog             !Priority is 12
 :
940    SUBEND                   !Priority returns to previous
950                             ! system priority
```

Logging ON... Conditions

The introduction to "Deferred Branching" noted that **asynchronous** ON... conditions can be **logged** if they occur while their ON... statement is active but disabled. This is not true of the error conditions. If an error cannot be immediately serviced by an ON... branch, your program pauses and displays an error message.

Each asynchronous ON... condition is represented in the BASIC Language System by a **log bit**. A log bit is similar to a Boolean variable. The test performed by the SOLC is, in effect, an `IF Log_bit THEN...` branch. When an ON... condition occurs, and an ON... statement is at least **active** for that condition, the BASIC system "logs" the condition by setting the log bit true. If the ON... statement is also **enabled**, it executes the branch immediately and clears the log bit.

If the ON... statement is disabled when the log bit is set true, the condition is not lost. The log bit remains true until one of the following occurs:

- the ON... statement becomes enabled, at which time the branch executes, setting the log bit false (clears the bit);
- an OFF... statement de-activates the ON... statement and sets the log bit false (clears the bit).

Like Boolean variables, log bits have only two states: true and false. If the condition occurs more than once while its ON... statement is disabled, only one branch occurs when the statement becomes enabled.

In the following example, an `ON CYCLE 1 GOSUB Cycle` could cause a branch to occur every second, but because it is a GOSUB branch, it is disabled while the program is executing in subprogram "Waiter". During "Waiter", 29 or 30 CYCLEs occur, but only one is logged. When "Waiter" SUBEXITs, the SOLC performed in line 630 detects the occurrence of a CYCLE and executes a **single** `GOSUB Cycle`.

```
610  ON CYCLE 1 GOSUB Cycle   !GOSUB Every second if possible
620  CALL Waiter(30)          !Invoke new context
630  GOTO Next_phase          !Continue in original context
640 !
650  SUB Waiter(Seconds)      !This subroutine just waits
660    WAIT Seconds           !and holds off ON GOTOs & GOSUBs
670    SUBEXIT                !while in this context
```

Nesting ON... Conditions

If an ON... statement is active for a specific condition and you execute another ON... statement specifying the same condition, the second statement overrides the first and replaces the first statement's condition value, priority and branch with the new value, priority and branch.

If both ON... statements were executed in the same context, this replacement is permanent. If the first ON... statement was executed in a different context than the second, the replacement is temporary.

In the following example, the statement

```
420  ON SIGNAL 4 CALL Service_4
```

applies during the execution of subprogram "New_context" only during the SOLC performed in line 490. When line 490 of subroutine "New_context" executes, the branch `CALL Any_signal` applies to SIGNAL 4. After line 520 of subroutine "New_context" (the SUBEXIT) executes, the branch reverts to `CALL Service_4`.

```
410  SUB Old_context              !Starts a context
420    ON SIGNAL 4 CALL Service_4 !Define branch for SIGNAL 4
430    CALL New_context           !Call another context
440    OFF SIGNAL 4               !Cancel SIGNAL 4 branch
450    SUBEXIT                    !Exit this context
460  SUBEND
470 !
480  SUB New_context              !Enter new context
490    ON SIGNAL 4 CALL Any_signal !Re-define SIGNAL 4 branch
500    CALL Busy_work             !Do some work
510    SUBEXIT                    !Return to previous context,
520  SUBEND                       !cancelling re-defined branch
```

There is no arbitrary limit on how many levels of ON... statements you can "nest" that specify the same condition. You can have as many levels as you have contexts in your program. A single subprogram can call itself recursively and execute on each entry a different ON... statement for the same condition. Each SUBEXIT restores the previous ON... priority and branch.

Nesting Side Effects

There are side-effects which can occur when you nest ON... statements that specify conditions representing unique system resources.

ON CYCLE, ON DELAYs and ON TIME/TIMEDATE each have only a single timer process per partition. If, for example, you nest ON CYCLE statements, the previous branch and priority are restored on exit from the nested context, but the previous CYCLE time value is not restored.

There is also only one interrupt enable mask per interface. If you nest ENABLE INTR and ON INTR statement pairs, the previous mask is not restored on exit from the nested context. These cases are individually noted in the discussion of each ON... statement, and in the BASIC Language Reference manual entry for each ON... statement.

Branches Deferred Due to Overlapped I/O

Overlapped I/O is discussed in detail in the "Advanced I/O Operations" chapter. A brief introduction is necessary here due to the way in which ON... conditions are handled during overlapped I/O. Most output from your program is not overlapped unless you specifically enable it.

When your program initiates an overlapped I/O operation, your I/O statement actually outputs its data to a block of system memory known as a buffer and your program resumes execution. A system-supplied program, called a process, monitors the transfer of the data from the buffer to the originally specified device or file. The execution of your program and the execution of the system process **overlap** each other.

If an ON... condition, such as an ERROR, occurs as a result of the overlapped I/O, it would be misleading for the system to report it at the next arbitrary SOLC. The ERROR was not caused by that line. ON... conditions caused by overlapped I/O are not reported or logged until the next "access" to the same I/O resource. An **access** is the next statement in your program which explicitly or implicitly specifies or accesses the same I/O resource.

The following example outputs a messages to an HP 2631B HP-IB printer. The message is smaller than the default buffer size (256), so the entire message is copied to the I/O buffer. The program continues execution and runs through the "Other_task" loop. If the printer is off-line or out of paper, the OUTPUT statement cannot output, and causes a TIMEOUT error after five seconds.

If the "Other_task" loop takes longer than five seconds, the TIMEOUT is not reported during the loop. It is reported when the loop completes and line 270 executes. This line attempts to "close" the I/O path name. Because it accesses the I/O path name, the ON TIMEOUT branch occurs during line 270.

```
200  OVERLAP                          !Global overlap mode
210  ASSIGN @Prt TO 406;OVERLAP       !Open I/O path name
220  ON TIMEOUT @Prt,5 CALL On_line   !Set up ON... branch
230  OUTPUT @Prt;Message$[1,128]      !"Launch" output
240  FOR Index=1 TO Large             !\
250    CALL Other_task(Index)         ! >Do other work
260  NEXT Index                       !/
270  ASSIGN @Prt TO *                 !Close I/O path name
 .
 .
 .
510  SUB On_line                      !Called on TIMEOUT
520    BEEP                           !Alert operator
530    DISP LIN(1);"Please put the printer ON-LINE."
540    SUBEXIT                        !Return to line which
550  SUBEND                           !detected error
```

If you need to explicitly check for ON... conditions during overlapped I/O, use the STATUS statement. This statement, discussed in the "Advanced I/O Operations" chapter can access I/O resources without actually performing I/O.

There are two exceptions to deferred ON... condition reporting. They are EOR (End-Of-Record) and EOT (End-Of-Transfer). These conditions exist specifically for the purpose of servicing conditions immediately. See the "Advanced I/O Operations" chapter.

Disabling ON... Branches

You can disable all **asynchronous** ON... conditions two ways.

- You can execute a DISABLE statement. This statement disables all asynchronous ON... conditions in all contexts of your program. You can re-enable them with the ENABLE statement.
- You can increase your system priority with the SYSTEM PRIORITY statement so that it is higher than that specified in the ON... statements. A SYSTEM PRIORITY of 15 disables all asynchronous conditions.

You can also disable any ON...GOTO and ON...GOSUB branch by changing context. You can disable ON...INTR conditions individually with the DISABLE INTR statement. There is no way to disable ON...RECOVER and ON...CALL branches for **error** conditions. You can only **deactivate** these branches. However, to prevent an infinite loop caused by an error in an ON ERROR...GOSUB/CALL service routine, ON ERROR branches are disabled during such routines.

Deactivating ON... Branches

You can deactivate ON... branches explicitly or implicitly. The duration of the deactivation can be temporary or permanent. You can also deactivate a specific ON... condition or a class of ON... conditions.

Explicit Deactivation

The explicit method of deactivating ON... conditions is the OFF... statement. The OFF... statement has two effects:

- it deactivates one or more ON... statements active in the OFF... statement's context;
- it sets the log bit false for the specified active conditions.

OFF... statements specifying a condition identifier deactivate only the identified condition. For example, OFF SIGNAL 4 affects only a SIGNAL 4. In some OFF... statements, the condition identifier is optional, in which case the OFF... deactivates an entire class of ON... statements. For example, OFF SIGNAL affects all SIGNALs.

If an OFF... statement deactivates any ON... statements which were not executed in the same context as the OFF..., those ON... statements are re-activated on exit from that context; however, if the condition occurred during the execution of the subprogram, it was not logged.

It is important to note that an OFF... statement **always** sets the log bit false for the conditions specified. Even if there is no ON... statement active in the OFF's context, the bit is still set false. OFF... statements do not affect semaphores (see the section on WAIT and WAIT FOR...).

Implicit Deactivation

There are circumstances in which the system automatically performs an OFF...; they are:

- closure or re-assignment of an I/O path name, such as, `ASSIGN @Name TO *` (note that re-assignment of a numbered file does not perform an OFF...);
- any statement or command which stops program execution, such as STOP or END.

There are OFF... considerations related to exiting contexts (RETURN..., SUBEXIT, SUBEND). They are:

- exiting a context deactivates ON... statements executed in that context (this is not quite the same as an OFF... because the exit does not set any log bits false);
- exiting a context closes open I/O path names opened in that context, causing an automatic OFF...

Simulating ON... Conditions

When you are writing a program which contains ON... statements it may not be practical to set up circumstances in which every ON... branch is tested. Generally, these are ON... conditions which are caused by some agency outside of your program. For example, if your ON END statement requires detecting the end of a large data file, it is very time consuming to test the service routine by reading through the entire file.

For this reason the BASIC Language System provides CAUSE statements which simulate those ON... conditions which might be difficult or impossible to actually cause. The ON... conditions for which a CAUSE statement is available are: END, EOR, EOT, EVENT, INTR, and TIMEOUT. Since a SIGNAL condition can only be caused by the SIGNAL statement, there is no need for a "CAUSE SIGNAL" statement.

Executing a CAUSE statement (or command) sets the log bit true for the specified ON... condition. If the condition is an EOR, EOT or EVENT, CAUSE also increments the semaphore associated with that condition's WAIT FOR... statement. See the "Waiting for ON... Conditions" section in this chapter.

A CAUSE statement differs from an actual ON... condition in that it never accesses any I/O resource involved. Except for EVENTs, it does not cause the actual condition, it only simulates it. CAUSE EVENT is not actually a simulation, since it is the only statement which can "cause" an EVENT condition.

An example of a CAUSE statement is illustrated in the following program. It normally takes this program at least 40 seconds to read through the file. Try entering the following command while the program is running:

```
CAUSE END @File EXECUTE
```

```
 10  DOUBLE Record,Record_data       !Define variables.
 20  CREATE BDAT "Junk_data",800,4   !Create file full of undefined
 30  ASSIGN @File TO "Junk_data"     !Open file to path name @File
 40  CONTROL @File,7;200,4           !Define EOF as record 200 byte 4
 50  ON END @File GOTO End_of_file   !Trap EOF error.
 60  Record=0                        !Initialize record counter.
 70  !
 80  LOOP                            !Forever (until EOF).....
 90    Record=Record+1               !Increment record count.
100    ENTER @File;Record_data       !Read garbage from file.
110    PRINT "Record#=",Record," Data=",Record_data
120    WAIT .2                       !Pause for 1/5 second.
130  END LOOP                        !Get next record.
140  !
150 End_of_file:OFF END @File        !Cancel trap.
160  ASSIGN @File TO *               !Close file.
170  PRINT "End of file in record#",Record
180  PURGE "Junk_data"               !Purge file.
190  STOP                            !Terminate.
200  END
```

Waiting for ON... Conditions

The concept of the ON... statement implies that your program has something else to do while it waits for one or more ON... conditions to occur. If it hasn't anything else to do, or if it finishes what it was doing before an expected ON... condition occurs, then you need to place it in an idle state in which it can still service the ON... conditions.

In order to branch on an ON... condition, your program must be in the RUN state. It cannot be in the STOP or PAUSE state. There are four ways you can place your program in an idle, or "do nothing" state, yet still be running. They are:

- a wait loop (not recommended);
- a timed wait loop (also not recommended);
- the "generic" WAIT statement;
- a specific WAIT FOR... statement.

Wasteful Waiting - The Wait Loop

Historically, engineering workstations such as desktop computers have been single-user single-program environments. If a program needed to wait for an ON... condition, but had nothing else to do, it could wait for the ON... in an idle loop or wait loop. The following is an example of a **wait loop**.

```
610  ON KEY 7 LABEL "Next Step" GOTO Next_step
620 Spin:GOTO Spin             !Wait for SFK 7
```

The Series 500 is a multi-tasking computer. Even though your program has nothing to do while waiting for someone to press SFK 7, there may be programs running in other partitions which have useful work to perform. The disadvantage of the wait loop is that it requires the computer (CPU) to execute useless instructions for your program when it could be executing useful instructions for some other program. In fact, if your partition priority is high enough, your program's wait loop can dominate all of the CPU's time.

Non-responsive Waiting - the WAIT(time) Statement

The WAIT(time) statement, which specifies a time in seconds, suspends your program for that period of time. The program is still in the RUN state during this time. You could wait for one or more ON... conditions with the following statement pair:

```
610  WAIT .5          !Wait 1/2 second.
620  GOTO 610         !Wait some more, unless condition occurred.
```

The disadvantage of this method is that no ON... branch can occur during the half second wait. The SOLC is performed only at the **beginning** of lines 610 and 620, and not during the `WAIT .5`. You can only respond to ON... conditions every half second.

Waiting for Any ON... Condition - WAIT

To avoid wasting CPU cycles in a wait loop, the BASIC Language System provides other wait statements. One of these statements is the simple, or generic WAIT. It has no argument or parameters and should not be confused with the WAIT(time) statement, which specifies time. The simple WAIT statement suspends the execution of your program until any ON... condition occurs. Your program remains in the RUN state but consumes no CPU time.

The following example shows the WAIT equivalent of the previous example:

```
610  ON KEY 7 LABEL "Next Step" GOTO Next_step
620  WAIT                      !Wait for SFK 7
```

If the WAIT is exited to take an ON...GOSUB or ON...CALL branch, the RETURN or SUBEXIT from the subroutine or subprogram returns to the next line after the WAIT. This is not true of WAIT loops, where execution returns into the loop.

You can also execute live keyboard operations while your program is executing a WAIT statement. This does not cause the program to be released from the wait state, unless an ON KBD branch is enabled or you execute a CAUSE statement.

Note that your program is released from the WAIT by the occurrence of any active ON... condition. The ON... branches need not be enabled, so it is possible for your program to be released from the WAIT and not take an ON... branch.

Don't Miss the Condition

When you use a WAIT statement, consider the possibility that the condition you intend to wait for can occur before you execute the WAIT statement. In this case you already have serviced the condition and can therefore wait indefinitely if it does not occur again.

A suggested way to avoid this situation is have your service routine set a variable which you can test before entering the wait state. For example:

```
310  Sfk_5=0         !Set flag variable false
320  ON KEY 5 LABEL "Disable    Test" CALL Disable_test
  .
  .
  .
410  IF NOT Sfk_5 THEN WAIT                  !Wait if appropriate
  .
  .
  .
510 SUB Disable_test
520     OFF KEY 5                  !Cancel branch
  .
  .
  .
580  Sfk_5=1                                 !Set flag TRUE
590  SUBEXIT                                 !Return
600 SUBEND
```

Waiting for a Specific ON... Condition

The simple WAIT statement has a disadvantage: your program resumes execution on the occurrence of **any** ON... condition. When you need to restrict program response to a single ON... condition, BASIC provides four WAIT FOR... statements which suspend your program until a specific ON... condition occurs. They are: WAIT FOR EOR, WAIT FOR EOT, WAIT FOR EVENT and WAIT FOR INTR.

These statements specify conditions which are usually outside the control of your program: EOR and EOT only occur during TRANSFER, which is usually overlapped; EVENTs can be signalled by programs in other partitions; interrupts (INTR) don't necessarily occur at predictable times.

WAIT FOR EOR, EOT and EVENT also have a special relationship with their associated CAUSE... statements. For each WAIT FOR... condition identifier the system maintains a DOUBLE variable called a **semaphore**. The EOR/EOT semaphores are local to a partition. The semaphore for each EVENT is global to the entire computer.

The WAIT FOR... statements decrements the semaphore by one and then tests to see if the semaphore is negative. The program waits if the semaphore is less than zero. If the semaphore is greater than or equal zero, the program continues. The occurrence of an ON... condition or the execution of a CAUSE... statement not only sets the log bit for the condition, it also increments the semaphore.

Although WAIT FOR... decrements the semaphore, it does not clear log bits. This allows both an ON... and WAIT FOR... specifying the same condition to service the same occurrence. The semaphores cannot be incremented higher than 2 147 483 647. No error occurs; the semaphore simply ceases incrementing.

An EVENT must be created with the CREATE EVENT statement before you can execute an ON, WAIT FOR or CAUSE EVENT statement. You can preset the value of the EVENT semaphore with the CREATE EVENT statement. See the "Partitions and Events" chapter for a more detailed description of events.

The EOR and EOT semaphores are only meaningful during an active TRANSFER statement involving the same I/O path name. The start of a TRANSFER sets the EOR and EOT semaphores to zero.

WAIT FOR INTR does not have a counting semaphore but does have other unique properties which are discussed in the "Advanced I/O Operations" chapter.

Concurrent WAIT FOR... and ON...
If your program has an ON... active but disabled and is waiting at a WAIT FOR... statement, the occurence of an ON... condition releases the WAIT FOR... but does not trigger the ON... branch until the ON... statement is enabled.

If the ON... is active and enabled, the ON... branch is taken immediately. Remember that **enabled** implies that the ON... statement's priority is higher that the current system priority.

Software Initiated Branches - ON SIGNAL

The SIGNAL is the generic ON... condition of the BASIC Language System. Your program detects SIGNALs with the ON SIGNAL statement and causes SIGNALs with the SIGNAL statement. There are 16 SIGNALs, numbered from 0 through 15. The meaning of each SIGNAL depends entirely on your application.

SIGNALs provide a means for different contexts to control one another by triggering each other's ON SIGNAL branches.

A typical use for SIGNAL is to allow a subprogram or function to alert a higher context that a serious problem exists. You use the ON SIGNAL...RECOVER syntax to define the branch. A context can have a normal subprogram return to the line following the CALL or FN invocation and can have an **error** return to another line in the context. Subprograms can return directly to the last context in which a recovery routine exists.

In the following example, the MAIN program defines SIGNAL 0 to mean that the entire program should be re-started. Each subprogram of function context is expected to resolve any problems or errors that arise. If it can't, it "escapes" to the MAIN program by causing SIGNAL 0.

```
 10  ALLOCATE X,Y,Z,Array(200)
 20 !   .
 30  ON SIGNAL 0 RECOVER Try_recover !Define abort path
 40 !   .
 50  CALL First_level          !Call a new context
 60 !   .
 70  STOP                      !Normal completion
 80 !   .
 90 Try_recover:               !Someone SIGNALed 0   !
100  DEALLOCATE X,Y,Z,Array(*)  !Release resources,
110  PRINT "RECOVER"
120  GOTO 10                   !and re-start
130 !   .
140  END
150 !   .
160 !   .
170  SUB First_level           !A sub context
180    CALL Second_level       !Call yet another
190 !   .
200    SUBEXIT                 !Return to MAIN
210  SUBEND
220 !   .
230 !   .
240  SUB Second_level          !Another context
250    ON ERROR GOTO Second_err!Trap local errors.
260 !   .
270    SUBEXIT
280 !   .
290 Second_err:                !Error, try to fix problem
300 !   .
310    IF NOT Fixed THEN SIGNAL 0                !Couldn't fix, abort.
320 !   .
330  SUBEND
```

Keystroke Initiated Branches - ON KEY

Traditional interactive programs offer you choices in the form of lists or menus during an input statement. You select the option and typically enter its number. Not only is this a slow process, but you can only redirect the program flow during a keyboard input statement.

BASIC has much more powerful ways of allowing you to alter program flow from the keyboard. Your computer's keyboard has 32 Special Function Keys, also referred to as SFKs. These keys are labelled (0 16) through (15 31). You execute SFK 0 thru SFK 15 simply by pressing the keys (0 16) through (15 31). You access SFK 16 through SFK 31 by pressing (SHIFT) and the key. If your CRT has softkeys, those keys duplicate, from left to right, SFKs 0 through 7.

These keys have two purposes in the BASIC Language System: typing aids and program SFK's. The "Computer Operating Features" chapter discussed the use of these keys as typing aids while editing or during a programmed keyboard input statement. While a program is running, and not awaiting keyboard input, each SFK can have a different function in your program. This section discusses the use of these keys in a running program.

Each SFK can represent a different program branch which is to occur when that key is pressed. For each key you want to define, you must execute a separate ON KEY statement. For example:

```
410   ON KEY 1 GOTO Exit_program          !End program on SFK 1
```

In addition to specifying the key number (0 thru 31), priority and branch, the ON KEY statement can specify a LABEL for keys 0 thru 7 which appears on the CRT screen. For example:

```
50   ON KEY 0 LABEL "ABORT     PROGRAM",15 RECOVER Abort
```

The preceding statement causes the string

```
ABORT
PROGRAM
```

to appear in the KEY LABELS SCREEN. The KEY LABELS SCREEN defaults to the two bottom lines of the CRT. A LABEL is a string expression of any length. The label area of the SCREEN is eight fields, each ten columns wide and 2 lines high. The label is left-justified and wraps around every ten columns. Excess characters are not displayed. Labels that are longer than the displayed area are left-justified. The excess portion of the label is not displayed. The fields for SFK 0 through SFK 7 begin in columns 1, 11, 21, 31, 41, 51, 61 and 71. Although not displayed, you can specify labels for SFKs 8 through 31.

You can use the CREATE SCREEN and ASSIGN...TO SCREEN statements to change the number of lines in the KEY LABELS SCREEN. If you reduce the width of this SCREEN to less than 80 columns, a field is only displayed if its entire ten columns are available. If the SCREEN does not begin in CRT column one, the fields do not align with the CRT softkeys.

An ON KEY statement label remains displayed as long as the statement is active. The label is not removed from the screen while the statement is disabled. It is only removed when the statement is deactivated, usually by an OFF KEY statement. All ON KEY statements are temporarily deactivated during PAUSE, EDIT...$, ENTER KBD, ENTER KBD USING, LINPUT and INPUT. During these statements the keys assume their typing aid definitions. During ON KBD, the SFKs are active. During ON KBD ALL, the SFKs are deactivated and return their keycodes in the KBD$ buffer.

If you execute two ON KEY statements in different contexts specifying different labels, each of the ON KEY label is displayed only in the context in which it is active.

You can display the key labels in inverse video, blinking, underline and color. You can do this in two ways:

- concatenate the CHR$ function to a LABEL string expression, using extension control character codes in the range 128 through 143;
- in a LABEL literal, press CTRL and one or more of the appropriately marked SFKs. This is not recommended since the blinking and color effects are visible only in the displayed listing of programs and not in printed listings.

You cannot turn on display enhancements in a LABEL by using escape sequences. DISPLAY FUNCTIONS mode is permanently enabled for the KEY LABELS SCREEN. Any escape sequence is displayed as characters, including the escape character.

If you press a key for which no ON KEY statement is active, and which has no typing aid function defined, the message UNDEFINED KEY appears in the MESSAGES SCREEN.

The following program shows some of the uses for ON KEY.

```
 10  ON KEY 0 LABEL "Show      Time     ",11 CALL Showtime
 20  ON KEY 1 LABEL "OFF KEYS  0,2 & 3 ",11 CALL Off_023
 30  ON KEY 2 LABEL "CALL a    subprog.",14 CALL Subprogram
 40  ON KEY 3 LABEL "GOSUB a   subrout.",10 GOSUB Subroutine
 50  ON KEY 7 LABEL "Terminate Program ",15 RECOVER End
 60  PRINT PAGE
 70 Wait:PRINT "Waiting in MAIN program"
 80  WAIT
 90  GOTO Wait
100 !
110 End:PRINT "ON KEY sample ended"
120  STOP
130  !
140 Subroutine:!
150  PRINT "In a GOSUB subroutine for 200000 loops"
160  FOR Busy=1 TO 200000
170  NEXT Busy
180  PRINT "Exiting subroutine"
190  RETURN
200  END
210 !
220  SUB Showtime
230    PRINT TIME$(TIMEDATE)
240    SUBEXIT
250  SUBEND
260 !
270  SUB Off_023
280    OFF KEY 0
290    OFF KEY 2
300    OFF KEY 3
310    PRINT "Waiting 3 seconds"
320    WAIT 3
330    SUBEXIT
340  SUBEND
350 !
360  SUB Subprogram
370    PRINT "In a subprogram for 200000 loops"
380    FOR Index=1 TO 200000
390    NEXT Index
400    PRINT "Exiting subprogram"
410    SUBEXIT
420  SUBEND
```

Clock Initiated Branches

If you need to have a portion of your program (a task) begin execution at a specific time or after a specific delay, you can use the WAIT(time) statement, which suspends your program for the specified period of time. However, your program cannot perform any other useful work while suspended.

If you want to schedule a branch to occur at a specified time, at a time and date or after an elapsed time, you can use the chronological conditions ON TIME, ON TIMEDATE, ON DELAY and ON CYCLE.

When you execute any of these statements, they create a **timer process** which executes in parallel with your program. This process is often referred to as a watch-dog process. Each timer process monitors the real-time clock and triggers an ON... condition when the specified time arrives (or elapses). Each program can have at most three of these processes, one each for ON CYCLE and ON DELAY, and one for either ON TIME or ON TIMEDATE.

In each of these statements, the time value you specify is converted to a REAL Julian timedate. The ON... condition occurs when the value of the real time clock first equals or exceeds this Julian timedate value. If you change the clock with SET TIMEDATE while any of these ON... statements are active, their branches may occur immediately or never.

A timer process only checks the real-time on every "tick" of the clock. The real-time clock defaults to a "tick" every 10 milliseconds. This determines the precision with which you can establish when the ON... condition occurs. Changing the "tick" interval is described in the "Non-Volatile Memory" chapter.

Each timer process maintains a single value for the time/date at which the branch should occur. If you nest ON... statements referencing the same timer process, the original branch is restored upon exiting the subordinate context, but the time/date value remains as most recently specified.

In the following example, subprogram "Administration" defines an ON... branch to occur at 4:00 PM (16:00) to blow the factory whistle. However, it calls subprogram "Facilities" to perform some tasks. "Facilities" defines a new branch to occur at 3:30 PM (15:30) to measure effluent. When "Facilities" returns to "Administration", the branch to "Factory_whistle" is restored, but it now occurs at 3:30 instead of 4:00.

```
410  SUB Administration
420    ON TIME (TIME("16:00")) CALL Factory_whistle
430    FOR Task=1 TO 123
440      CALL Security_check(Task)
450      CALL Facilities
460    NEXT Task            !Factory_whistle called at 15:30 now
470    SUBEXIT
480  SUBEND
490  !
500  SUB Facilities
510    ON TIME (TIME("15:30")) CALL Smokestack
520    CALL Effluent
530    CALL Recycle
540    SUBEXIT
550  SUBEND
```

Branching at an Absolute Time - ON TIME/TIMEDATE

ON TIME and ON TIMEDATE define a branch which can occur at (or after) the specified time. For ON TIME, you specify a time of day in seconds since midnight. For ON TIMEDATE, you specify a timedate. The form of a timedate is a numeric expression, which, when converted to a REAL value, is the number of seconds since the base of the Julian calendar.

If you only have the time in hours, minutes, seconds and fraction form, convert it to a string and use the TIME function to obtain the number of seconds since midnight `TIME("12:45:00")`. If you only have the date in day, month and year form, convert it to a string, and use the DATE function to obtain the Julian day `DATE("24 JAN 84")`. Add the Julian date to the seconds and you have a Julian timedate `TIME("12:45:00")+DATE("24 JAN 84")`

ON TIME and ON TIMEDATE are effectively the same statement. They share the same watch-dog timer process. If you specify both statements in the same context, the second replaces the first. Likewise, OFF TIME statement cancels both ON TIME and ON TIMEDATE statements. Although ON TIMEDATE specifies a future Julian timedate and ON TIME specifies a time (in seconds) within the next 24 hours, the ON TIME value is always converted to a TIMEDATE for the time process.

For example, the statement:

```
420   ON TIME Time CALL Backup
```

is identical in effect to:

```
380   T=TIMEDATE                        !Get current time & date
390   Date=DATE(DATE$(T))               !Extract date portion
400   Hour=TIME(TIME$(T))               !Extract time portion
410   Tomorrow=86400*(Time<Hour)        !0 or 86400 as needed
420   ON TIMEDATE Date+Tomorrow+Time CALL Backup
```

ON TIME specifies a time within the next 24 hours. If you specify in ON TIME a time value which has already passed in the current day, the ON... branch occurs at that time on the next day.

Branching on an Elapsed Time - DELAY & CYCLE

If you want a branch to occur in ten minutes, you could request the current time with the TIME(TIME$(TIMEDATE)) functions and establish an ON TIME branch to occur at TIME(TIME$(TIMEDATE)) + (60*10). It is more convenient to use ON DELAY. The ON DELAY condition occurs after a time interval specified in seconds.

A typical use for the ON DELAY statement is to set a "timeout" for ON... conditions. Suppose at some point there is no useful work for your program to do, you expect one or more ON... conditions to occur in the near future, but you do not want to wait more than one minute for an ON... to occur. Use the following technique.

```
1130   ON DELAY 60 GOTO Times_up   !Give up after 1 minute
1140   WAIT                        !Wait for any ON...
```

In the previous example, the ON DELAY statement sets up a "software timeout". If no other ON... condition occurs within one minute, the ON DELAY branch does, insuring that your program does not wait indefinitely. Note that any other ON... service routine should execute a DISABLE or OFF DELAY if it must not be interrupted by the ON DELAY branch.

You cannot use ON DELAY to "timeout" I/O operations, such as an OUTPUT statement. ON DELAY is only detected at SOLC. Even though an OUTPUT statement may be waiting after the DELAY period has expired, the branch cannot be taken until the OUTPUT statement completes. To timeout I/O statements, use the ON TIMEOUT statement.

When You Need Two Timer Processes

ON DELAY and ON TIME/TIMEDATE merely provide you with two methods of specifying a time in the future. However, they represent two independent timer processes. If you need to establish two ON... branches for different times in the future, you can specify one as a DELAY and the other as a TIME or TIMEDATE.

Branching Repeatedly on an Elapsed Time - CYCLE

If your program needs to execute a branch regularly, you could execute a new ON TIME with an incremented time value each time an ON TIME branch occurred. It is more convenient to use ON CYCLE. The ON CYCLE statement specifies a time interval in seconds. While the ON CYCLE statement is enabled, one ON CYCLE branch occurs per interval indefinitely.

A typical use for ON CYCLE is to take an instrument reading at regular intervals. The "ON...GOSUB" section of this chapter has an example showing such a use. Here is another use for ON CYCLE.

If you have a time-consuming (compute-bound) program, you might want to know which portions of the program are using the most CPU time. The following program forces an ERROR every CYCLE, capturing the line number of the line just executed. The ERROR causes a CALL to subroutine "Histogram" which obtains the number of the line whose SOLC detected the CYCLE ERROR. "Histogram" increments an array element for that line, recording the relative amounts of time your program spent in each line.

```
    1   COM Line(5:9999)                    !Histogram array
    2   MAT Line=(0)                        !Clear array if re-RUN
    3   ON ERROR CALL Histogram             !Call Histogrammer
    4   ON CYCLE .1 CALL Nonexistent        !every CYCLE attempt
        .
        . Your program code goes here
        .
10001   OFF CYCLE                           !Sample code to
10002   OFF ERROR                           !print out the
10003   PRINT                               !histogram
10004   FOR Line_no=5 TO 9999               !<<change to prog size<<
10005     PRINT Line_no,Line(Line_no)
10006   NEXT Line_no
10007   END
10008   !
10009   SUB Histogram
10010     COM Line(5:9999)                  !Histogram array
10011     DIM Message$[80]                  !For reading ERRM$
10012   !
10013     Message$=ERRM$                    !Get error message
10014     Start=POS(Message$," IN ")+4      !Find line number
10015     Stop=POS(Message$,":")-1          !Find colon after it
10016     IF Stop=-1 THEN Stop=LEN(Message$) !If none, use length
10017     Solc=VAL(Message$[Start,Stop])    !Get SOLC line
10018     Line(Solc-1)=Line(Solc-1)+1       !Incr. count for previous
10019     SUBEXIT                           !Resume
10020   SUBEND
10021   !
```

This routine does not accurately account for program lines which take longer than .1 seconds to execute. You must increase the cycle time in such cases.

Error Initiated Branches

An error is any condition which would normally cause your program to PAUSE and produce an error message. An active ON TIMEOUT, ON END or ON ERROR causes the program to branch upon the occurance of an error rather than PAUSE and produce the error message.

ON error conditions do not wait until the statement finishes execution before taking the branch. The statement cannot finish because it encoutered the error. The branch is taken immediately.

ON error conditions have a priority of 16, the highest system priority. The priority is reduced to 15 within the service routine to allow error conditions to interrupt each other.

DISABLE does not affect ON error conditions.

ON ERROR

ON ERROR is triggered by any error conditions unless an ON END or ON TIMEOUT is active. ERROR 59 and 60 are trapped by an active ON END. ERROR 168 is trapped by an active ON TIMEOUT. ON ERROR traps all other errors.

If an execution error occurs while servicing an ON ERROR CALL or ON ERROR GOSUB, program execution PAUSEs.

In the following program, the ON ERROR traps any error except an end of file on file #1. Without line 30, ON ERROR would trap ERROR 59 also.

```
 10  ON ERROR GOSUB Fixit
 20  ASSIGN #1 TO "TEST"
 30  ON END #1 GOTO Lasts
 40  FOR I=1 TO 100
 50    READ #1;A
 60  NEXT I
 70  STOP
 80 Fixit:  PRINT "ERROR OCCURRED";ERRN
 90         STOP
100 Lasts:  PRINT "END OF FILE"
110 END
```

After an ON ERROR GOSUB or ON ERROR CALL routine has executed, the program returns to the beginning of the statement which caused the error. It does not return to the next statement. The subroutine or subprogram is expected to correct the error. If it does not, the error can occur again, and your program may loop indefinitely.

The subroutine Fixit in the following program attempts to correct errors. If it cannot, it displays the error number and line number which caused the error and then pauses until the user corrects the error.

```
 10  ON ERROR GOSUB Fixit
 20  INPUT "NEW FILE NAME",File$
 30  CREATE DATA File$,100
 40  ASSIGN #1 TO File$
     .
     .
     .
500 Fixit:  SELECT ERRN
510         CASE 54      !DUPLICATE FILE NAME
520            INPUT "FILE NAME ALREADY EXISTS. PURGE(Y/N)?",P$
530            IF P$="Y" THEN
540               PURGE File$
550            ELSE
560               PRINT "FILE ALREAD EXISTS.  PROGRAM CANNOT CONTINUE"
570               STOP
580            END IF
590         CASE 53      !INVALID FILE NAME
600            INPUT "FILE NAME IS INVALID.  INPUT ANOTHER",File$
610         CASE 80      !NO DISC, DISC DOOR OPEN
620            PRINT "CHECK THE DISC.  PRESS CONT TO CONTINUE"
630            PAUSE
640         CASE 52      !INVALID MSUS
650            INPUT "RE-INPUT THE FILE. YOU MSUS IS INVALID",File$
660         CASE ELSE    !ANY OTHER ERROR
670            PRINT "AN ERROR HAS OCCURRED. CORRECT THE ERROR"
680            PRINT " AND PRESS CONT."
690            PRINT ERRM$
700            PAUSE
710         END SELECT
720         RETURN
730 END
```

There are four functions which return error information. You can use these functions within the error handling routine to determine the error which occurred.

Function	Return information
ERRDS	returns device selector of the device or interface on which the most recent I/O error occurred.
ERRL	returns a boolean value which indicates if the last error encountered occurred within the specified line. This function is the only method with which a compiled program can determine the line in which an error occurred.
ERRM$	returns a string which indicates the error number and location of the last program error, either the line number or subprogram name. If the ERRORS_ENG option is loaded, it returns the error message.
ERRN	returns the number of the most recent program execution error.

ON END

ON END traps end-of-file and end-of-record errors, ERROR 59 and 60. An END condition is caused by any of the following:

- an attempt to read or write beyond the end of a file record in random mode;
- an attempt to read beyond the end of a file, or write more than one record beyond the end of a serial file;
- attempting to move either the empty or fill pointers past each other in a BUFFER;
- the CAUSE END statement.

For information about random and serial file access, refer to the "File Access and Data Transfer" chapter.

The ON END statement refers to **one** file number or I/O path name. You need one ON END for each file number or I/O path name.

OFF END specifies either a single file number, all file numbers or a single I/O path name.

In the following program, two files are opened with the ASSIGN # statement. ON END #2 is used to move the pointer to the last record, and then to indicate when the file is full. ON END #1 is used to tell the user when an invalid record number is input.

```
 10  INPUT "SOURCE FILE",S_file$
 20  INPUT "DESTINATION FILE",D_file$
 30  ASSIGN #1 TO S_file$
 40  ASSIGN #2 TO D_file$
 50  ON END #1 GOSUB Past_end
 60  ON END #2 GOTO No_more_room
 70  INPUT "WHICH RECORD TO TRANSFER?",Record
 80  READ #1;Data$
 90  DISP Data$
100  INPUT "TRANSFER THIS RECORD",T$
110  IF T$="Y" THEN PRINT #2;Data$
120  !
130  ! Other code
140  !
400 Past_end:  PRINT "RECORD ";Record;" IS PAST THE END OF FILE"
410            INPUT "NEW RECORD NUMBER",Record
420            RETURN
421            !
430 No_more_room:!
440            PRINT "THE FILE ";D_file$;" IS FULL"
450            GOTO End
460 !
470 !
```

ON TIMEOUT

ERROR 168, device timeout, is trapped by an ON TIMEOUT. There is no system default timeout. If no ON TIMEOUT is active for a specific I/O path name or interface select code, an I/O statement may wait indefinitely.

Timeouts apply only to external devices and not to the internal CRT, KBD, PRT, SCREEN, graphics, tone generator or mass storage.

The ON TIMEOUT statement specifies both a time and an interface select code or I/O path name. If an I/O statement takes longer than the specified time to complete, the branch is taken. If the ON TIMEOUT statement specifies an interface select code, the specified timeout interval defines the default timeout for any I/O path names assigned to that interface select code.

```
200  ASSIGN #1 TO "LIST_FILE"
210  ON END #1 GOTO Last
220  ASSIGN @Prt TO 406;OVERLAP
230  ON TIMEOUT @Prt,5 CALL On_line
240  FOR Record=1 TO 99999
250    READ #1;Message$
260    OUTPUT @Prt;Message$
270  NEXT Record
272 Last:ASSIGN #1 TO *
280  ASSIGN @Prt TO *
290  END
300  SUB On_line
310    PRINT "THERE IS A PROBLEM WITH THE PRINTER"
320    PRINT "FIX IT AND PRESS CONT"
330    PAUSE
340    SUBEXIT
350  SUBEND
```

Chapter 8
Partitions and Events

The memory of the computer can be divided into sections called partitions. Each partition can run a program and interact with the user. Each partition can access the full capabilities of the computer, all of the computer's resources and peripherals. The resources are shared by the partitions. A partition uses the keyboard and private screens on the CRT when it is the foreground partition. Any partition can use the other resources of the computer such as files, printers, and other I/O devices. Partitions are, in effect, virtual computers.

When you power up the computer, the system creates one partition, INITIAL_PART. This is the partition which you have been using for your programming.

Using multiple partitions has several advantages:

- Several programs can run simultaneously.
- You can do program development in one partition while programs are running in other partitions.
- Multiple CPU's are more efficiently used with multiple partitions.

The following table summarizes the statements discussed in this chapter.

Partition Statements

To perform this operation:	Use this statement:
Create a partition.	CREATE PARTITION
Delete a partition.	DEL PARTITION
Move a partition to the foreground.	ATTACH, (SHIFT)(STEP) ((NEXT PART))
Determine if the partition is in the foreground.	SYSTEM$("FOREGROUND")
Delete the contents of a partition.	SCRATCH P
List the partitions and their contents.	PARTITION STATUS
Run programs in ALL partitions.	(SHIFT)(RUN)
Stop programs in ALL partitions.	(SHIFT)(STOP)
Pause programs in ALL partitions.	(SHIFT)(PAUSE)
Continue programs in ALL partitions.	(SHIFT)(CONTINUE)

Screen Statements

To perform this operation:	Use this statement:
Create a screen.	CREATE SCREEN
Delete a screen.	DEL SCREEN
Return the device selector of a screen.	SCREEN(n)
Move a screen.	MOVE SCREEN
Reset all screens to their default definitions.	RESET SCREEN

Events

To perform this operation:	Use this statement:
Create an event.	CREATE EVENT
Delete an event.	DEL EVENT
Increment the event semaphore and cause the branch to be taken on any enabled ON EVENT.	CAUSE EVENT
Decrement the event semaphore, and wait for the event if the semaphore is negative.	WAIT FOR EVENT
Return the current value of the event semaphore.	EVENT LEVEL
Define a branch to be taken when a CAUSE EVENT occurs.	ON EVENT
Deactivate the ON EVENT branch.	OFF EVENT

Creating a Partition

In the CREATE PARTITION statement, you can specify several characteristics of the partition being created, such as its size and priority. (These are shown in the BASIC Language Reference manual.) If you do not specify values for these characteristics, default values are used. The following example, uses the default values.

Execute the following command:

```
    CREATE PARTITION "CLOCK"
```

A partition named CLOCK has now been created.

To get access to the partition, press the (SHIFT) and (STEP) keys simultaneously. This is the (NEXT PART) key. Now, you are in the CLOCK partition.

You could have named the partition anything you wanted (up to 16 characters in length). The following program shows you why that name was chosen.

Using the Editor, enter the following program:

```
    10    CREATE SCREEN 60,1,40,1,40
    20    PRINTER IS SCREEN(60)
    30    PRINT DATE$(TIMEDATE),TIME$(TIMEDATE)
    40    WAIT 1
    50    GOTO 30
    60    STOP
    70    END
```

Press (RUN).

In the upper right corner of the CRT the date and time is displayed. The time is updated each second.

Press (NEXT PART) again.

You are now in the first partition, INITIAL_PART. It was created when the computer was powered up or the last time a SCRATCH A was done. The clock program is still running in the CLOCK partition. In INITIAL_PART, you can run another program, use the Editor, or execute any command without interferring with the CLOCK partition.

Foreground and Background

The partition with access to the CRT and keyboard is the **foreground** partition. There is only one foreground partition at any time. All other partitions are **background** partitions. The number of background partitions is limited only by the amount of memory in the computer.

The NEXT PART key makes the current foreground partition a background partition and the next background partition becomes the foreground partition. The **next partition** to become the foreground partition is shown in the partition status output as the partition listed after the current foreground partition. If the current foreground partition is the last partition in the list, the next partition is the first one in the list. Partitions are placed in the list in the reverse of the order that they were created.

For example, suppose you created three partitions in the following order: PART1, PART2, PART3. The list of partitions is:

```
PART3          ←——next partition
PART2
PART1
INITIAL_PART   ←——foreground partition
```

When you press NEXT PART the list changes to:

```
PART3          ←——foreground partition
PART2          ←——next partition
PART1
INITIAL_PART   ←——background partition
```

The ATTACH statement also moves a partition into the foreground. The partition named in the statement moves to the foreground, and the old foreground partition moves into the background.

```
ATTACH "CLOCK"
```

Note that there is no actual "moving" of partitions from one area of memory to another. There is an internal indicator which tells the computer which partition is the foreground partition and which partitions are background partitions.

For example, partition INITIAL_PART is currently the foreground partition. Partition CLOCK is currently a background partition. The statement `ATTACH "CLOCK"` moves CLOCK to the foreground and INITIAL_PART to the background.

Attaching the partition to the foreground associates the keyboard and CRT to that partition. It does not affect the **execution state** of any partition.

Partition Status

The PARTITION STATUS statement returns the status of every partition and program in the system.

For example, execute: `PARTITION STATUS;ALL`

```
PARTITION NAME   FGND  STATE            PRIORITY  TYPE
===============  ====  ===============  ========  =============
CLOCK                  WAIT                    3  BASIC
INITIAL_PART       *   COMMAND                 3  BASIC
```

FGND is the abbreviation for foreground. The * tells which partition is the foreground partition. The program can also find out if it is a foreground or background partition with the SYSTEM$ function. SYSTEM$("FOREGROUND") returns a 1 if the program is in the foreground partition and returns a 0 if the program is in a background partition.

STATE refers to the state of the program in the partition. That is, whether the program is stopped, waiting, paused, or running, or whether a command was just executed in the partition.

Creating, Deleting and Clearing Partitions

In creating the CLOCK partition, you used the CREATE PARTITION statement. The discussion of CREATE PARTITION in the BASIC Language Reference Manual describes many optional parameters which can be used. These parameters define the size, type and priority of the partition. A command can also be included in the statement which is executed as soon as the partition is created. A return variable can be included to return an error number if an error occurs when the statement is executed. For example,

```
CREATE PARTITION "PART1";EXECUTE "LOAD ""P1"",1", RETURN Err
```

Deleting Partitions

When you no longer need a partition, memory can be reclaimed by deleting that partition. The program in that partition can execute the DEL PARTITION statement as its last statement, (last because when the partition is deleted, the program is also deleted).

A program in another partition or you can also delete the partition. The partition being deleted must be a background partition, and the program in it must be STOPped.

The SCRATCH A command deletes all partitions. After SCRATCH A, INITIAL_PART is created again. All programs must be STOPped before a SCRATCH A is executed. (SHIFT) (STOP) stops all programs.

Clearing Partitions

If you want to reset all the defaults in a partition, including deleting any program in the partition, use the SCRATCH P command. This command is like SCRATCH A except that it is local to the partition.

SCRATCH A deletes all programs and all partitions, resets all devices, variables and modes (see the Master Reset Table in the Useful Tables section of the BASIC Language Reference manual). SCRATCH P deletes the program in the foreground partition and resets the variables and modes in that partition.

Imbedded Commands

A major reason for creating many partitions is to enable several programs to run simultaneously. The LOAD command can be included in the CREATE PARTITION statement so you don't have to manually load and run the programs.

For example, the autostart program, AUTOST, could create four partitions and load and run programs in each of them using only a few statements.

```
10 CREATE PARTITION "MACHINE1"; EXECUTE "LOAD""MACH"",1"
20 CREATE PARTITION "MACHINE2"; EXECUTE "LOAD""MACH2"",1"
30 CREATE PARTITION "MACHINE3"; EXECUTE "LOAD""MACH3"",1"
40 CREATE PARTITION "STOPPAGE";INTR PRIORITY 2, PRIVATE
   MEMORY 65536,EXECUTE "LOAD""TROUBLE"",1"
50 ATTACH "STOPPAGE"
60 DEL PARTITION
70 END
```

In the previous example, lines 10, 20 and 30 create partitions, and load and run programs in each of them. The LOAD command is executed in the new partition as if it were typed in from the keyboard. The creating partition does not get an error if the LOAD fails, since the LOAD is executed in the new partition. If an error occurs the error message is displayed on that partitions message screen.

Line 40 creates a partition, allocates private memory to it and loads and runs a program. Private memory and interrupt priority are described later.

Line 50 attaches the partition STOPPAGE to the foreground, thus moving the partition in which this program is running to the background. Since this program is not needed after it completes, it is deleted in line 60, however, line 60 can only be executed from the background. Thus, the reason for line 50.

Private Memory Partition

You can guarantee that a partition always has sufficient memory in which to run by specifying PRIVATE MEMORY in the CREATE PARTITION statement. The system allocates the amount of memory specified to the private memory partition so that other partitions cannot interfere with memory availability within the private memory partition. An error occurs if a private memory partition attempts to use more memory than was allocated to it in the CREATE PARTITION statement.

You should include sufficient memory in the memory size of a private memory partition for the partition to be created and initialized . The minimum memory size varies depending on what option binaries are present in the system.

Partition Priority

Each partition has a process priority associated with it that determines when it can use the CPU. This should not be confused with the system priority, which determines whether a particular ON condition is currently enabled. The partition priority can be in one of two ranges. The first is the "normal" priority range, from 1 to 5, with 5 being the highest priority. The second range is the interrupt priority range, 1 or 2, with 2 being the higher. The interrupt priority is always higher than the "normal" priority.

The priority of a partition determines when it has access to the CPU and thus when it can execute statements. For the following discussion, assume that there is only one CPU. When there are two or three CPU's, the algorithm follows a similar pattern.

The highest priority partition always executes exclusive of any lower priority partitions until it suspends execution. Execution may be suspended in many different ways. The simplest way is to change to the WAIT, PAUSE, STOP, or INPUT state via the associated statements or to wait on an I/O device.

For example, a higher priority partition can allow a lower priority partition to execute by executing a WAIT n, where n is the number of seconds the lower priority partition is to execute. Another way to suspend the execution of a partition is to have the operation system "wait" for an external event, such as an I/O transaction. In this case, the operating system suspends the higher priority partition (waiting for an interrupt) and allows the lower priority partition to execute until the interrupt occurs.

For example, a higher priority partition printing a program listing allows a lower priority partition to execute at the same time the listing is being produced. This happens even with only one CPU, because the printer is a slower device than the CPU.

If there are two or more partitions with the same priority (and there is not a higher priority partition), they share the CPU in a round robin time-sharing manner. The default for this time-share interval is 10 milliseconds; every 10 milliseconds the operating system allows the "next" partition to execute. See the Non-Volatile Memory section of the "Getting to Know Your System" chapter for changing this interval.

Note

Use of partition priorities may lock out low priority partitions, lock out the keyboard or cause deadlocks. Use partition priorities with care.

A word of caution concerning the use of partition priorities is in order here. Consider the case where a low priority partition (A) is in the foreground, and a high priority partition (B) is in the background. The foreground cannot execute anything as long as the background is using the CPU. Assume the following events occur.

1. The background releases the CPU for a short period of time, say to wait for an interrupt, and the foreground begins executing a command.
2. In the middle of the command that the foreground is executing, the background partition resumes execution.

The state of the foreground is now highly dependent on the nature of the statement or command that was being executed. If that statement was PARTITION STATUS, it, like many others, requires that the system access a semaphored internal data structure. If the foreground **lost** the CPU while that data structure was still locked to that process, then that data structure remains locked.

3. The background partition attempts to access that same data structure with another PARTITION STATUS statement.

Partition B, the background partition with the higher priority, now waits for A to release or "UP" the semaphore protecting the system data structure. However, A is a lower priority partition and is waiting for B to release the CPU.

Normally, A now executes enough instructions to free the lock for which B is waiting, then B resumes execution.

Note that if there is a third partition with a higher priority than A, it will execute, leaving A and B quiescent.

To avoid this and similar problems, the following situations should be avoided.

- Avoid using a "10 GOTO 10" construct. While this has been a common technique for "waiting" for an event such as a key press, the BASIC Language System provides a better way to do this with the WAIT statement. Failure to follow this suggestion can lead to a partition that will not release the CPU to anyone.
- If a partition has a higher priority than the foreground partition, it should relinquish the CPU periodically using the statements mentioned earlier to allow the foreground to perform certain tasks.
- Order resource allocations and deallocations in such a way as to avoid deadlock. An example of this is given later in the "Deadlock" section.
- Use interrupt priority partitions only when they are in the foreground when in a single CPU system.

Interrupt Partitions

The interrupt priority partition provides the user with a greater degree of predictability in processing interrupts. Basically, the user process in an INTR priority partition executes at a higher priority than the operating system processes, such as the keyboard service process. This means that while an interrupt priority partition is executing, no other process can be dispached. For example, if an INTR partition was performing some calculation, keystrokes would be ignored, since the keyboard process cannot execute as long as the INTR partition has the CPU. Note that this implies that if an INTRs partition ever gets into an infinite loop, the computer would have to be turned off, since the STOP key would be ignored!

INTR partitions should only be used when an application **must** process an interrupt as fast as possible. In this application they can be indispensible, since otherwise extraneous external events can ruin interrupt response time.

Input and Output

A program can get input from any device other than the keyboard whether it is the foreground or a background partition. A program can send output to any device whether it is the foreground or background partition.

Output to the Internal CRT

Output sent to the CRT is displayed if the program sending the output is the foreground partition. It is also displayed if the output is sent to a **public screen** on the CRT. When a partition becomes the foreground partition, output previously sent to that partition's **private screens** is shown.

A program creates a screen with the CREATE SCREEN statement.

```
CREATE SCREEN Screen_no,Start_row,Start_col,Height,Width
```

Public screens are those screens defined with a screen number from 51 to 99. A public screen created in one partition, can be used by any partition. Public screens are displayed on the CRT regardless of which partition is the foreground partition. Private screens have a screen number from 1 to 50. Only the partition which created the screen has access to it and private screens are only displayed on the CRT when the partition which defined them is the foreground partition.

Default Screens

There are six screens created at power-up, after SCRATCH A or after a RESET SCREENS statement. Each screen is used for different functions: printing, displays, keyboard input, results and messages, the run light and key labels. The organization of these screens is described in the "Getting to Know Your System" chapter.

The RESET SCREENS statement deletes all screens and then creates the six default screens. The default screens are private screens. Each partition has these six screens.

```
RESET SCREENS
```

Output to a Screen

To direct output to a particular screen, you use the SCREEN function. For example,

```
PRINTER IS SCREEN(10)
OUTPUT SCREEN (60)
```

SCREEN(n) can be the device selector in any I/O statement, such as, ENTER, OUTPUT, STATUS, CONTROL, IOSTATE, RESET and ABORTIO.

Input from the Keyboard

A program can receive input from the keyboard only if it is in the foreground partition.

There is one KBD$ buffer for each partition. (The "Introduction to Input" chapter describes this buffer.) A keystroke is stored in the foreground partition's KBD$ buffer. That partition can be switched to a background partition, and the next keystrokes are stored in the new foreground partition's KBD$ buffer. A partition can read its KBD$ buffer at any time.

Keyboard Commands

If you change the default mass storage device, using the command MSI, it is changed for the foreground partition only: the background partitions are not affected. Commands such as MSI, PRINTER IS, PRINT ALL IS, GRAPHICS ON/OFF, ALPHA ON/OFF and GRADS/RADS/DEG similarly affect only the foreground partition.

Commands which effect the execution state of the partitions may effect all partitions. If you press STOP or RUN, only the program in the foreground partition is affected. STOP, RUN, PAUSE, and CONTINUE when shifted, stop, run, pause or continue programs running in all partitions. SCRATCH A effects all partitions. DEL PARTITION effects the partition specified.

Output to Shared Devices

When sending output to a shared device, the program should gain exclusive access to that device. If it doesn't, output from several programs could merge. The device could be a printer, file, or other output device. A program can gain exclusive access to any device using EVENTS.

Events

An event, or event semaphore, is a signalling device used by cooperating programs in two or more partitions. A program schedules itself by querying the value of an event. A value less than one indicates the number of programs that are already waiting for the event. A positive value indicates the number of programs that can request the event without having to wait.

An event is a system variable which you can increment, decrement and query the value of. It is global to the computer: an event created in one partition is accessable by any other partition.

Why Use Events

You use events to synchronize processes in partitions, to share resources or cause a partition to do something. For example, if several partitions want to use a printer, create an event that all partitions agree to query before they use the printer. If the event has a value of 1, the printer is not being used. If the event has a value of 0, the printer is being used. If the event has a value of less than 0, the printer is being used and other partitions are waiting to use it.

Example Resource Sharing

In the following example, Program A wants exclusive access to a global resource, a graphics printer. Any other program should wait until Program A is finished with the resource before it tries to use it.

Program A:

```
500  CREATE EVENT "GRAPHICS AVLBL",1
  .
  .
  .
600  WAIT FOR EVENT "GRAPHICS AVLBL"
610  !Graphics statements
620  !and output here
  .
  .
  .
800  CAUSE EVENT "GRAPHICS AVLBL"
810  !Done with the resource. Now others can use it.
```

All programs must cooperate if this is to work. Only one partition creates the event; all others use it.

How to Create and Delete Events

When you create an event, using CREATE EVENT, you give it a name (up to 16 characters in length) and an initial value (default is 0). For example,

```
CREATE EVENT "SIMPLE"
CREATE EVENT "MY_PRINTER",0
```

When you delete an event, using DEL EVENT, no partitions can be waiting for that event; the event semaphore must be non-negative. Also, no ON EVENT can be active for that event in any partition.

For example,

```
100  IF EVENT LEVEL("Graphics")<0 THEN GOTO Waiting
110  ON ERROR GOSUB "On_event"
120  DEL EVENT "Graphics"
130  OFF ERROR
  .
  .
  .
700 On_event: !There is an ON EVENT active
710         PRINT "Cannot delete event because of"
720         PRINT "  active ON EVENT"
730         WAIT 5       !Wait for other partition to OFF EVENT
740         RETURN
```

How to Use Events

In the resource sharing example, an event was assigned to a printer. When a program wants to use that printer, it would execute a WAIT FOR EVENT statement. If the event is currently positive, the program does not wait but continues, and the event is decremented by one. If the event is less than one when the WAIT FOR EVENT statement is executed, the program is put in a wait queue.

In the WAIT FOR EVENT statement there is a CONDITIONAL option. If the CONDITIONAL option is used and the event level is less than one when the statement is executed, the program does not wait, the CONDITIONAL variable is set to zero and the event level is not decremented. If the event level is greater than zero, the CONDITIONAL variable is set to one and the event level is decremented. The program must check the CONDITIONAL variable to see if the WAIT was successful.

```
10 WAIT FOR EVENT "Printer";CONDITIONAL Result
20 IF Result=1 THEN GOTO Got_it
30 IF Result=0 THEN GOTO Others_wait
```

At any time, a program can query an event to determine its value. This is done with the EVENT LEVEL function.

```
A=EVENT LEVEL("Some event")
```

After a program has finished with the printer, or other device or process, it executes a CAUSE EVENT statement. CAUSE EVENT increments the event level by one. If there is a program waiting for the event, that program is released from the queue, and the event level is decremented by one by the released program's WAIT FOR EVENT statement.

CAUSE EVENT also triggers all ON EVENT statements active for that event. A program executes an ON EVENT statement and then continues execution. When a CAUSE EVENT statement is executed in any partition, any program which specified an ON EVENT branches to the line specified in the ON EVENT statement.

Example: Synchronization

The following program segments show how one program creates a partition and waits for the program in that partition to begin executing before it continues.

Program 1:

```
500  CREATE EVENT "OK TO GO",0
510  CREATE PARTITION "B";EXECUTE "LOAD"" PROGRAM2"",1"
520  WAIT FOR EVENT "OK TO GO"
530  !Program 2 has started execution now.
 .
 .
 .
```

Program 2:

```
10  !Program 2 is loaded
20  CAUSE EVENT "OK TO GO"
30  !Program 1 can now execute.
```

In this example, Program 1 cannot assume that Program 2 is **still** running when Program 1 continues to run. Program 2 could begin and finish executing before Program 1 gets any CPU time.

Example: Causing Action in Another Partition

The following program segments show how a program in one partition can cause an action to occur in another partition.

Program A:

```
100  CREATE EVENT "Get data"
 .
 .   Do some work
 .
500  CAUSE EVENT "Get data"             !Start other partition
510  ON EVENT "Get data" GOSUB Process_data
 .
 .   Do other work
 .
900 Process_data:                       !Other partition is done
910                                     !Process the data
```

Program B:

```
10  ON EVENT "Get data" GOTO30
20  WAIT
30
 .
 .  get the data
 .
210  CAUSE EVENT "Get data"             !Signal other partition
220  GOTO 10                            !Done with routine
```

The partition which creates the events and causes them could be a "master" partition in the foreground which interfaces with the background partitions.

DEADLOCK

Deadlock is a process-related situation that can arise when two or more processes are waiting for each other. Basically, a deadlock occurs when process A has resource 1 and process B has resource 2. Then A requests access to resource 2 and B requests access to resource 1 simultaneously. The net result is that B is waiting for A to release resource 1, but A cannot since it is waiting for B to release resource 2.

The BASIC system attempts to avoid user deadlock by allowing the STOP key to "bump" a process off of a semaphore. This avoids most but not all problems for user deadlock.

An example is the situation where you actually specify to forego the use of the "bumpable" semaphore by specifying the NONSTOP attribute in an ASSIGN@ statement. This can lead to user-instigated deadlock that permanently **locks up 2 or more partitions**.

The following example shows how this can be done. Note that if you try this example, you will have to turn off the computer to stop the deadlock.

In partition "A", we have the following program:

```
100 CREATE EVENT "DIE1"
110 CREATE EVENT "DIE2"
120 CREATE PARTITION "DEADLOCK";EXECUTE "LOAD ""DEADLOCK2"",1"
130 ASSIGN @A TO "FILE1";NONSTOP  !Requires IO option
140 ASSIGN @B TO "FILE2";NONSTOP
150 LOCK @A                       !DEADLOCK2 will lock @B
160 CAUSE EVENT "DIE1"            !Synchronize with DEADLOCK2
170 WAIT FOR EVENT "DIE2"
180 ASSIGN @B TO *
190 !THE PROGRAM WILL NEVER GET TO THIS LINE!!!!!!!
200 END
```

File "DEADLOCK2" contains the following program:

```
130 ASSIGN @A TO "FILE1";NONSTOP
140 ASSIGN @B TO "FILE2";NONSTOP
150 LOCK @B
160 WAIT FOR EVENT "DIE1"         !SYNCHRONIZE WITH DEADLOCK1
170 CAUSE EVENT "DIE2"
180 ASSIGN @A TO *
190 !THE PROGRAM WILL NEVER GET TO THIS LINE!!!!!!!
200 END
```

Step through this examples and see how this causes a deadlock between the two partitions.

1. DEADLOCK1 first creates two events, "DIE1" and "DIE2", for later use in synchronization.
2. DEADLOCK1 then creates a partition named "DEADLOCK2" and tells the new partition to load and run the program "DEADLOCK2".
3. The next 4 statements execute at the same time (for practical purposes).
4. DEADLOCK2 is loading and running its program.
5. DEADLOCK1 opens two files in NONSTOP mode, and then LOCKs @A. Meanwhile, DEADLOCK2 is opening the same two files and LOCKs @B.

You now have the necessary ingredients for deadlock to occur. One process (DEADLOCK1) has 2 resources (@A and @B) while another process has the same resources. The LOCK causes the other process (partition) to wait (for an UNLOCK) in order to gain access to that resource. Then, each program tries to close the file currently locked by the other program. These two partitions are now deadlocked! STOP does not help, because the I/O path names were declared to be non-stoppable. Otherwise, STOP would cause the files to be unlocked and closed, and an I/O aborted message would appear.

While this example may be artificial, it does demonstrate the essential characteristics of process deadlock. Note that if other partitions are present at the time the above deadlock occurs, then the entire system is not deadlocked: only these two partitions are permanently locked. You can still use NEXT PART, SHIFT STOP, as long as the keyboard resource was not the contested resource that caused the deadlock in the first place.

Message Manager Example

The following example, Message Manager, shows a way that partitions can send data to each other. This example demonstrates using events to synchronize access to a shared resource, in this case the message file. Note that this program does not handle errors. If you plan to use this program to send messages between partitions, replace the PAUSE statements with error handling routines.

```
130   SUB Init_msg_mgr
160   !
170   ! BASIC MESSAGE MANAGER
190   !
200   !
210   ! Description:
220   !
230   !    This is a "general purpose" message manager designed to
240   !    facilitate synchronization and information exchange between
250   !    cooperating partitions.  The message manager consists of the
260   !    following subroutines:
270   !
280   !        Init_msg_mgr : initializes the message manager.
290   !                       Multiply initializing the manager is ok.
300   !                       ALL users of the manager must initialize
310   !                       it in their partitions.
320   !                       Only partitions created at the time of
330   !                       the first (system-wide) call to init_msg_mgr
340   !                       are eligible to send and receive messages.
350   !        Send_message : Sends information to a partition.
360   !                       The message may be string or numeric data.
390   !        Recv_message : Receives information from another partition.
400   !
410   !    The message manager must be initialized before use by each
420   !    partition that wishes to participate in mesasge traffic.
430   !    There can be a maximum of 10 partitions.
440   !
450   !    Note that string messages can also be tagged as a keyboard string.
460   !    This can be used to execute any valid statement in another
470   !    partition.  An example is the STOP or RUN key.
480   !    The IO option must be present to send a keyboard string.
490   !    See the definition of SEND_MSG for more details.
500   !
501   !    The options required for these subprograms are:
502   !        MS, MEMORY_VOL and optionally IO.
510   !
520   !*****************************************************************
```

```
530   !
540   !
550   OPTION BASE 1
560   COM /Msg_mgr/ #1,#2,#3,#4,#5,#6,#7,#8,#9,#10,Parts$(11)[80],N_parts,Fsize, Dev$
570   INTEGER Size,Fill,Empty
580   Size=32000        ! Number of bytes of memory available to message
590                     !
600   Dev$=":MEMORY,0"! Device to hold the messages.
610                     ! :MEMORY,0 is fastest,  change to disc if you need
620                     ! more space.
630                     !
640       !
650       ! Get the names of the partitions currently in the computer.  Only
660       ! these partitions can participate in message traffic.
670       !
680   PARTITION STATUS  TO Parts$(*);ALL,NO HEADER,COUNT N_parts
690   IF N_parts>10 THEN
700      DISP "MESSAGE MANAGER INITIALIZATION ABORTED. TOO MANY PARTITIONS."
710      WAIT 2
720      DISP ""
730      SUBEXIT
740   END IF
750   Fsize=(Size-35*256) DIV N_parts
760       !
770       ! CHECK FOR ALREADY INITIALIZED
780       !
790   ON ERROR GOTO Initialized
800   CREATE EVENT "MSG_MGR_SEMA",1
810   OFF ERROR
820       !
830       ! SET UP EVENTS AND MESSAGE LINKS
840       !
850   IF Dev$=":MEMORY,0" THEN INITIALIZE Dev$&";DISCFORMAT SDF",Size DIV 256
860   Fill=5
870   Empty=5
880   FOR I=1 TO N_parts
890      CREATE EVENT Parts$(I)[1;16]
900      CREATE BDAT Parts$(I)[1;16]&Dev$,Fsize,1
910      ASSIGN #1 TO Parts$(I)[1;16]&Dev$
920      PRINT #1;Fill,Empty
930      ASSIGN #1 TO *
940   NEXT I
950       !
960       ! CREATE INFO FILES ON THE MEMORY VOLUME FOR OTHER PARTITIONS
970       ! TO USE IN INITIALIZING THE MSG_MGR.
980       !
990   CREATE BDAT "MSG_MGR"&Dev$,4
1000   ASSIGN #1 TO "MSG_MGR"&Dev$
1010   PRINT #1;N_parts,Parts$(*)
1020   !
1030 Initialized:  ! If another part has already initialized, then just open
1040                ! message link files and read the map.
1050                !
1060   ASSIGN #1 TO "MSG_MGR"&Dev$
1070   OFF ERROR
1080   READ #1;N_parts,Parts$(*)
1090   ASSIGN #1 TO *
1100   !
1110 Assign_files: !  Open the message link files.
1120   FOR I=1 TO N_parts
1130      ASSIGN #I TO Parts$(I)[1;16]&Dev$
1140   NEXT I
1150   !
1160   SUBEND
1170   !
1180   !
1190   SUB Send_msg(Rcvr$,Type,OPTIONAL Msg$,One_num,Nums(*),N)
1210   !
1220   !***********************************************************
1230   !
```

```
1240  ! SEND_MSG
1250  !
1260  !  Rcvr$= Name of partition to send the message to.
1270  !         Null string means all partitions.
1280  !
1290  !  TYPE = 0: STRING
1300  !         1: EXECUTE STRING
1310  !         2: NUMERIC
1320  !         3: NUMERIC ARRAY
1330  !
1390  !
1400  !  Msg$     = Value of string message.
1410  !
1420  !  One_num = Value of (single) numeric message
1430  !
1440  !  Nums(*)  = Value of numeric array message.
1450  !
1460  !  N        = Number of items in Nums(*)
1470  !
1480  !*****************************************************************
1490  !
1500  !
1510  COM/Msg_mgr/#1,#2,#3,#4,#5,#6,#7,#8,#9,#10,Parts$(*),N_parts,Filesize,Dev$
1520  INTEGER Fill,Empty
1530     !
1540     ! Search for the receiver
1550     !
1560  WAIT FOR EVENT "MSG_MGR_SEMA"    ! Makes sure I am only one preparing a
1570                                   ! message.  Can't allow contention
1580                                   ! for the fill and empty pointers
1590                                   ! in a given message link.
1600                                   !
1610  FOR I=1 TO N_parts
1620     IF TRIM$(Parts$(I)[1;16])=Rcvr$ THEN  GOTO Foundit
1630  NEXT I
1640  DISP "RECEIVER DOES NOT EXIST!"
1650  WAIT 2
1660  SUBEXIT
1670  !
1680 Foundit: !
1690     !
1700     ! Get the current queue pointers from the start of the receiver's file
1710     !
1720  READ #I,1;Fill,Empty            ! MESSAGES CAN START AT RECORD 5
1730  !
1740  SELECT Type
1750     CASE 0,1
1760        Size=12+LEN(Msg$)        !8 FOR TYPE, 4 FOR STRING LENGTH
1770     CASE 2
1780        Size=16                  !8 FOR TYPE, 8 FOR MSG
1790     CASE 3
1800        Size=16+N*8              !8 FOR TYPE, 8 FOR N,  +8*N
1810     CASE ELSE
1820        DISP "BAD MESSAGE TYPE"
1830        PAUSE
1840     END SELECT
1850     !
1860     ! QUEUE CONVENTIONS:
1870     !   FILL points to the next byte to write to
1880     !   EMPTY points to the next byte to read from
1890     !   FILL=EMPTY   ==> QUEUE IS EMPTY
1900     !   FILL=EMPTY-1 ==> QUEUE IS FULL
1910     !
1920     IF Fill+Size>Filesize THEN      ! Wrap-around.
1930        PRINT "WRAPAROUND"
1940        PRINT #I,Fill;-1             ! Flag to let reader know to wrap.
1950        Fill=5                       ! For now, do not allow a message to
1960                                     ! wrap.
1970     END IF !
```

```
1980     IF (Fill+Size) MOD Filesize=Empty THEN
1990        DISP "ERROR ---> QUE FULL!   MAKE FILES BIGGER"
2000        PAUSE
2010     END IF
2020         !
2030         ! Write message to file
2040         !
2050     SELECT Type
2060         !
2070        CASE 0,1
2080           PRINT #I,Fill;Type,Msg$
2090        CASE 2
2100           PRINT #I,Fill;Type,One_num
2110        CASE 3
2120           PRINT #I,Fill;Type,N
2121           FOR J=1 TO N
2122              PRINT #I;Nums(J)
2123           NEXT J
2130     END SELECT
2140         !
2150         ! Update fill and empty pointers
2160         !
2170     Fill=Fill+Size
2180     PRINT #I,1;Fill,Empty
2190         !
2200         ! Notify receiver of message on queue
2210         !
2220     CAUSE EVENT "MSG_MGR_SEMA"
2230     CAUSE EVENT Parts$(I)[1;16]
2240     !
2250  SUBEND
2260  !
2270  !
2280  !   RECEIVE MESSAGE SUPPORT
2290  !
2300  SUB Receive_msg(Result,OPTIONAL Msg$,One_num,Nums(*),N,Conditional)
2320  !
2330  !****************************************************************
2340  !
2350  !                        RECEIVE MESSAGE
2360  !
2370  !  Get a message from caller's message queue.  Msg_mgr must have been
2380  !  previously initialized.
2390  !
2400  !  PARAMETERS:
2410  !
2420  !  Conditional : =1 ->receive a message ONLY if there is one to receive.
2430  !                =0 ->receive a message.  Wait if one is not ready.
2440  !
2450  !  Result      : =0 ->message received was a string and is returned in
2460  !                     Msg$.
2470  !                =1 ->message received was a string which has been sent
2480  !                     to the keyboard.
2490  !                =2 ->message received was a single number (real) and
2500  !                     is returned in the One_num parameter..
2510  !                =3 ->message received was a real array and is returned
2520  !                     in Nums(*).  The number of elements read into
2530  !                     Nums(*) is returned in N.
2540  !                =4 ->No message was received.
2550  !
2560  COM/Msg_mgr/#1,#2,#3,#4,#5,#6,#7,#8,#9,#10,Parts$(*),N_parts,Filesize,Dev$
2570  DIM Name$(1:1)[80]         !NAME OF THIS PARTITION
2580  INTEGER I                  !FILE NUMBER FOR THIS PARTITION
2590  INTEGER Fill,Empty
2600     !
2610     ! Find this partitions name and file number
2620     !
2630  PARTITION STATUS  TO Name$(*);NO HEADER
2640  I=1
```

```
2650  IF N_parts=0 THEN
2660     DISP "ERROR ===> CALLER HAS NOT INITIALIZED THE MESSAGE MANAGER"
2670     PAUSE
2680  END IF
2690  !
2700  LOOP
2710     EXIT IF Parts$(I)[1;16]=Name$(1)[1;16]
2720     I=I+1
2730  END LOOP
2740     !
2750     ! Wait for message to become available.  User may have requested
2760     ! a conditional receive.
2770     !
2780  IF Conditional THEN
2790     WAIT FOR EVENT Name$(1);CONDITIONAL Result
2800     IF Result=0 THEN    ! NO MESSAGE. SET RESULT TO 4 AND RETURN
2810        Result=4
2820        SUBEXIT
2830     END IF
2840  ELSE
2850     WAIT FOR EVENT Name$(1)[1;16]
2860  END IF
2870     !
2880     ! Protect message traffic and read empty and fill pointers
2890     !
2900  WAIT FOR EVENT "MSG_MGR_SEMA"
2910  READ #I,1;Fill,Empty
2920     !
2930     ! READ MESSAGE TYPE
2940     !
2950  READ #I,Empty;Type
2960  IF Type=-1 THEN          ! WRAP-AROUND
2970     Empty=5
2980     READ #I,Empty;Type
2990  END IF
3000     !
3010     ! READ MESSAGE
3020     !
3030  SELECT Type
3040     CASE 0
3050        READ #I;Msg$
3060        Size=LEN(Msg$)+4
3070     CASE 1
3080        READ #I;Msg$
3090        Size=LEN(Msg$)+4
3091        OUTPUT KBD;Msg$
3100     CASE 2
3110        READ #I;One_num
3120        Size=8
3130     CASE 3
3140        READ #I;N
3141        FOR J=1 TO N
3142           READ #I;Nums(J)
3143        NEXT J
3150        Size=8+8*N
3160     CASE ELSE
3170        DISP "ERROR ==> BAD MESSAGE TYPE RECEIVED.   :::: ";Type
3180        PAUSE
3190  END SELECT
3200  Empty=Empty+8+Size
3210  PRINT #I,1;Fill,Empty
3220     !
3230     ! Release message manager semaphore.
3240     !
3250  CAUSE EVENT "MSG_MGR_SEMA"
3300  SUBEND
```

Chapter 9 Variables

A variable is one or more words of memory assigned a symbolic name or identifier in your program. There are two forms that any type of variable can have: simple or array. A simple variable is a single nonsubscripted data item, while an array is a collection of data items of the same type referred to with a common name and different subscripts.

The names given variables are known as BASIC names. This is because the naming convention is specific to this implementation of the BASIC language.

BASIC names are literals from 1 to 15 characters long. The first character is an uppercase letter (A thru Z) or an upper case national character from the range CHR$(161) thru CHR$(254). The remaining characters are lowercase letters, digits, the underscore character or national characters.

Some examples of BASIC names are:

```
New_value
Counter
Degrees_cent
```

The use of descriptive names improves the readability of your program.

Simple Variables

Simple variables have one value and are operated upon individually. There are two kinds of simple variables, simple numeric variables and simple string variables.

Simple Numeric Variables

Simple numeric variables are variables which contain one numeric value. That value can be an integer or real number. Here are some examples of assigning values to simple numeric variables.

```
Next_value=138
True_value=.002*89
Seconds=Minutes/60
```

Simple String Variables

It is often necessary to use non-numeric data in your computer operations. A word, a name or a message can be stored in the computer as a string of characters. `"WARNING!"`, `"Load Next Module Please..."` and `"1357.9"` are examples of character strings.

Like a numeric variable, each string variable has a name. String variables are differentiated from numeric variables by a dollar sign, '$', suffixed to their name. Some examples of string names are:

```
Name$
Address$
City$
Country$
Message$
Phone_number$
```

Some examples of assigning characters to a string variable are:

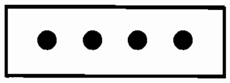
```
Name$="YOUR_NAME"
Address$="306 W. Edwards"
City$="HOUGHTON"
Message$="THE MESSAGE IS ""YES"" "
Phone_number$="906-482-4376"
```

Quotation marks are used to delimit the beginning and ending of the string, however they are not counted as part of the string. If you want to include a quote mark "" as part of your string, use two quotes, """.

Note that numbers also have character representations.

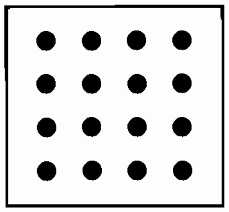
Array Variables

An array is a collection of data items of the same type. The structure of this collection is determined by the array's **dimensions**. A one-dimensional array, also known as a vector, can be thought of as a row of items. For example:



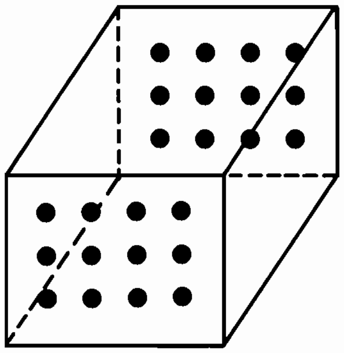
This array has one row and four data items.

A two-dimensional array, also known as a matrix, is composed of rows and columns. For example:



This array has four rows, four columns and twelve data items.

A three-dimensional array is comprised of planes, rows and columns. For example:



This array has two planes, three rows and four columns. There are a total of twenty four data items.

Although difficult to picture, four, five and six dimensional arrays are possible.

Array variables not only have a BASIC name, but subscripts to denote each element in the array using that BASIC name. Each element of an array is a simple variable which can be operated upon individually. However, the elements can also be operated upon together. Here are some examples of array elements.

```
Array_name(1,1)
Next_array(2)
Output_array(2,4,1,5,6)
```

Array elements can be denoted by as many as six subscripts. The number of subscripts needed is equal to the number of dimensions in the array. The actual number of elements which you can have in a single array is dependent upon the amount of memory in your machine.

You can also choose what values you want to use as subscripts. The acceptable range for a subscript is −2 147 483 647 to 2 147 483 647. The default for the lower bound for a subscript is zero.

A whole array, not just one element, can also be specified using the `(*)` symbol. For example,

```
PRINT Output_array(*)
```

means "Print the entire array `Input_array`".

Numeric Arrays

A numeric array consists of a collection of numbers of the same type. For example, you can have an array of integers or an array of real numbers. Generally, when you use an array, it is because the data in the array elements have some special relationship.

For example, you might consider using an array if you have 100 data points for a graph. In this case, the data is related in that they are all points on the graph. You probably wouldn't want to use a single array to hold data from a number of unrelated graphs.

String Arrays

Just as a numeric array is a collection of numbers of the same type, a string array is a collection of character strings of the same maximum length. Each string is one element in the string array like a number is one element in a numeric array.

Storage Formats

Data can be represented in a variety of ways inside your computer. Any variable, simple or array, is of a specific type or format. This "type" defines how the data in the variable is stored in memory. Each data type is best for representing a specific kind of data. Careful choice of data types for your variables can save you a great deal of memory and speed up processing of your programs.

Numeric Data

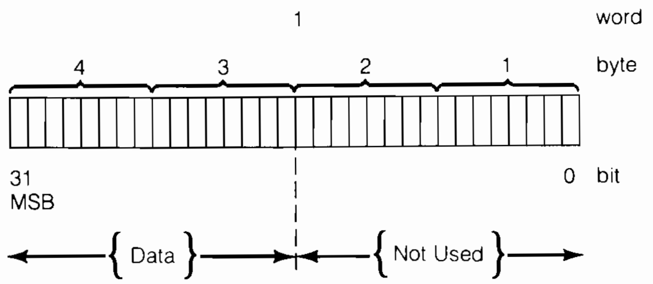
Numeric data types can be in either nonfractional or fractional form, referred to as integers or reals. Note the lowercase letters used in integer and real to denote the generic terms. Integers are identified with either a INTEGER or DOUBLE type declaration; reals with either a SHORT or a REAL type declaration. The upper case letters indicate a specific type.

If you do not explicitly identify a number as a specific type, the default type is REAL.

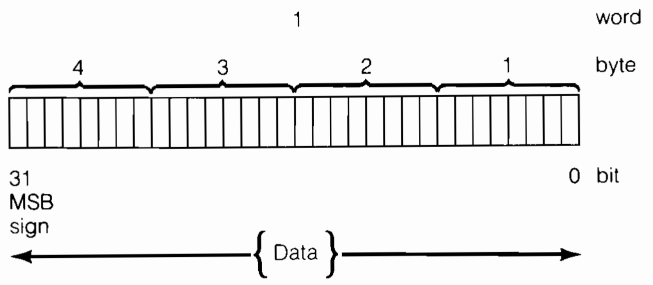
An integer number is stored in a two's complement binary representation, and a real number in a signed-magnitude and exponent representation. These representations differ from the binary coded decimal representation used by some computers. (For most applications, you needn't worry about this point. However, if you want more information, refer to the "Bit Manipulations" chapter.)

Integers

Your computer uses a 16-bit two's complement representation for INTEGER and a 32-bit two's complement representation for DOUBLE. The most significant bit (MSB) is the sign bit; 0 if non-negative and 1 if negative. The INTEGER and DOUBLE storage formats are shown in the following pictures.



A simple INTEGER occupies a half-word (2 bytes or 16 bits) and has a range from −32 368 to 32 367.



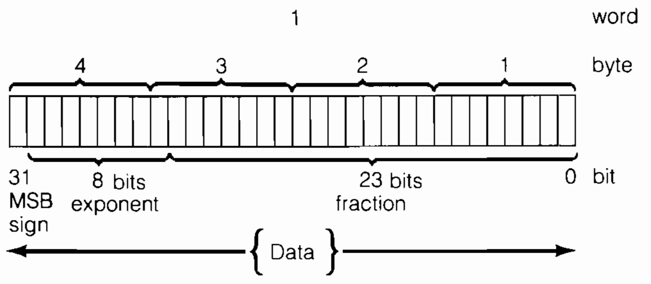
A simple DOUBLE variable occupies 1 word (4 bytes or 32 bits) and has a range from −2 147 483 648 to 2 147 483 647. Operations which use DOUBLE data are faster than those which use INTEGER data.

Reals

A sign-magnitude representation is used for real numbers. Again two formats are supported: SHORT, a 32-bit format and REAL, a 64-bit format.

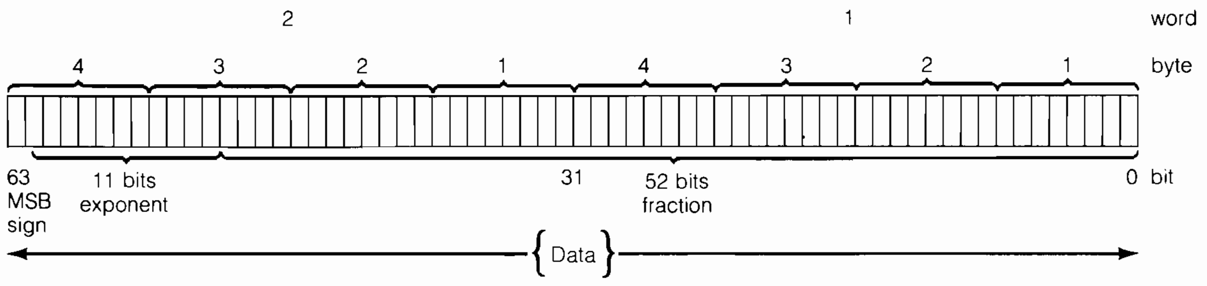
Real numbers have their range expressed in the their exponent and their precision expressed in their fraction or mantissa. The number takes the form; fraction × base^E, where E is the exponent.

SHORT storage format looks like:



The exponent is the binary exponent plus 127. There are seven significant digits, in base ten, represented by the fractional part. A simple SHORT variable occupies one word (4 bytes or 32 bits) and has a range from $\pm 1.2 \times 10^{-38}$ to $\pm 3.4 \times 10^{38}$.

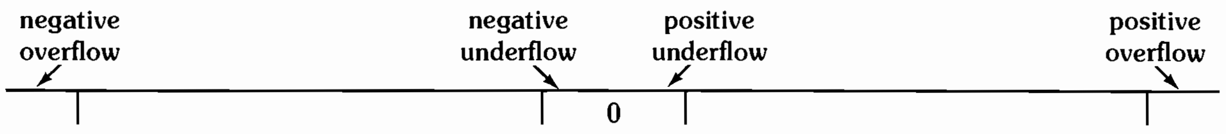
REAL storage format looks like:



The exponent is the binary exponent plus 1023. There are a maximum of sixteen to seventeen significant digits in base 10 represented by the fractional part. A simple REAL variable occupies two words (8 bytes or 64 bits) and has a range from $\pm 2.2 \times 10^{-308}$ to $\pm 1.8 \times 10^{308}$.

The sign of the binary fraction is in the most significant bit (MSB) and is a 0 if plus and a 1 if minus. These numbers are stored in a normalized form in which the binary point is to the left of the fraction field and an implied leading 1 is to its left. Refer to the "Bit Manipulations" chapter for information about normalizing real numbers.

There are some numbers that are impossible to represent in a real storage format. The following chart shows these regions:



Overflow and underflow occur at the points where the number becomes denormalized and depend on the type, SHORT or REAL, that you are using.

To avoid overflow and underflow errors, use the DEFAULT ON statement. If a real math error occurs when you use the DEFAULT ON statement, the appropriate default value is substituted and execution continues. With the DEFAULT option on, a REAL overflow produces ±MAXREAL, while a SHORT overflow reults in ±SHORT of MAXREAL. Underflow in either case causes a 0 to be places in the REAL/SHORT variable.

MAXREAL is approximates 1.797×10^{308} and MAXREAL put into (coerced to) a SHORT variable (with the DEFAULT ON statement active) is approximately 3.402×10^{38}. MINREAL is approximately 2.225×10^{-308} and when coerced into a SHORT variable (with the DEFAULT ON statement active) gives a value of 0.

Denormalized, INF (INFinity) and NAN (Not A Number) numbers are not supported and cause an error.

Since there is no way to represent certain decimal numbers in binary arithmetic, be careful in the use of REALs and SHORTs in boolean, comparison, and FOR...NEXT operations. For example:

```
 1   REAL A
10   SHORT S
11   A=.1000000
12   S=A
20   IF S=A THEN
21      PRINT "S CAN EQUAL EXACTLY 0.1000000"
22   ELSE
23      PRINT "NUMBERS NOT EQUAL GOING FROM DECIMAL TO BINARY."
24   END IF
30   END
```

Output:

```
NUMBERS NOT EQUAL GOING FROM DECIMAL TO BINARY.
```

Since S is not equal to 0.1000000, the following problem can occur in the above program.

```
110   If S=0.10000000 THEN STOP
```

the STOP condition is never true.

The "Bit Manipulations" chapter discusses bit representations of decimal and binary numbers.

Which Type to Use

There are two major considerations to choosing a variable's data type.

1. Is that variable always going to contain a whole number or will it sometimes have a fractional part?
2. What is the range of values that the variable is expected to contain?

If the variable always contains an integer, then use the INTEGER or DOUBLE data type. If the variable has values in the range −32 768 to 32 767, INTEGER is the most space saving choice. For integers in the range from −2 147 483 648 to 2 147 483 647, use the DOUBLE data type.

If your number contains a fractional as well as a whole number part, use the SHORT or REAL data types. If the variable's values are in the range $\pm 1.2 \times 10^{-38}$ to $\pm 3.4 \times 10^{38}$, use SHORT. If the variable's values are in the range of $\pm 2.2 \times 10^{-308}$ and $\pm 1.8 \times 10^{308}$, use REAL.

SHORT or REAL should also be used for those variables which only contain integers but which need a larger range. Note that the precision of the values are less.

Conversion Between Numeric Data Types

Some applications require the transfer of numbers from one data type to another. Because of inherent differences in internal format, values may be altered in the transition producing incorrect results during computation.

When numbers are converted from SHORT or REAL to INTEGER or DOUBLE, they are first rounded to a whole number. (If the first digit to the right of the decimal is five or greater, the units digit is increased by one, if it is four or less, the units digit remains unchanged.) If the real variable is outside of the range of integer space for which it is targeted, an overflow/underflow error occurs.

Conversion from REAL to SHORT which results in a value outside of the SHORT range again causes an overflow/underflow error.

Precision can be lost when putting a REAL or DOUBLE into a SHORT. The following is an example of the precision lost due to rounding in going from DOUBLE to SHORT.

```
10  DOUBLE Big_integer
20  SHORT Little_real
30  Big_integer=123455541
40  Little_real=Big_integer
50  PRINT "THE BIG INTEGER IS ",Big_integer
60  PRINT "THE LITTLE REAL IS ",Little_real
70  END
```

Output:

```
THE BIG INTEGER IS 123455541
THE LITTLE REAL IS 123455500
```

The FIXED 1 statement in the following two programs forces the numeric output to be printed as real numbers with one digit to the right of the decimal point.

```
 1   FIXED 1
10   DOUBLE Big_integer
20   SHORT Little_real
30   Big_integer=123451111
40   Little_real=Big_integer
50   PRINT "THE BIG INTEGER IS ",Big_integer
60   PRINT "THE LITTLE REAL IS ",Little_real
70   END
```

Output:

```
THE BIG INTEGER IS 123451111.0
THE LITTLE REAL IS 123451112.0
```

```
 1   FIXED 1
10   DOUBLE Big_integer
20   SHORT Little_real
30   Big_integer=123455541
40   Little_real=Big_integer
50   PRINT "THE BIG INTEGER IS ",Big_integer
60   PRINT "THE LITTLE REAL IS ",Little_real
70   END
```

Output:

```
THE BIG INTEGER IS 123455541.0
THE LITTLE REAL IS 123455544.0
```

Notice that when more than seven significant decimal digits in a SHORT number are forced to print out with the FIXED statement, the digits to the right of the seventh digit in the SHORT number have no relationship to the digits to the right of the seventh digit in the original DOUBLE number. The digits printed are, however, the decimal representation of the actual binary number stored.

The "Bit Manipulations" chapter discusses this concept in greater detail.

String Data

String data is a series of characters such as "Abc4 + %zx". This type of data is stored as a series of bytes, each representing one character.

The binary value of a single byte, normally considered un-signed, is in the range 0 through 255 (decimal). Each numeric value represents a single character.

Bytes with numeric values in the range of 0 through 127 represent ASCII characters, that is, the matching of a numeric value with the graphic or symbolic representation of the character on a printer or display, is defined by the American Standard Code for Information Interchange. There are two groups of characters in this range: the control characters (0 through 31 and 127) and the displayable characters (32 through 126)

Bytes with numeric values in the range 128 through 255 represent **extension** characters specific to your computer. The symbolic representation of these characters is defined only for the internal printer and display of your computer. There are also two groups of characters in this range: the extension control characters (128 through 159 and 255) and the extension displayable characters (160 through 254).

Refer to the Character Codes Table in the "Useful Tables" section of the BASIC Language Reference for a list of character codes and their graphic symbols. Refer to the Control Character Effects table in that manual for a discussion of the use of the two ranges of control characters.

When you declare a string variable, enough memory is reserved to accommodate the maximum declared length of the string. A string consists of a word containing the "current length" of the string, followed by a byte array containing the characters of the string. A zero is placed in the current length word when the string is created.

String storage format in memory looks like:

WORD	**1**	**2**	**3**	**4**	**5**	**6**	.....
LENGTH	C.L.	**1-4**	**5-8**	**9-12**	**13-16**	**17-20**	.....

(C.L. is the current length.)

Only the number of bytes equal to the current length is accessable. This current length can vary during program execution from the zero, or null string, up to the maximum length initially declared for the string. Although the acceptable amount of storage for an individual string is 1 to 2 147 483 647 bytes, the actual length is bounded by the amount of memory available. The default length of a string variable is eighteen characters. The current length of the string at the time of creation is zero.

The total number of bytes needed for storage is equal to:

$$\text{string bytes} = ((7+\text{length})/4)\times 4$$

Here is an example.

```
String$="TEST1"
```

The length of the String$ is 5 characters long, so that the number of bytes, without overhead, necessary for storage is:

$$((7+5)/4)\times 4 = 12 \text{ bytes or } 3 \text{ words.}$$

Notice that the total number of bytes needed is rounded up to the next full word.

Specific examples of initial and assignment conditions of memory are:

Initial condition:

WORD	**1**	**2**	**3**	**4**
LENGTH	**0**	**reserved but inaccessable**		

Assignment condition:

	1	**2**	**3**		**4**
WORD	**1**	**2**	**3**		**4**
BYTE	**1 2 3 4**	**1 2 3 4**	**1 2 3 4**		**1 2 3 4**
	5 **current** **length**	**T E S T**	**1**	**inaccessable**	

Initially four words are saved to store Name$. When Name$ is assigned the value "TEST1", a 5 is stored in word 1 since there are five characters or letters in "TEST1". The characters "T","E","S","T" are stored in word 2, and a "1" is stored in the first byte of word 3. The last three bytes of word 3 as well as all of word 4 contain currently inaccessable information.

Array Data

Arrays are stored sequentially in memory in row major order. Row major order means that all columns of a row are stored sequentially, proceeding from the first row to the last row, in successive planes. In a multi-dimensional array, the right-most subscript varies fastest and the left-most subscript varies slowest.

Here are some examples of row major storage. Note that the lower bound of the array element subscripts is one.

One-dimensional Array: Array(Column)

Array(6) Column

1	**2**	**3**	**4**	**5**	**6**
x	x	x	x	x	x

Stored as:

1	**2**	**3**	**4**	**5**	**−6**

Two-dimensional Array: Array(Row,Column)

Array(3,4)

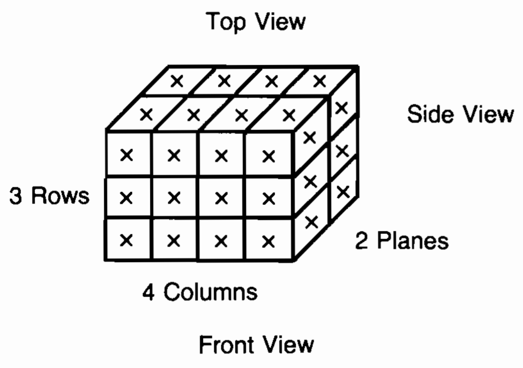
	Column			
	1	2	3	4
1	x	x	x	x
Row 2	x	x	x	x
3	x	x	x	x

Stored as:

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

Three-dimensional Array: Array(Plane,Row,Column)

Array(2,3,4)



Stored as:

				Plane 1							
row 1				row 2				row 3			
1,1,1	1,1,2	1,1,3	1,1,4	1,2,1	1,2,2	1,2,3	1,2,4	1,3,1	1,3,2	1,3,3	1,3,4

				Plane 1							
row 1				row 2				row 3			
2,1,1	2,1,2	2,1,3	2,1,4	2,2,1	2,2,2	2,2,3	2,2,4	2,3,1	2,3,2	2,3,3	2,3,4

Four, five and six dimensional arrays are stored in a similar fashion, with the right-most subscript varying fastest to the left-most subscript varying slowest.

Storage order becomes important when using a statements such as:

```
260  PRINT Array(*)
```

when the entire array is printed out in the order it is stored.

```
 10  OPTION BASE 1
 20  INTEGER Array(2,3,4)
 30  Array(1,1,1)=1        !You may be wondering why a loop
 40  Array(1,1,2)=2        !was not used.
 50  Array(1,1,3)=3        !
 60  Array(1,1,4)=4        !It was because explicit input into
 70  Array(1,2,1)=5        !each array element was wanted.
 80  Array(1,2,2)=6
 90  Array(1,2,3)=7
100  Array(1,2,4)=8
110  Array(1,3,1)=9
120  Array(1,3,2)=10
130  Array(1,3,3)=11
140  Array(1,3,4)=12
150  Array(2,1,1)=13
160  Array(2,1,2)=14
170  Array(2,1,3)=15
180  Array(2,1,4)=16
190  Array(2,2,1)=17
200  Array(2,2,2)=18
210  Array(2,2,3)=19
220  Array(2,2,4)=20
230  Array(2,3,1)=21
240  Array(2,3,2)=22
250  Array(2,3,3)=23
260  Array(2,3,4)=24      !Finally finished!
270  PRINT Array(*)
280  END
```

Output:

```
1          2          3          4
5          6          7          8
9          10         11         12
13         14         15         16
17         18         19         20
21         22         23         24
```

The number of bytes of overhead needed for arrays is equal to (the number of dimensions + 1) × 12.

Chapter 10
Declaring Variables

Your program needs to reserve memory for the variables it uses. There are two kinds of memory in which to do this. The first uses the local program environment, while the second uses data segments external to the local program environment. Local storage is invoked by using INTEGER, DOUBLE, SHORT, REAL and DIM declarations, while ALLOCATE, BUFFER and COM reserve external storage. Although programs which use externally stored variables take longer to execute, they do have certain features which make them desirable. The choice of which to use is dependent upon your program needs.

Local Storage

Local storage is that memory which has been specifically set aside to hold the data for one particular program. Local storage is much quicker to access than external storage but its data is more difficult to pass to other programs.

Declaring Simple Numeric Variables

You can declare simple numeric variables two ways: implicitly and explicitly.

Implicit Declaration

To implicitly declare a variable, just assign a value to a BASIC name. For example, the following program lines all implicitly declare a simple numeric variable.

```
 10  New_value=123
 35  FOR X=1 TO 100
200  Sink_value=Source_value*Phasor
```

If `New_value`, `X` and `Sink_value` are introduced for the first time at these lines in the program, memory is allocated for them when the lines are executed and they are assigned the values specified in the statements.

Implicitly defined numeric variables are always of type REAL.

Explicit Declaration

To explicitly allocate local storage for simple numeric variables, use the INTEGER, DOUBLE, SHORT or REAL statements. Here are some examples.

```
 10  INTEGER Small_integer,Month,Inventory
 40  DOUBLE Large_integer
100  SHORT Small_real
150  REAL Large_real,Orbit
```

`Small_integer`, `Month` and `Inventory` are all INTEGER variables. `Large_integer` is a DOUBLE and `Small_real` is a SHORT. `Large_real` and `Orbit` are REALs.

The DIM statement can also be used to declare REAL variables. For example:

```
20   DIM Cpu_time,Next_real
```

Declaring Simple String Variables

The maximum number of characters a string variable can contain must be either explicitly or implicitly declared before you use it. This is called dimensioning a string variable. Although this maximum string length can be any positive integer up to 2 147 483 647, the actual length is limited by the amount of memory available in your machine. The computer saves this maximum length as well as the current number of characters you have assigned to your string variable.

Implicit Dimensioning

As with numeric variables, to implicitly define a string just assign a value to a string name. For example:

```
112   New_string$="Exec.profile must be re-run."
 34   Error$="WARNING, POSSIBLE OVERFLOW."
212   Next_string$=Old_string$
```

If you do not explicitly dimension a string, the default length is eighteen characters. Strings whose maximum length is smaller than eighteen can be dimensioned explicitly, thereby saving space.

Explicit Dimensioning

Explicit dimensioning of string variables increases the readability of your programs, results in fewer errors, facilitates documentation and maintenance, and can save space in memory.

The DIM statement explicitly dimensions and reserves memory for simple string variables. Here is an example.

```
30   DIM Name$[34],String$[2],Real_value
```

The square brackets [ ] specify the maximum number of characters in the corresponding string. Note that the example DIM statement also defines a real variable, `Real_value`.

When this statement is executed, all strings specified have a maximum length equal to the dimensioned length and a current length of zero. After you assign some values to your string, this current length changes to the number of characters in the string. Only the number of characters less than or equal to the current length are accessible to you.

Although you can put less than the maximum declared number of characters into your string, if you try to put more than the maximum number of characters into the string, BASIC ERROR 18 - "Substring out of range or bad string length." is generated.

Dimensioning an Array

To use an array, your computer must know the array's size - the number of dimensions, or rank, and the number of elements, or size, of each dimension. When an array is dimensioned, its maximum size is stated so that the system can reserve the appropriate number of words of storage. These data items can be character data or any type of numeric data; however, all the data items in an array must be of the same type (INTEGER, REAL, String and so on).

You use subscripts enclosed in parentheses to dimension arrays. Each dimension is separated by a comma. The naming of the elements in any array is done by specifying where you want each dimension to start and where you want the dimension to end.

Implicit Dimensioning
If an array element is used in a program or keyboard computation, but has not been previously defined in a declaration statement, the array is said to be implicitly dimensioned. For example:

```
60  City_pop(2,2,6)=25.7
```

Implicitly defined arrays have the default lower bound (as specified by the OPTION BASE statement) and an upper bound of 10 for all dimensions. For example, with OPTION BASE 0, `City_Opop` has eleven planes, eleven rows and eleven columns as its maximum size. The value 25.7 is placed in element, `City_pop(2,2,6)`.

All future references to an implicitly dimensioned array must refer to the array as having the same number of dimensions.

Explicit Dimensioning
The maximum working size of an array can be specified in the INTEGER, DOUBLE, SHORT, REAL and DIM declaration statements. Some examples are:

```
 1  OPTION BASE 1
10  INTEGER Num_students(15,40),Counter
20  DOUBLE Num_photons(3600),City_pop(500),Number
30  SHORT Job_stats(6,2,2), Scale_factor
40  REAL Salary(4500),Accumulator
50  DIM Name$(60)[50],Location$,Table(45)
```

Note that single variables (scalars), as well as arrays, can be declared with the same statement.

To work with the total number of students at a university with fifteen departments, each offering forty different classes, the first INTEGER array `Num_students` could be used. This tells the computer to reserve 15×40 (600) INTEGER storage units for `Num_students`. Since each INTEGER array element requires only sixteen bits, or one-half of a word for storage, `Num_students` requires 300 words in memory. INTEGER is the most space saving choice for arrays of numeric data which are all in the range from −32 768 to 32 767. Since single INTEGER variables (scalars) use one full word of storage, they provide no space savings compared to scalar DOUBLEs. Accessing INTEGER scalars and arrays is slower than accessing DOUBLE scalars and arrays.

The discrete number of photons emitted per second over an hour period of time requires a memory location that can store these larger integers. The DOUBLE array, `Num_photons`, can be used. Each element in a DOUBLE array occupies one word; `Num_photons` reserves 3600 words in memory. The values in each element of the DOUBLE array can be in the range from −2 147 483 648 to 2 147 483 647.

An adult physical fitness program might want to monitor the average monthly number of miles run and time required over a two month period, for each of their six members, by using the SHORT array, `Jog_stats`. These are real numbers (with a fractional part) such as 12.5 miles or 98.52 minutes. Each element in a SHORT array occupies one word in memory; `Jog_stats` reserves 24 (6×2×2) words. One can be fairly sure that the numbers represented remains in the SHORT range from 1.2×10^{-38} thru 3.4×10^{38}. REAL numbers have approximately sixteen digits of precision, which may be important to the accuracy of your program.

Since the data items in an array all must be of the same type, the names of the adults in the program cannot be stored in `Jog_stats`, which is a numeric data type. The DIM statement reserves a corresponding string array, `Name$`, to contain these names. When the string array is dimensioned, the number of elements in the array is specified first with round brackets ( ), and then the number of characters each element can hold is specified with square brackets [ ], as in `Name$(6)[5]`. The default number of characters for a string variable is eighteen.

`Jog_stats` and `Name$` are manipulated in a single loop in the following program.

```
 10  OPTION BASE 1
 20  SHORT Jog_stats(6,2,2)
 30  DIM Name$(6)[5]
 40  READ Name$(*)
 41  DATA "HENRY","JOE","MARCY","SARAH","TOM","WENDY"
 50  READ Jog_stats(*)
 60     DATA  8.4,  71.2,  9.8,  82.4
!DATA FOR HENRY
 70  DATA 5.2, 45.6, 4.8, 41.3        !DATA FOR JOE
 80  DATA 6.3, 48.1, 6.3, 47.2        !DATA FOR MARCY
 90  DATA 2.1, 23.4, 2.2, 22.3        !DATA FOR SARAH
100  DATA 6.2, 36.3, 6.3, 35.4        !DATA FOR TOM
110  DATA 4.7, 41.3, 4.2, 42.6        !DATA FOR WENDY
120  !
121  PRINT ,"MILES    MINUTES     MILES    MINUTES"
130  FOR I=1 TO 6
131    PRINT Name$(I),
140    FOR J=1 TO 2
150       FOR K=1 TO 2
160          PRINT Jog_stats(I,J,K);"      ";
170       NEXT K
180    NEXT J
181    PRINT
190 NEXT I
200 END
```

Outputs:

```
            MILES      MINUTES        MILES       MINUTES
HENRY        8.4         71.2          9.8         82.4
JOE          5.2         45.6          4.8         41.3
MARCY        6.3         48.1          6.3         47.2
SARAH        2.1         23.4          2.2         22.3
TOM          6.2         36.3          6.3         35.4
WENDY        4.7         41.3          4.2         42.6
```

The salaries of 4500 computer programmers in an electronics firm can be stored in the REAL array, `Salary`. Since each element in a REAL array occupies two words, `Salary` reserves 9000 (4500×2) words. The dollar value of any one of these salaries will probably not exceed the maximum value of 1.8×10^{308}. (the range for a REAL variable must be -2.2×10^{-308}. to $1.8...10^{308}$.

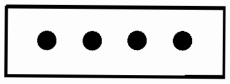
You can set the lower and upper bounds for each dimension of an array be specifying integers separated by a colon. For example:

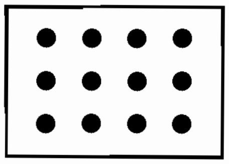
```
    10  INTEGER Cloud_types(1970:1982)
```

`Cloud_types` has one dimension with thirteen elements, the first being 1970 and the last, 1982. Specifying the upper and lower bounds for your array permits more meaningful subscripts.

A single constant used as a subscript indicates an upper bound and must be a non-negative integer in the range of DOUBLE and greater than or equal to the current lower bound setting. Subscripts can be in the range of ±DOUBLE; however, you can use negative number only when specifying lower and upper bounds.

In the following diagram, the one-dimensional array can be names `Array(1:4)` to indicate that it contains one row with a maximum of four columns. `Array(1:4)` contains four data items or elements.





The two-dimensional array could be name `Array(1:3,1:4)` to indicate that it has a maximum of three rows and four columns. Instead of repeatedly specifying a lower bound 1, the OPTION BASE statement can be used. For example:

```
    10  OPTION BASE 1
    20  REAL Array(3,4),Sum(2,3,4)
```

Note that dimensions are added on the left. When looking at the sequential storage of the array in memory, the right-most subscript varies fastest and the left-most subscript varies slowest.

The OPTION BASE statement can occur only once in each context. If use, OPTION BASE must precede any explicit variable declarations. The two values for OPTION BASE are 0 and 1.

Redimensioning an Array

By redimensioning an array, you can reorganize it into a more useful configuration. When an array is redimensioned, the bounds in each dimension can be changed and it is given a new working size. Any elements not included in the new working size are not accessible but are still saved in memory as part of the array. When new values are assigned to elements of a redimensioned array the values of the inaccessible elements are not changed.

A redimensioned array must retain the same number of dimensions as originally specified. The total number of elements cannot exceed the number originally specified. The maximum length of a string element cannot be changed.

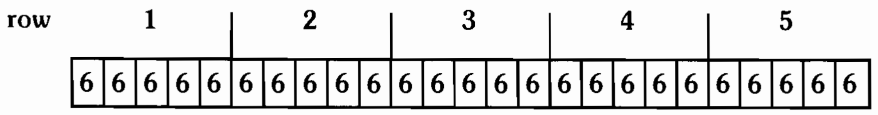
The following program illustrates what happens when an array is redimensioned.

```
 10   OPTION BASE 1
 20   INTEGER Array(5,5)
 30   MAT Array=(6)
 40   PRINT "THE ORIGINAL 5 BY 5 ARRAY IS:"
 41   FOR I=1 TO 5
 42     FOR J=1 TO 5
 43         PRINT "          ";Array(I,J);
 44     NEXT J
 45     PRINT
 47   NEXT I
 48   !
 50   REDIM Array(2,3)
 60   MAT Array=(3)
 61   PRINT
 70   PRINT "THE REDIMENSIONED 2 BY 3 ARRAY IS:"
 71   FOR I=1 TO 2
 72      FOR J=1 TO 3
 74          PRINT "          ";Array(I,J),
 75      NEXT J
 76      PRINT
 78   NEXT I
 79   !
 80   REDIM Array(5,5)
 81   PRINT
 90   PRINT "Array IS NOW BACK TO ITS ORIGINAL SIZE."
 91   FOR I=1 TO 5
 92      FOR J=1 TO 5
100          PRINT "          ";Array(I,J);
101      NEXT J
102      PRINT
103   NEXT I
110   PRINT "NOTE WHICH ELEMENTS CHANGED."
120   END
```

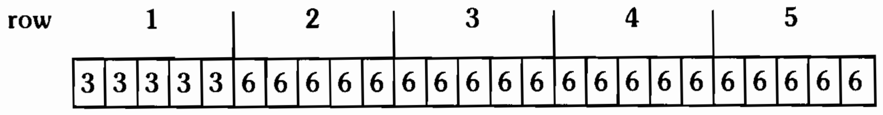
Outputs:

```
THE ORIGINAL 5 BY 5 ARRAY IS:
 6          6          6          6          6
 6          6          6          6          6
 6          6          6          6          6
 6          6          6          6          6
 6          6          6          6          6

THE REDIMENSIONED 2 BY 3 ARRAY IS:
 3          3          3
 3          3          3

Array NOW BACK TO ITS ORIGINAL SIZE:
 3          3          3          3          3
 3          6          6          6          6
 6          6          6          6          6
 6          6          6          6          6
```

The elements which have changed are the result of the row major storage. Before the REDIM statement the section of memory storing Array(5,5) looked like:



After the REDIM `Array(2,3)` and subsequent re-assigning of the number 3 into each element, the section of memory storing `Array(5,5)` held:



When you redimension an array, the first memory locations are used as the spaces for the values of the redimensioned array.

External Storage

In addition to storing data in the local program environment, you can store data in data segments external to the program environment. COM, ALLOCATE and BUFFER reserve external storage.

COM

COM is an area of memory in which variables in a "common" memory area are accessable to more than one program context. COM is typically used so that a subprogram does not have to allocate its own memory space for these variables and to reduce the number of passed parameters. Common can be accessed only in the partition in which it was created. Here is an example of a COM declaration.

```
200  COM Input_data,INTEGER Output_array(10),File$,Scratch_data
```

Specifying a variable type implies that all variables which follow in the list are of the same type. The default type is REAL. After a string variable, I/O path name or file number, unspecified numeric variables are REAL.

The variable `Scratch_data` would be REAL, since it occurred after the string variable `File$`.

There are both labelled and unlabelled common areas. Variables declared in COM statements without a block label become part of "blank" common. A labelled area merely gives the block of common variables a name. It is mandatory for the common label to be the same for all contexts using it; it is mandatory for all the corresponding variables to be of the same type.

The previous COM example wa an unlabelled COM statement. An example of a labelled COM statement is:

```
20  COM /Test_Data/Test_#,Input,INTEGER Output(10),File$,Data
```

Unlabelled COM variables have a faster access time than labelled COM variables.

Non-global blocks are allocated the first time they are encountered in a subprogram and are eliminated when the same or another program is run (only if they are not redeclared).

Here is an example of a MAIN program and a subprogram sharing the COM group of variables.

```
 10  COM /Group/ Init,Offset,Test_data(10)
 20  Init=5.0
 30  CALL Sub_program
 40  Test_data(2)=6.5
 50  Test_data(10)=Test_data(2)+Init+Offset
 60  PRINT "Test_data(10) equals ";Test_data(10)
 70  END
 80  SUB Sub_program
 90  COM /Group/ Reset,Temp,Test_data(10)
100  Temp=10.0
110  SUBEND
```

Outputs:

```
Test_data(10) equals 21.5
```

The value of the variable, `Offset`, calculated in `Sub_program` under the corresponding name of `Temp`, is passed to the main program through the common block labelled `Group`.

Array declaration in the COM statements used in a subprogram can either explicitly specify the array bounds or use the (*) notation. If explicit bounds are specified, they must match in size and number of dimensions with their corresponding COM declarations. If COM statements using the implicit (*) notation for an array are encountered for a COM block which does not exist, an error occurs, since the system does not have enough information to reserve appropriate storage for the array.

Names in the COM declarations need not match; however the types must be the same.

ALLOCATE

The ALLOCATE statement dynamically allocates memory during program execution. Only as much storage as is actually needed by a specific run of the program is therefore allocated.

Memory reserved by the ALLOCATE statement is freed by the DEALLOCATE statement. These deallocated segments, or pieces of memory, are automatically collected and returned to the available memory list. ALLOCATED segments are deallocated when the program or subprogram is exited.

If your program requires many big numbers or arrays, (such as when using large arrays for temporary storage or computations) which do not all need to be in memory at the same time, use ALLOCATE/DEALLOCATE. This reserves the memory as it is needed and then returns the space when you are finished with it.

Variables listed in the ALLOCATE statement can be passed in a parameter list. The variables in an ALLOCATE statement cannot appear in any declaration statement or be implicitly declared within the same program or subprogram context. The default type is REAL. The subscript bounds in array declarations and the string length specifiers may be numeric expressions; since these are evaluated at run time when the array is allocated. These are specifically prohibited in other declaration statements.

The total number of bytes allocated for all variables is limited by partition memory. The maximum size of any string or array is 524 284 bytes. As the unit size of a variable goes up, the limit goes down; INTEGER array - 262 130 elements, REAL - 65 532 elements.

Numeric arrays are REAL unless explicitly declared otherwise. When a type is specified, all numeric arrays and variables are of that type until another type or string is specified. After a string, numeric arrays and variables are again REAL.

Here are some examples of the ALLOCATE statement.

```
110   ALLOCATE A,B,DOUBLE Big(12*M,2*X-1,3),Name$,Default_real
170   ALLOCATE Something(6,7,8)
```

After the string variable `Name$`, the default type is REAL.

BUFFER

BUFFER is a parameter which causes space to be allocated out of absolute memory and not out of the local context memory. Absolute memory is an area which is identified as a specific location in memory and never relative to any starting point.

BUFFERs are used to hold incoming or outgoing numeric array or string scalar data. When the input or output has exhausted the supply of data in the buffer, the buffer is then refilled. Wear is reduced on the medium since it does not have to be accessed as often.

The BUFFER parameter:

- can be used to increase execution speed, since another I/O process accessing the BUFFER used in a TRANSFER statement may proceed in parallel with program execution;
- can assign a maximum of 0.5 Mbytes to any one variable in any separate context, that is each subprogram.

Here are some examples of using the BUFFER parameter.

```
10   ASSIGN @Io_path_name TO BUFFER Array(*)
20   INTEGER Array(-128:255) BUFFER
```

You can increase the amount of memory available to your local variables by using the BUFFER parameter. For example, the addition of the BUFFER parameter in statement 10:

```
10   DIM A(65536) BUFFER
20   DIM B(10)
30   END
```

causes the program to run without error. The memory given to the array, `A`, is out of external system storage. The array, `B`, has a beginning address which is a small number (offset from the origin of the context's starting address).

In the memory restrictions example, dimensioning the array, `A`, without the BUFFER parameter causes the array `A`'s ending address to be at the limit of the local context's memory address space. Now line 20 dimensioning the array, `B`, causes `B`'s starting address ot be a bigger number than the context can handle. A memory overflow error occurs.

Since each variable with a BUFFER parameter can have up to 0.5 Mbytes, you can use up to the available memory of your computer with several BUFFER variables. The LIST command, outputs the available memory.

An easier way to gain the maximum amount of memory for your variables is to use COM. Each named COM block is limited to 0.5 Mbytes of memory exclusive of BUFFER variables. There is a limit of 512 different COM blocks, including blank COM. The size of blank COM is limited by the computer's memory size.

When a string is initially declared with the BUFFER parameter, the maximum length is placed in the first word as the current length and the remainder of the byte string is filled with null characters. In the following example, the current length of `String$`, as found by the LEN function, is 10 although no assignment had been made to the string. The NUM function found a 0, which is the ASCII representation of the "NULL", or skip character, in position one of `String$`.

```
10   DIM String$[10] BUFFER
20   INTEGER Ascii
30   Num=LEN(String$)
40   Ascii=NUM(String$[1;1])
50   PRINT " THE LENGTH OF THE ASCII STRING IS",Num
60   PRINT " THE ASCII REPRESENTATION OF THE FIRST CHARACTER IS",Ascii
70   PRINT String$
80   PRINT "THIS IS THE LAST LINE OF OUTPUT. "
90   END
```

Outputs:

```
THE LENGTH OF THE ASCII STRING IS 10
THE ASCII REPRESENTATION OF THE FIRST CHARACTER IS 0

THIS IS THE LAST LINE OF OUTPUT
```

Since the maximum length is stored as the current length, you can examine the contents of the buffer via the string name.

The main use of BUFFER is as a tool in input and output and is addressed in the I/O chapters.

An example of a common error is:

```
10   DIM Astring$[10]                !Astring cur len is 0
20   DIM String$[10] BUFFER          !String$ cur len is 10
30   ASSIGN @B TO BUFFER String$     !IO pathname is String$
40   String$="XYZ"                   !Put XYZ in String$
50   PRINT LEN(String$)              !Print cur len of String$
60   PRINT LEN(Astring$)             !Print cur len of Astring$
70   ENTER @B USING "%,K";Astring$ !Put String$ into Astring$
80   PRINT LEN(Astring$)             !See how long Astring is now.
90   END
```

Outputs:

```
3
0
0
```

The information about where to access BUFFER data is kept in a separate register; therefore assignments into the BUFFER place the data but not the length of the I/O transfer. The data is there -the I/O processor has no way to see it.

Two brief examples follow showing some simple uses of a BUFFER variable to input from a device and to output to a device. The chapter on "Bit Manipulations" explains this use of BUFFER variables in much greater detail.

If you use the BUFFER variable for input from a device your program segment for input looks like:

```
 70  ASSIGN @A TO Device
 80  ASSIGN @B TO BUFFER
 90  TRANSFER @A TO @B
100  ENTER @B;String$
```

or if you change statements 90 and 100, your program segment looks like this for output:

```
 90  OUTPUT @B
100  TRANSFER @B TO @A
```

Other Memory Considerations

This section discusses other memory considerations.

Memory Restrictions

The BASIC Language System is not able to locate variables which start more than 2^{19} bytes past the beginning of a program or subprogram. This results in the following restrictions.

- When you declare local variables in the beginning of your program or subprogram, declare a large array last. For example:

  ```
  90  DIM A(65536)
  100  DIM B(10)
  ```

  results in BASIC ERROR 2 (Memory overflow) when you run your program and attempt to allocate the array B.

  ```
   90  DIM B(10)
  100  DIM A(65536)
  ```

  results in no error.
- When you use the ALLOCATE statement, you are limited to 2^{19} bytes for these variables.
- When you use COM, you are limited to 2^{19} bytes for labelled COM variables; however, unlabelled COM variables are only limited by the system's memory.

Subprograms

Each subprogram sets up its own local environment; declarations, assignment statements, statement labels, and so on. Storage of local variables is temporary, and is returned to the main user Read/Write memory upon return to the calling program. Excluding the COM area, the only way for the main program to share variables with its subprograms is through the argument lists, called parameters.

A subprogram is not the same as a plain "subroutine", which is invoked with a GOSUB statement. A subroutine is within the current program context and accesses the enclosing context's variables. CALLs to subprograms cause program execution to be slower than if GOSUBs are used. In situations where the separate environment of a subprogram is not needed and speed is important to you, use GOSUB and a subroutine.

Recursion is the ability of a subprogram to call itself. A copy of the new local variables is dynamically generated each time a recursive call is made. You are limited by the available memory in the number of calls you may make. Recursion is a powerful tool which simplifies iterative program coding and, in many cases, decreases program execution time.

The following recursive program calculates nine factorial.

```
 10   DOUBLE Number,Factorial
 20   Number=9
 30   Factorial=1
 40   CALL Recursive(Number,Factorial)
 50   PRINT Number;" factorial is";Factorial
 51   END
 60   SUB Recursive(DOUBLE Value,Result)
 80   IF Value>1 THEN  CALL Recursive(Value-1,Result)
 90   Result=Result*Value
100   PRINT "The factorial now is:";Result
110   SUBEND
```

Outputs:

```
The factorial now is:  1
The factorial now is:  2
The factorial now is:  6
The factorial now is:  24
The factorial now is:  120
The factorial now is:  720
The factorial now is:  5040
The factorial now is:  40320
The factorial now is:  362880
```

See the "Subprograms" chapter for more information about subprograms.

Array Considerations

Some of the advantages in storing information in arrays are the following.

- System functions enable you to perform operations on arrays easily and efficiently.
- Arrays allow grouping of many similar data items in one storage area.
- Program code is easier to read, write, alter and maintain. For example, an entire array can be read or printed with a single statement.

```
 80   READ Array(*)
250   PRINT Array(*)
```

A common error is to confuse the subscript of the array with the value stored in that array element. The memory cell's name of `Joy_stats(31,4,1)` is irrelevant to whatever value is stored in it; 11.8, 524.2, or any other value can be stored, as long as it is consistent with its type declaration.

Arrays require little effort to declare and manipulate, but they also require little effort to consume vast amounts of memory and to create magnificently subtle errors.

Since REAL arrays use two full words of storage for each element, SHORT and DOUBLE arrays each use one word per element, and INTEGER arrays use one-half of a word for each element:

- use the type of array requiring the least amount of memory consistent with the needs of your program;
- use ALLOCATE to reserve space for large temporary arrays as they are needed, and then free this space with the DEALLCOATE statement;
- use COM when you want several contexts, program and subprograms, to share the same data items.

Subscripts for multidimensional arrays must be used consistently. Attempting to refer to an array element outside of the declared dimension (subscript) range, for example in a FOR...NEXT loop, results in an error. Implicitly defining an array with subscripts outside of the default range of 0 or 1 to 10 also causes an error.

Chapter 11
Numeric Computations

This chapter covers two main topics:

1. mathematical operations;
2. intrinsic and trigonometric functions.

Mathematical Operations

Mathematical operations covers three areas:

1. arithmetic, relational, and logical operators;
2. evaluation hierarchy, the order in which mathematical expressions are evaluated;
3. real math error processing, what happens when you tell the computer to represent a number it is not able represent.

Operators

Numeric data is stored in the memory of the computer in four possible formats. Non-fractional numbers can be stored as either INTEGERs or DOUBLEs while fractional numbers can be stored as either SHORTs or REALs.

The following table illustrates the characteristics of each type.

Type	Bits Used for Storage	Significant Digits	Range
INTEGER	16	4 or 5	−32 768 thru 32 767
DOUBLE	32	9 or 10	−2 147 483 648 thru 2 147 483 647
SHORT	32	6 or 7	$\pm 1.2 \times 10^{-38}$ thru $\pm 3.4 \times 10^{38}$
REAL	64	16 or 17	$\pm 2.2 \times 10^{-308}$ thru $\pm 1.8 \times 10^{308}$

One numeric type is said to be higher than another if the range of magnitudes that can be represented in that type is larger. From highest precedence to lowest the types are REAL, SHORT, DOUBLE and INTEGER.

The term coercion is used when the value of an expression is changed from one variable type to another. The system can force coercion for incompatability reasons, or you can force the coercion by using functions such as FLT. For example, if the variable I is declared INTEGER and the variable R is declared REAL, the value of I is coerced to type REAL before the assignment R = I. Note that this coercion does not affect the value of the variable I. Similarly, in the assignment I = R, the value of R is coerced to type INTEGER before the assignment is executed.

When two operands with different types are combined using arithmetic operators, the type of the result is the higher of the types of the two operands.

There are two exceptions to this rule.

1. The "/" or division operator causes both operands to be coerced to REAL.
2. If both operands are of type SHORT, the result is coerced to type REAL.

Arithmetic Operators

The arithmetic operations that can be performed on your computer are shown in the following table.

Operator	Meaning
+	Addition
−	Subtraction
*	Multiplication
/	Division
^	Exponentiation
DIV	Truncated quotient of dividend and divisor
MOD	X MOD Y = X − Y × (X DIV Y)
MODULO	X MODULO Y = X − Y × INT(X/Y)

The following programs illustrate the functions of the arithmetic operators (note the difference between MOD and MODULO).

```
1    INTEGER A
2    DOUBLE B
3    SHORT C
4    REAL D,F
5    A=12
6    B=11
7    C=10.0
8    D=15.6
10   PRINT "A^2+B/C-D*A EQUALS",A^2+B/C-D*A
11   END
```

Outputs:

```
A^2+B/C-D*A EQUALS -42.1
```

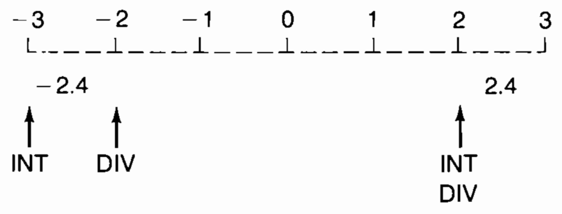
```
10   X1=12
11   X2=-12
20   Y=5
30   PRINT "X1 MOD Y EQUALS",X1 MOD Y
31   PRINT "X2 MOD Y EQUALS",X2 MOD Y
40   PRINT "X1 MODULO Y EQUALS",X1 MODULO Y
41   PRINT "X2 MODULO Y EQUALS",X2 MODULO Y
50   END
```

Outputs:

```
X1 MOD Y EQUALS          2
X2 MOD Y EQUALS         -2
X1 MODULO Y EQUALS       2
X2 MODULO Y EQUALS       3
```

The difference in the answers when using MOD or MODULO results because the DIV operator returns the truncated result of the division of X and Y (12 DIV 5 = 2, −12 DIV 5 = −2), while the INT operator returns the greatest integer which is less than or equal to the expression (INT(12/5) = 2, INT(−12/5) = −3).

Truncation means that the fractional part of a number is cut off or dropped. This chart illustrates the difference betwpen the truncated result of a number, returned by DIV, and the greatest integer less than the number, which INT returns.



The .4 is dropped in truncation resulting in either 2 or −2 as the result of the DIV function. The INT function returns the greatest integer which less than the result, 2 or −3.

Relational Operators

Relational operators determine the value relationship between two expressions. This can be especially useful for program branching if a specified condition is true.

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

The result of a relational operation is either a 1 (if the relation is true) or a 0 (if the relation is false).

Here are some examples of relational operations:

```
 1  INTEGER A,B,C
10  INPUT "ENTER THREE VALUES (A,B,&C)",A,B,C
30  IF (A*B)<=C THEN
40     PRINT A;"*";B;"<=";C;"IS TRUE."
50  ELSE
60     PRINT A;"*";B;"<=";C;"IS FALSE."
70  END IF
80  END
```

Entering the values 1, 2, 3 results in:

1 * 2 <= 3 IS TRUE.

The equals sign is also used in the assignment statement, where the variable is to the left of the equals sign and the value to be assigned is to the right. If the equals sign is used in such a way that it might be either an assignment or a relational operation, the computer assumes that it is an assignment operation. For example, X=Y=Z assigns the value of Z to Y and X. X=(Y=Z) assigns either a 0 or a 1 to X, depending on whether X and Y are equal or not equal.

Logical Operators

The logical operators, **AND**, **OR**, **EXOR**, and **NOT** are used for creating expressions that can have only two values, either 0 (false) or 1 (true). (Although true can be any nonzero value, 1 is used for simplicity.) These are called Boolean expressions, and are created by both relational as well as non-relational expressions.

If an expression is relational, for example A<B, its true or false designation is determined by the particular relational value. If A is 5 and B is 999, then A<B is true, and the value is 1. If an expression is non-relational, it is true if its arithmetic value is any value other than 0, and false if its arithmetic value equals 0.

The **AND** operator compares two expressions. If both expressions are true, the result is true (1). If one or both of the expressions is false, the result is false (0).

The **OR** operator compares two expressions, returning the value true (1) if one or both of the expressions is true, and false (0) if neither expression is true.

The **EXOR** (exclusive or) compares two expressions. If only one of the expressions is true, the result is true (1). If both are true, or both are false, the result is false (0).

The **NOT** returns the opposite of the logical value of an expression. If the expression is true, the result is false (0). If the expression is false, the result is true(1).

The following table illustrates the results returned by the logical operators comparing two expressions:

Operands		**Logical Operators**				
A	**B**	**A AND B**	**A OR B**	**A EXOR B**	**NOT A**	**NOT B**
0	0	0	0	0	1	1
0	1	0	1	1	1	0
1	0	0	1	1	0	1
1	1	1	1	0	0	0

Here is a program which uses logical operators.

```
 10   INTEGER A,B,C,D
 20   A=0
 30   B=2
 40   C=4
 50   D=4
 60   IF A<B AND C=D THEN
 70      PRINT "A is less than B      AND      C equals D"
 80   ELSE
 90      PRINT "The expression A<B AND C=D is false(0)."
100   END IF
110   !
120   IF A AND C=D THEN
130      PRINT "A AND C=D are both TRUE or are each equal to 1."
140   ELSE
150      PRINT "Either A OR/AND C=D has a value equal to 0."
160   END IF
170   !
180   IF A OR B THEN
190      PRINT "A OR/AND B are TRUE(1)."
200   ELSE
210      PRINT "Both A AND B are false(0)."
220   END IF
221   !
230   IF A EXOR B THEN
240      PRINT "One value is TRUE AND one value is FALSE."
250   ELSE
260      PRINT "A AND B are either both TRUE or both FALSE."
270   END IF
280   !
290   PRINT "NOT A equals";NOT (A)
300   PRINT "NOT B equals";NOT (B)
310   PRINT "NOT A OR NOT B equals";NOT (A) OR NOT (B)
320   END
```

Outputs:

```
A is less than B     AND     C equals D.
Either A OR/AND C=D has a value equal to 0.
A OR/AND B are TRUE(1).
One value is TRUE AND one value is FALSE.
NOT A EQUALS 1
NOT B EQUALS 0
NOT A OR NOT B EQUALS 1
```

Evaluation Hierarchy

When an expression has more than one operation, the order in which the computer performs the operations depends on the following hierarchy.

Highest Priority	( ) parenthesis
	Functions
	^ exponentiation
	*,/,MOD,MODULO,DIV
	+, −, unary +, unary −
	All relational operators
	NOT
	AND
Lowest Priority	OR,EXOR

An expression is scanned from left to right. Each operator is compared to the operator on its right. If the operator to the right has a higher priority, then that operator is compared to the next operator on its right. This continues until an operator of equal or lower priority is encountered, the highest priority operation, or the first of the two equal operations, is performed. Then any lower priority operations on the left are compared to the next operator to the right. This comparison continues until the entire expression is evaluated.

The order of evaluation for an expression is demonstrated by:

```
10  Answer=2+3*6/(7-4)^2
20  PRINT Answer
30  END
```

Outputs:

```
4
```

The order of evaluation for the expression is:

```
2+3*6/(7-4)^2  multiplication
2+18/(7-4)^2   evaluate parentheses
    2+18/3^2   exponentiation
      2+18/9   division
         2+2   addition
           4   result
```

Whenever you are in doubt as to the order of execution for any expression, use parentheses to indicate the order. Using parentheses for "implied" multiplication is not allowed; 2(5 – 1) must appear as 2*(5 – 1).

The sytem assigns a value to a variable **when** the variable is encountered in a line of program code. If the value of a variable changes within a line of code, the last value of the variable is the one used in subsequent operations involving that variable. An example demonstrating deferred variable binding (giving a value) follows.

```
10  A=10
20  Value=FNVarbind(A)+A
30  PRINT Value
40  END
50  DEF FNVarbind(A)
60  A=5
70  RETURN 20
80  FNEND
```

Outputs:

```
25
```

The value of A is changed from 10 to 5 in the function FNVarbind; this value of 5 is then added to the returned value of 20 to give a result of 25.

If line 20 is changed to:

```
20  Value=A+FNVarbind(A)
```

then Value equals 30, since A is originally 10 and is **then** added to the number the function Varbind returns, 20, producing 30 as the final result for Value.

This delay in the binding of variables may be different than it is in systems you have previously encountered.

Real Math Error Processing

When a SHORT or REAL math error occurs, your program normally pauses or goes to an error branching routine. The DEFAULT ON statement circumvents overflow and underflow real math errors, produced with improper arguments, by supplying default values. Integer math errors can still occur. The DEFAULT OFF statement enables normal math error processing.

```
10  DEFAULT ON          !Enables the math error defaults.
20  PRINT "  I","10/I"
30  FOR I=-5 TO 5
40     PRINT I,10/I
50  NEXT I
60  END
```

Outputs:

```
  I          10/I
 -5           -2
 -4           -2.5
 -3           -3.33333333333
 -2           -5
 -1           -10
  0           1.79769313486E+308
  1           10
  2           5
  3           3.33333333333
  4           2.5
  5           2
```

Note that when 10 is divided by 0 causing a REAL overflow error, the default value, MAXREAL, is returned. All digits of precision for MAXREAL are not printed because of the output format. Try running this program with DEFAULT OFF.

The default underflow for REAL variables is 0. If you use DEFAULT ON with SHORT variable types, the default for overflow is approximately 3.402E+38 and the default for underflow is 0.

Math Functions and Statements

This section covers functions, logarithmic and exponential functions, and trigonometric functions and statements. See the "Subprograms" chapter for a detailed discussion of function subprograms and single-line functions.

General Functions

ABS returns the absolute value of its argument. The result is the same type (INTEGER, DOUBLE, SHORT, REAL) as the argument.

```
10  INPUT "ENTER THE PRINTER SELECT CODE.",A
20  IF SGN(A)=-1 THEN
30     PRINT "SELECT CODE CANNOT BE NEGATIVE."
40     PRINT "THE ABSOLUTE VALUE ",ABS(A)," WILL BE USED."
41  ELSE
42     PRINT "YOU INPUT ",A,"AS YOUR SELECT CODE."
50  END IF
61  IF A>8 OR A<-8 THEN
62     PRINT "YOUR NUMBER IS TOO LARGE, TRY AGAIN."
63     GOTO 10
64  END IF
65  PRINTER IS ABS(A)
70  END
```

Outputs:

If you enter 6 as your select code:

```
YOU INPUT 6 AS YOUR SELECT CODE.
```

If you enter −6 as your select code:

```
SELECT CODE CANNOT BE NEGATIVE.
THE ABSOLUTE VALUE  6  WILL BE USED.
```

If you enter 55 as your select code:

```
YOU INPUT 55 AS YOUR SELECT CODE.
YOUR NUMBER IS TOO LARGE, TRY AGAIN.
```

DROUND rounds a numeric expression to the specified number of decimal digits. If the number of digits specified is less than or equal to 0, DROUND returns a 0. The results are approximate since the numbers are stored internally in the binary number system. DROUND is useful for checking equality to a specified number of digits or for standardizing internal value accuracy.

```
10  Num=12345.12345
20  PRINT Num,"rounded to two digits is",DROUND(Num,2)
30  PRINT Num,"rounded to four digits is",DROUND(Num,4)
40  PRINT Num,"rounded to seven digits is",DROUND(Num,7)
50  PRINT Num,"rounded to -1 digits is",DROUND(Num,-1)
60  END
```

Outputs:

```
12345.12345          rounded to two digits is    12000
12345.12345          rounded to four digits is   12350
12345.12345          rounded to seven digits is  12345.12
12345.12345          rounded to -1 digits is     0
```

FLT returns the value of the argument in REAL precision. If you multiply a SHORT number by a DOUBLE number, the DOUBLE is coerced to SHORT and precision may be lost. Although the result is then coerced to REAL, the number of significant digits is less than that for the original DOUBLE number.

```
10   INTEGER A
20   SHORT B
21   A=1234
22   B=2.101
30   C=FLT(A)*B
40   PRINT "A EQUALS",A,"B EQUALS",B,"C EQUALS",C
50   END
```

Outputs:

```
A EQUALS   1234   B EQUALS    2.101   C EQUALS 2592.63408709
```

FRACT returns the value `expression - INT(expression)` For a positive fractional number FRACT returns the part to the right of the decimal point. For a negative fractional number, FRACT returns the difference between the expression and the value of the nearest integer which is less than the expression. The example following INT shows the relationship of FRACT and INT.

INT returns the greatest integer which is less than or equal to the expression. The result is the same type as the argument.

```
 10   Pos_num=12.6
 20   Neg_num=-12.6
 30   Value1=INT(Pos_num)
 40   Value2=INT(Neg_num)
 50   Value3=FRACT(Pos_num)
 60   Value4=FRACT(Neg_num)
 70   PRINT "Using the INT function with",Pos_num,"equals",Value1
 80   PRINT "Using the INT function with",Neg_num,"equals",Value2
 90   PRINT "Using the FRACT function with",Pos_num,"equals",Value3
100   PRINT "Using the FRACT function with",Neg_num,"equals",Value4
110   END
```

Outputs:

```
Using the INT function with 12.6 equals      12
Using the INT function with -12.6 equals    -13
Using the FRACT function with 12.6 equals    .6
Using the FRACT function with -12.6 equals   .4
```

Note the impact of the FRACT function on negative numbers. INT(12.6) equals 12, but INT(− 12.6) equals − 13. FRACT(12.6) equals 12.6 − 12 or .6; while FRACT(− 12.6) equals − 12.6 − (− 13) or .4.

MAX	returns the greatest value in a list of numeric expressions and arrays.
MAXREAL	returns the largest positive REAL number available in range of the computer(1.7976931348623157E + 308).
MIN	returns the smallest value in a list of numeric expressions and arrays.
MINREAL	returns the smallest positive REAL number available in the range of the computer(2.2250738585072014E − 308).

```
 1  STANDARD
10  INPUT "INPUTFOUR VALUES",A,B,C,D
20  PRINT "FOUR VALUES:",A,B,C,D
30  PRINT "MAXIMUM:",MAX(A,B,C,D)
40  PRINT "MINIMUM:",MIN(A,B,C,D)
42  IMAGE K,X,D.20DESZZZ
50  PRINT USING 42;"MAXREAL EQUALS",MAXREAL
60  PRINT USING 42;"MINREAL EQUALS",MINREAL
70  END
```

Outputs:

```
FOUR VALUES     1     2     3     4
MAXIMUM         4
MINIMUM         1
MAXREAL EQUALS  1.79769313486231500000E+308
MINREAL EQUALS  2.22507385850720100000E-308
```

Line 42 is an advanced I/O concept and was used only to output all digits stored in the computer of MAXREAL and MINREAL. Without this line only 15 significant digits are printed. An explanation can be found in "The Image Specifier" chapter.

The following program illustrates putting MAXREAL and MINREAL into a SHORT variable.

```
10  DEFAULT ON
11  SHORT Maxshort,Minshort
20  Maxshort=MAXREAL
30  PRINT "The SHORT equivalent of MAXREAL is:";Maxshort
40  Minshort=MINREAL
50  PRINT "The SHORT equivalent of MINREAL is:";Minshort
60  END
```

Outputs:

```
The SHORT equivalent of MAXREAL is: 3.402823E+38
The SHORT equivalent of MINREAL is: 0
```

PI returns an approximate value of π(3.1415926537931).

PROUND rounds a numeric expression to the digit position corresponding to the specified power of ten. The results are approximate since the numbers are stored internally in the binary number system. Specifying −2 is useful for output of monetary values.

```
10  REAL A
11  A=454.789
20  PRINT "ORIGINAL NUMBER:",A
30  FOR I=-2 TO 3
40     PRINT "POWER OF 10:",I,PROUND(A,I)
50  NEXT I
60  END
```

Outputs:

```
ORIGINAL NUMBER:               454.789
POWER OF 10:      -2           454.79
POWER OF 10:      -1           454.8
POWER OF 10:       0           455
POWER OF 10:       1           450
POWER OF 10:       2           500
POWER OF 10:       3           0
```

RANDOMIZE specifies a seed for the RND function. If no expression is specified, the computer arbitrarily choses a uniform seed based on the clock.

```
RANDOMIZE(TIMEDATE/(TIMEDATE+PI))
```

RND returns a pseudo-random number greater than 0 and less than 1. The random number returned is based on a seed set to 37 480 660 at power-on, SCRATCH A, SCRATCH P, SCRATCH or program prerun. Each succeeding use of RND returns a random number which uses a number proportional to the previous random number as a seed. You can modify the seed with the RANDOMIZE statement. If you want the same set of random numbers every time you run your program use RND without RANDOMIZE or initialize by using RANDOMIZE with a constant value expression. If you want different data each time you run your program, use RANDOMIZE without an expression and let the system set the seed according to the clock.

```
10  PRINT RND
20  RANDOMIZE
30  PRINT RND
40  END
```

Output when run on 7 October 1982 18:18:14:

```
1.74532923929E-02
.467362898154
```

SGN returns 1 if the argument is positve, 0 if it equals 0, and −1 if it is negative. The value returned is a DOUBLE.

SQR returns the REAL square root of a non-negative argument.

```
10  INPUT "INPUT THE FIRST SIDE OF YOUR TRIANGLE ",A
30  IF SGN(A)=0 THEN
40     PRINT "YOU INPUT A NEGATIVE NUMBER FOR SIDE ONE"
41     GOTO 10
42  ELSE
43     INPUT "INPUT THE SECOND SIDE OF YOUR TRIANGLE ",B
44     IF SGN(B)<=0 THEN
45        PRINT "YOU INPUT AN ERRONEOUS VALUE FOR SIDE TWO."
46        GOTO 43
47     ELSE
48        Hypotenuse=SQR(A^2+B^2)
49        PRINT "THE SIDES YOU ENTERED ARE",A,B
51        PRINT "THE HYPOTENUSE EQUALS ",Hypotenuse
52     END IF
53  END IF
62  END
```

If you enter 24 and 45 your output is:

```
THE SIDES YOU ENTERED ARE    24    45
THE HYPOTENUSE EQUALS        51
```

Logarithmic and Exponential Functions

These three functions return REAL arguments.

EXP The exponential function returns the value of the constant Naperian **e** (≈2.718 281 828 495 05) raised to the power of the computed expression. This is the inverse of the **LOG** function.

LGT The common log function returns the logarithm (base 10) of a positive-valued expression.

LOG The natural log function returns the logarithm (base **e**)of a positive-valued expression.

```
10  A=2
20  B=4
30  PRINT "NUMBER";TAB(15),"EXP";TAB(35),"LGT";TAB(50),"LOG"
40  PRINT A;TAB(10),EXP(A);TAB(27),LGT(A);TAB(44),LOG(A)
50  PRINT B;TAB(10),EXP(B);TAB(27),LGT(B);TAB(44),LOG(B)
60  END
```

Outputs:

```
NUMBER         EXP                  LGT               LOG
2         7.38905609893      .301029995664     .69314718056
4         54.5981500331      .602059991328     1.38629436112
```

Trigonometric Functions and Statements

The trigonometric functions use the angular unit mode, degrees radians, or grads, which is currently set. A trigonometric statement is used to set the angular unit mode. If no angle mode is specified in a program, the default is radians.

Statements

To set the **degree mode** use the DEG statement. There are 360 degrees in a full circle.

To set the **grad mode** use the GRAD statement. There are 400 grads in a full circle.

To set the **rad mode** use the RAD statement. There are 2π radians in a full circle.

Functions

These functions return REAL values:

ACS	The arccosine is the principal value of the angle which has a cosine equal to the argument. It is the inverse of COS. The argument expression must be in the range from -1 to $+1$.
ASN	The arcsine is the principal value of the angle which has a sine equal to the argument. It is the inverse of SIN. The argument expression must be in the range from -1 to $+1$.
ATN	The arctangent is the principal value of the angle which has a tangent equal to the argument. It is the inverse of the TAN function. The argument expression must be in the range of REAL.
COS	The cosine function returns the cosine of the argument. The argument must have an absolute value less than 2.981 E+8 radians. The value returned is in the range from -1.0 to $+1.0$.
SIN	The sine function returns the sine of the argument. The argument must have an absolute value less than 2.981 E+8 radians. The value returned is in the range from -1.0 to $+1.0$.
TAN	The tangent function returns the tangent of the argument. The argument must have an absolute value less than 1.490 E+8 radians. The value returned is in the range from -1.0 to $+1.0$.

```
  1  Angle=30
 10  PRINT "THE ORIGINAL ANGLE IS",Angle
 20  FIXED 4
 30  PRINT SPA(22);"SIN";SPA(7),"COS";SPA(7),"TAN";SPA(7),"ASN"
 40  FOR I=1 TO 3
 50     ON I GOSUB Degrees,Radians,Grads
 60     PRINT SIN(Angle);TAB(30),COS(Angle);TAB(40),TAN(Angle);TAB(50);
 61     S=SIN(Angle)
 62     PRINT ASN(S)
 70  NEXT I
 71  STOP
 80 Degrees:  DEG       !Set DEGree mode.
 90  PRINT "DEGREES:",
100  RETURN
110 Radians:  RAD       !Set RADian mode
120  PRINT "RADIANS:",
130  RETURN
140 Grads:    GRAD      !Set GRAD mode.
150  PRINT "GRADS:",
160  RETURN
170  END
```

Outputs:

```
        THE ORIGINAL ANGLE IS  30
                 SIN         COS        TAN       ASN
 DEGREES:     .5000       .8660      .5774    30.0000
 RADIANS:    -.9880       .1543    -6.4053   -1.4159
 GRADS:       .4540       .8910      .5095    30.0000
```

Chapter 12
String Operations

There are a variety of operators and functions which you can use to manipulate strings. This chapter covers these operations.

To perform this operation:	Use this function:
Concatenate two strings.	&
Specify a sorting order.	LEXICAL ORDER IS
Find the current length of a string.	LEN
Find the position of one string within another string.	POS
Delete all leading and trailing blanks.	POS$
Reverse the order of the characters within a string.	REV$
Repeatedly concatenate a string of characters.	RPT$
Make all characters in the string lowercase.	LWC$
Make all characters in the string uppercase.	UPC$
Convert a number (character code) to its string equivalent.	CHR$
Convert a string numberic to its number equivalent.	NUM
Convert a number to a string.	VAL$
Convert a string to a number.	VAL
Convert an integer to a string.	IVAL$
Convert a string to an integer.	IVAL
Convert a DOUBLE number to a string.	DVAL$
Convert a string to a double number.	DVAL

Working with Strings

You can split and combine strings to create new ones using the string concatenation operator and substring notation.

String Concatenation

The string concatenation operator, '&', joins two strings together. Its result is frequently assigned to a third string. An error occurs if the concatenation operation produces a string that is longer than the dimensioned length of the third string being assigned. The following programs combine strings.

```
 10  DATA "ADAM ", "BRAVO ", "CHARLEY ", "XRAY ", "YANKEE ", "ZEBRA ", "ONE "
, "TWO ", "THREE ", "FOUR ", "FIVE ", "SIX "
 20  RANDOMIZE
 30  X=RND*10E10 MODULO 12
 40  RESTORE 10
 50  FOR I=1 TO X
 60    READ String_1$
 70  NEXT I
 80  RANDOMIZE X
 90  X=RND*10E10 MODULO 12
100  RESTORE 10
110  FOR I=1 TO X
120    READ String_2$
130  NEXT I
140  String_3$=String_1$&String_2$
150  PRINT "The current password is:  "&String_3$
160  END
```

Typical Output:

```
    The current password is:  BRAVO THREE
    The current password is:  XRAY XRAY
    The current password is:  ONE CHARLEY
```

Substrings

A substring is any part of a string which is made up of zero or more contiguous characters. You can use substrings to manipulate or look at only part of a string.

A substring specifier in an expression is different than a substring specifier in a destination or assignment statement. For example in the string, `B$=A$&"TION"`, `B$` is a destination string and `A$` is a string in an expression, also known as a source string.

A substring is specified by placing substring specifiers in brackets after the string name. There are three forms a substring can have:

- `String$[`first character position`]`

  The first character position is a numeric expression which is rounded to an integer. The substring is made up of the character corresponding to "first character position" thru the last character where "last character" is defined as follows.

String Type	Definition of "last character"
source	implicit last character position is current last character
destination	implicit last character position is the last position required by the length of the string being assigned the result, but no greater than the dimensioned length of the source string.

  For both strings, the first character position can be no greater than current length plus one.

- `String$[`first character position, last character position`]`

  This type of substring includes the beginning and ending characters and all in between. The last character must also meet the following criteria.

String Type	Definition of "last character"
source	last character position can be no greater than current last character
destination	last character position can be no greater than dimensioned last character.

  For both strings, the first character position can be no greater than current length plus one.

- `String$[`first character position; number of characters`]`

  The substring begins with the specified first character in the string and is the specified length. The number of characters specified cannot exceed the current length, plus one, minus the beginning character position.

  Note that when a comma separates the qualifiers, the latter number is taken to be the ending character postion; whereas, when a semicolon is the separator, the latter number specifies the length of the substring.

Assignment results also differ among the specification types. For example `A$` and `A$[first]` terminate the resulting string after the last character in the source. If the source is greater than the destination `A$[first,last]` and `A$[first;length]` fill the gap with blanks.

Here are some examples of these three forms of a substring.

```
 10  DIM String$[26]
 20  String$="AbCdEfGhIJKlMnOpQrStUvWxYz"
 30  PRINT String$
 40  PRINT String$[1,7]
 50  PRINT String$[9,15]
 60  PRINT String$[1;7]
 70  PRINT String$[1;13]
 80  PRINT String$[8,8]
 90  PRINT String$[8;8]
100  PRINT String$[8;1]
110  END
```

Outputs:

```
AbCdEfGhIJKlMnOpQrStUvWxYz
AbCdEfG
IJKlMnO
AbCdEfG
AbCdEfGhIJKlM
h
hIJKlMnO
h
```

In this case `String$` was implicitly dimensioned to have 18 characters, so any operation, such as `String$[15;6]`, which results in a string longer than that produces BASIC ERROR 18 - "Substring out of range or bad string length." `String$[15,6]` always produces an error since the ending character position is smaller than the beginning character position minus one.

String Expression Evaluation

The relational operators used for numeric computation can also be used for the evaluation of string expressions. These compare the sequence number for each character, the one with the lesser value is smaller in the relational test. The **sequence number** is the place in the "alphabet" where that character occurred.

For example, the sequence number for A, 65, is less than the number for B, 66, so "A"<"B" is true. You can use one of the standard sequence orders or create your own. "Lexical order" is the name for this ordering sequence.

Relational Operators

Strings are compared, character by character, from left to right until a difference is found. If one string ends before a difference is found, the shorter string is considered the lesser. For example, "Kath" is smaller than both "Kath " and "Katherine".

Any one of these relational operators may be used to compare strings:

=	equal to	<>	not equal to
<	less than	>	greater than
<=	less than or equal to	>=	greater than or equal to

An example of a program using relational operators follows.

```
 10  LOOP
 20    INPUT "WHAT IS THE FIRST STRING?",String1$
 30    INPUT "WHAT IS THE SECOND STRING?",String2$
 40    SELECT String1$
 50    CASE <String2$
 60      PRINT String1$&" IS LESS THAN "&String2$
 70    CASE >String2$
 80      PRINT String1$&" IS GREATER THAN "&String2$
 90    CASE =String2$
100      PRINT String1$&" IS THE SAME AS "&String2$
110    END SELECT
120  END LOOP
130  END
```

Lexical Order

Each character must be transferred into the computer in a coded form, for example, "H"=01 001 000, "I" = 01 001 001, "{" = 01 010 011. Notice that each character uses eight bits for its coding. Your computer uses the ASCII coding scheme.

Lexical order sets the collating order of the ASCII codes used to represent characters. The sequence number is the ordering or ranking of these ASCII codes. The code for the character "h" is given a sequence number that is smaller than the sequence number for the code for "i". This is called the lexical order of the code. Relational tests use this lexical ordering to perform their testing operations. An explanation of the more complex features of the non-ASCII lexical orders is in the BASIC Language Reference.

The statement "LEXICAL ORDER IS" assigns one of several possible lexical orders which are used for sorting, lexical case functions and string relational operators.

Since many countries have their own character sets, these statements provide convenient methods to specify different lexical orders:

```
LEXICAL ORDER IS ASCII
LEXICAL ORDER IS STANDARD
LEXICAL ORDER IS FRENCH
LEXICAL ORDER IS GERMAN
LEXICAL ORDER IS KATAKANA
LEXICAL ORDER IS SPANISH
LEXICAL ORDER IS SWEDISH
```

Although the character code for each character remains the same in different lexical orders, the sequence numbers do not necessarily have to be the same. For example:

Character	**Char Code**	**Sequence Numbers**			
		ASCII	**FRENCH**	**GERMAN**	**SPANISH**
ā	178	178	97	119	148

When STANDARD is specified, the collating sequence and upper and lowercase mapping is that associated with the keyboard option on your machine. When FRENCH, GERMAN, SPANISH, or SWEDISH is specified, the lexical order is the one appropriate to the corresponding language and alphabet. If either ASCII or KATAKANA is specified, or STANDARD in English or Katakana machines, the lexical order is the ASCII character set. STANDARD is the default power on or reset case.

You can also create your own lexical order by executing:

```
LEXICAL ORDER IS Table(*)
```

When a table is specified, the user must provide the necessary collating sequence and can optionally provide a upper/lowercase mapping. Table must be a one dimensional, INTEGER array specifier with a capacity no less than 257 and no greater than 593.

The tables of lexical orders and a detailed method of creating your own table are in the BASIC Language Reference.

String Functions

Several intrinsic functions are provided by BASIC to let you manipulate a string. They are especially useful in text processing applications.

String Operations

There are several functions which perform operations on their string arguments.

LEN returns the current number of characters in the argument. At the time of declaration, the current length is 0 unless the string is declared to be a BUFFER, in which case it is set to the maximum length of the string.

```
 10   DIM String2$[10]
 20   DIM String3$[10] BUFFER
 30   Num1=LEN(String1$)
 40   Num2=LEN(String2$)
 50   Num3=LEN(String3$)
 60   PRINT "Implicitly declared String1$ has a current length of ",Num1
 70   PRINT "Explicitly declared String2$ has a current length of ",Num2
 80   PRINT "Explicitly declared with BUFFER parameter String3$"
 90   PRINT "    has a current length of            ",Num3
100   String1$="Anything"
110   String2$=" can"
120   String3$=" happen."
130   PRINT "String1$ now has a current length of",LEN(String1$)
140   PRINT "String2$ now has a current length of",LEN(String2$)
150   PRINT "String3$ now has a current length of",LEN(String3$)
160   END
```

Outputs:

```
Implicitly declared String1$ has a current length of 0
Explicitly declared String2$ has a current length of 0
Explicitly declared with BUFFER parameter String3$
   has a current length of 10
String1$ now has a current length of 8
String2$ now has a current length of 4
String3$ now has a current length of 8
```

POS returns the position of the first occurrence of a substring, within another string. If the substring is not found or if you attempt to find the "null" substring, a zero is returned.

```
 10   DIM String$[40]
 20   INTEGER Position
 30   String$="ONE TWO THREE FOUR FIVE SIX"
 40   Sub_string$="THREE"
 50   Position=POS(String$,Sub_string$)
 60   IF Position THEN
 70      PRINT "THREE WAS FOUND AT POSITION ",Position
 80   ELSE
 90      PRINT "THREE WAS NOT FOUND."
100   END IF
110   END
```

Outputs:

```
THREE WAS FOUND AT POSITION  9
```

TRIM$ returns a string with all leading and trailing blanks removed. Internal blanks are not affected.

```
10  String$="     STRING     "
20  New_string$=TRIM$(String$)
30  PRINT "("&String$&")"
40  PRINT "("&New_string$&")"
50  END
```

Outputs:

```
(     STRING     )
(STRING)
```

REV$ returns a string with the characters in the reverse order of the characters in the argument string.

```
10  String1$="0123456789"
20  String2$=REV$(String1$)
30  PRINT String1$
40  PRINT String2$
50  END
```

Outputs:

```
0123456789
9876543210
```

RPT$ repeatedly concatenates a string of characters the number of times specified.

```
10  DIM Line$[60]
20  Line$=RPT$("#",60)
30  PRINT Line$
40  END
```

Outputs:

```
 ############################################################
```

LWC$ returns a string in which all uppercase characters are converted to lowercase characters. The current lexical order is used.

```
10  Cap_string$="HOW Are YOU?"
30  PRINT "Original string was ";Cap_string$
40  PRINT "After LWC$ it became ";LWC$(Cap_string$)
50  END
```

Outputs:

```
Original string was   HOW Are YOU?
After LWC$ it became  how are you?
```

UPC$ returns a string in which all lowercase characters are converted to uppercase characters. The current lexical order is used.

```
10   Sm_string$="how ARE yOU?"
20   Cap_string$=UPC$(Sm_string$)
30   PRINT "Original string was";Sm_string$
40   PRINT "After UPC$ it became";Cap_string$
50   END
```

Outputs:

```
Original string was   how ARE yOU?
After UPC$ it became  HOW ARE YOU?
```

String-Numeric Conversions

There is both a numeric 1, and a character "1". The string-numeric conversion functions take advantage of this dual representation.

The system scans the argument for each function from left to right until it comes to an invalid or nonlegal character. The valid characters are given in the following table.

Base	Valid Characters
2	0,1
8	0-7
10	0-9, optional + − first character[1]
16	0-9, A-F, a-f

In general the system expects well formed numbers. When a character is encountered which is not valid, the scanning stops and the value of the valid character(s) is returned. If no valid character is found, an error occurs.

A convenient way to accept any keyboard character as valid input is to look at all incoming data as character variables and then to convert the characters into their equivalent numeric constants, using one of the string-to-numeric conversion functions.

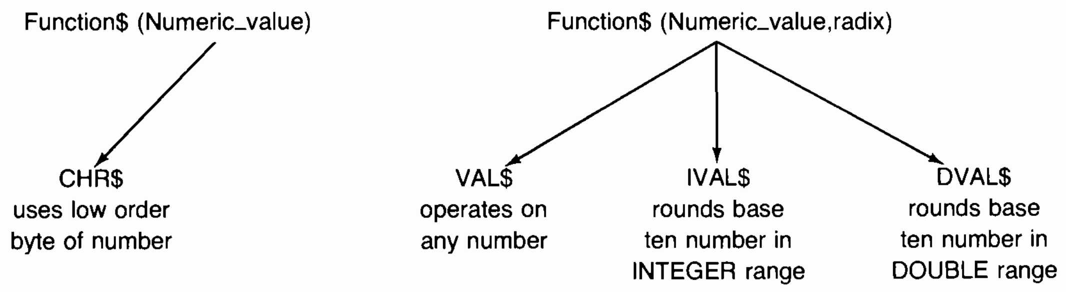
A few of the reasons for doing numeric-to-string conversions are:

- to make the numeric constant fit into a string array;
- for ease in outputting numbers without leading blanks;
- to make it simpler to pick out a specific digit in a number.

1 The characters e, E, D and L are valid for the VAL function only.

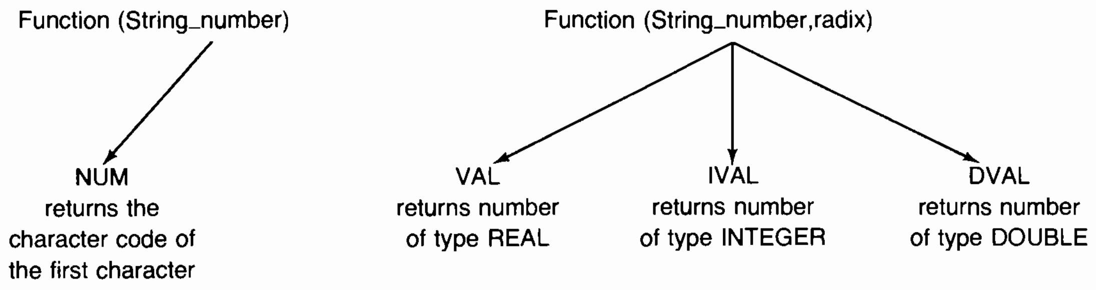
The following chart gives an overview of the string-numeric conversions.

Numeric-to-String Conversions


Function$ (Numeric_value)
Function$ (Numeric_value,radix)
CHR$ uses low order byte of number
VAL$ operates on any number
IVAL$ rounds base ten number in INTEGER range
DVAL$ rounds base ten number in DOUBLE range

Note that the numeric argument is always in base ten or decimal form. The radix refers to the string returned.

String-to-Numeric Conversions


Function (String_number)
Function (String_number,radix)
NUM returns the character code of the first character
VAL returns number of type REAL
IVAL returns number of type INTEGER
DVAL returns number of type DOUBLE

Note that the radix again refers to the string, however the string is now the argument passed to the function.

The CHR$ function converts a number, the character code, into the equivalent string character. The value is rounded and a MODULO 256 is performed. The low order byte of the 32-bit integer thus obtained is used; the high order bytes are ignored.

The NUM function converts the first character in the argument to its equivalent character code. The number returned is in the range from 0 to 255.

The VAL function converts a string expression into a numeric value. Embedded blanks are ignored; VAL("15 16 17") is 151617, not 15. The first nonblank character in the string must be a valid character for a numeric constant. The result is a REAL number.

The VAL$ function returns a string representation of the value of the argument. The returned string is in the current print format mode (FIXED, FLOAT or STANDARD). The string contains the same characters (digits) that appear when the number is printed.

IVAL returns the integer value of a numeric string expression. There must be no blanks anywhere in the string expression and it must evaluate to the ASCII representation of a binary, octal, decimal, or hexadecimal integer in the range of INTEGER.

The IVAL$ does the opposite of IVAL; IVAL$ returns the ASCII string representation of a base 10 argument in the range of INTEGER. The string is expressed in radix base binary, octal, decimal, or hexadecimal.

DVAL does much what IVAL does except that DVAL returns an integer in the range of DOUBLE. Again there must be no blanks anywhere and the string expression must evaluate to the ASCII representation of a binary, octal, decimal or hexadecimal integer in the range of DOUBLE.

DVAL$ is the opposite of DVAL and like IVAL$, but with a DOUBLE range. This function returns the ASCII string representation of a base 10 argument in the range of DOUBLE. The string is expressed in one of four radices: binary, octal, decimal, or hexadecimal.

The differences and similarities of the above functions when operating on their respective argument types may be seen in the following programs.

Numeric-to-string

```
 10  DIM Dval_result2$[64]
 20  Argument1=123.
 30  Argument2=32569
 40  Chr_result$=CHR$(Argument1)
 50  PRINT " Chr_result$ equals ",Chr_result$
 60  !
 70  Val_result1$=VAL$(Argument1)
 80  PRINT "Val_result1$ on argument1 equals ",Val_result1$
 90  !
100  Val_result2$=VAL$(Argument2)
110  PRINT "Val_result2$ on argument2 equals ",Val_result2$
120  !
130  Ival_result2$=IVAL$(Argument1,2)
140  PRINT "Ival_result2$ in base 2 equals ",Ival_result2$
150  !
160  Ival_result8$=IVAL$(Argument1,8)
170  PRINT "Ival_result8$ in base 8 equals ",Ival_result8$
180  !
190  Ival_result10$=IVAL$(Argument1,10)
200  PRINT "Ival_result10$ in base 10 equals ",Ival_result10$
210  !
220  Ival_result16$=IVAL$(Argument1,16)
230  PRINT "Ival_result16$ in base 16 equals ",Ival_result16$
240  !
250  ! Now look at the DOUBLE version.
260  Dval_result2$=DVAL$(Argument2,2)
270  PRINT "Dval_result2$ in base 2 equals ",Dval_result2$
280  !
290  Dval_result8$=DVAL$(Argument2,8)
300  PRINT "Dval_result8$ in base 8 equals ",Dval_result8$
310  !
320  Dval_result10$=DVAL$(Argument2,10)
```

```
330  PRINT "Dval_result10$ in base 10 equals ",Dval_result10$
340  !
350  Dval_result16$=DVAL$(Argument2,16)
360  PRINT "Dval_result16$ in base 16 equals ",Dval_result16$
370  END
```

String-to-numeric

```
 20  Argument1$="110"
 30  Argument2$="110111"
 40  Num_result=NUM(Argument1$)
 50  PRINT " Num_result equals ",Num_result
 60  !
 70  Val_result1=VAL(Argument1$)
 80  PRINT "Val_result1 on argument1$ equals ",Val_result1
 90  !
100  Val_result2=VAL(Argument2$)
110  PRINT "Val_result2 on argument2$ equals ",Val_result2
120  !
130  Ival_result2=IVAL(Argument1$,2)
140  PRINT "Ival_result2 in base 2 equals ",Ival_result2
150  !
160  Ival_result8=IVAL(Argument1$,8)
170  PRINT "Ival_result8 in base 8 equals ",Ival_result8
180  !
190  Ival_result10=IVAL(Argument1$,10)
200  PRINT "Ival_result10 in base 10 equals ",Ival_result10
210  !
220  Ival_result16=IVAL(Argument1$,16)
230  PRINT "Ival_result16 in base 16 equals ",Ival_result16
240  !
250  ! Now look at the DOUBLE version.
260  Dval_result2=DVAL(Argument2$,2)
270  PRINT "Dval_result2 in base 2 equals ",Dval_result2
280  !
290  Dval_result8=DVAL(Argument2$,8)
300  PRINT "Dval_result8 in base 8 equals ",Dval_result8
310  !
320  Dval_result10=DVAL(Argument2$,10)
330  PRINT "Dval_result10 in base 10 equals ",Dval_result10
340  !
350  Dval_result16=DVAL(Argument2$,16)
360  PRINT "Dval_result16 in base 16 equals ",Dval_result16
370  END
```

Chapter 13
Array Operations

This chapter discusses various operations you can perform on arrays.

Array Functions and Statements

Your BASIC system includes many functions to determine the characteristics of an array or evaluate each element of an operand array.

Characteristic Functions

These three functions tell you the size of any array.

BASE returns the lower bound for a valid subscript in the specified dimension of the array. The dimensions are numbered from left to right. For example:

```
DIM Array(2:4,8:24)      !BASE returns the lower
Base=BASE(Array,2)     !bound of the specified
DISP Base                !dimension

8
```

RANK returns the number of dimensions (1 to 6) in the array. For example:

```
DIM Array(2:4,8:24)
Rank=RANK(Array)      !RANK returns the number
DISP Rank             !of dimensions in an array

2
```

SIZE returns the number of elements in a specified dimension of an array. The dimensions are numbered from left to right. For example:

```
DIM Array(2:4,8:24)
Size=SIZE(Array,2)      !Size returns the number
DISP Size               !of elements in specified
                        !dimension
17
```

The following program uses BASE to determine the starting subscript for the array in the FOR...NEXT loop and SIZE to determine the ending subscript.

```
 10  OPTION BASE 1
 20  INTEGER Array(5,5)
 30  MAT Array=(6)
 40  PRINT
 50  PRINT "THE ORIGINAL ARRAY IS:"
 60  CALL Printer(Array(*))
 70  !
 80  REDIM Array(2,3)
 90  MAT Array=(3)
100  PRINT
110  PRINT "THE REDIMENSIONED 2 BY 3 ARRAY IS:"
120  CALL Printer(Array(*))
130  !
140  REDIM Array(5,5)
150  PRINT
160  PRINT "THE ORIGINAL ARRAY NOW IS:"
170  CALL Printer(Array(*))
180  PRINT "NOTE WHICH ELEMENTS CHANGED."
190  END
200  SUB Printer(INTEGER Matrix(*))
210    DOUBLE I,J,Row_min,Row_max,Col_min,Col_max
220    IF RANK(Matrix)<>2 THEN RETURN       !Only handle 2-D arrays
230    Row_min=BASE(Matrix,1)               !Lower bound rows
240    Row_max=Row_min+SIZE(Matrix,1)-1     !Upper bound rows
250    Col_min=BASE(Matrix,2)               !Lower bound columns
260    Col_max=Col_min+SIZE(Matrix,2)-1     !Upper bound columns
270    FOR I=Row_min TO Row_max
280      FOR J=Col_min TO Col_max
290        PRINT "       ";Matrix(I,J);
300      NEXT J
310      PRINT
320    NEXT I
330    PRINT
340  SUBEND
```

Outputs:

```
THE ORIGINAL ARRAY IS:
        6        6        6        6        6
        6        6        6        6        6
        6        6        6        6        6
        6        6        6        6        6
        6        6        6        6        6
THE REDIMENSIONED ARRAY IS:
        3        3        3
        3        3        3
THE ORIGINAL ARRAY NOW IS:
        3        3        3        3        3
        3        6        6        6        6
        6        6        6        6        6
        6        6        6        6        6
        6        6        6        6        6
NOTE WHICH ELEMENTS CHANGED.
```

Array Adding Functions

SUM	returns the sum of all elements in an array. The summing operation is performed in REAL precision, independent of the type of the array. The returned value is REAL.
MAT...CSUM	returns the sum of all elements in each column of a matrix. The sum of the elements in the first column of the operand matrix is stored in the first element of the resultant vector, and so on. The first dimension of a matrix represents the row, the second represents the column. The result vector is redimensioned, if necessary, to the number of columns in the operand matrix.

MAT...RSUM returns the sum of all elements in each row of a matrix. The sum of the elements in the first row of the operand matrix is stored in the first element of the result vector, and so on. The result vector is redimensioned if necessary to the number of rows in the operand matrix. The following program illustrates the use of SUM, MAT...CSUM and MAT...RSUM.

```
 10  OPTION BASE 1
 20  SHORT Jog_stats(6,2,2),Matrix(6,2),Columns(2),Rows(6)
 30  DIM Name$(6)[5]
 40  READ Name$(*)
 50  DATA "HENRY","JOE","MARCY","SARAH","TOM","WENDY"
 60  READ Jog_stats(*)
 70  DATA 8.4, 71.2, 9.8, 82.4        !DATA FOR HENRY
 80  DATA 5.2, 45.6, 4.8, 41.3        !DATA FOR JOE
 90  DATA 6.3, 48.1, 6.3, 47.2        !DATA FOR MARCY
100  DATA 2.1, 23.4, 2.2, 22.3        !DATA FOR SARAH
110  DATA 6.2, 36.3, 6.3, 35.4        !DATA FOR TOM
120  DATA 4.7, 41.3, 4.2, 42.6        !DATA FOR WENDY
130  !
140  PRINT Blank$,"MILES    MINUTES     MILES    MINUTES"
150  FOR I=1 TO 6
160    PRINT Name$(I),
170    FOR J=1 TO 2
180      FOR K=1 TO 2
190        PRINT Jog_stats(I,J,K);"     ";
200      NEXT K
210    NEXT J
220    PRINT
230  NEXT I
240  PRINT
250  PRINT "The sum of all elements in Jog_stats is: ";SUM(Jog_stats)
260 !
270  FOR I=1 TO 6                          !Put miles for each jogger
280    FOR J=1 TO 2                        !into two-dimensional matrix
290      Matrix(I,J)=Jog_stats(I,J,1)
300    NEXT J
310  NEXT I
320 !
330  MAT Rows=RSUM(Matrix)
340  PRINT
350  PRINT "The sums of all elements (MILES) in each row are:"
360  PRINT Rows(*);
370  PRINT
380 !
390  MAT Columns=CSUM(Matrix)
400  PRINT
410  PRINT "The sums of all elements (MILES) in each column are:"
420  PRINT Columns(*);
430  END
```

Outputs:

```
             MILES    MINUTES      MILES      MINUTES
   HENRY      8.4       71.2        9.8         82.4
   JOE        5.2       45.6        4.8         41.3
   MARCY      6.3       48.1        6.3         47.2
   SARAH      2.1       23.4        2.2         22.3
   TOM        6.2       36.3        6.3         35.4
   WENDY      4.7       41.3        4.2         42.6

   The sum of all elements in Jog_stats is:  603.599992514

   The sums of all elements (MILES) in each row are:
   18.2  10  12.6  4.3  12.5

   The sums of all elements (MILES) in each column are:
   32.9  33.6
```

Array Functions Involving Linear Algebra

DET returns the determinant of the specified square matrix (one having the same number of rows as columns) or the last matrix which was inverted with the MAT...INV statement. If the determinant is 0, no inverse of the matrix exists (however, an ill-conditioned matrix can produce a non-zero determinant with a poor inverse). The computation is performed in REAL precision arithmetic if the operand is REAL; otherwise it is performed in SHORT precision. The result is REAL. This program uses DET.

```
 10  OPTION BASE 1
 20  INTEGER A(3,3)
 30  SHORT B(3,3)
 40  READ A(*)
 50  DATA 3,1,2,4,1,-6,1,0,1
 60  PRINT "THE ORIGINAL MATRIX IS :"
 70  FOR I=1 TO 3
 80    FOR J=1 TO 3
 90      PRINT A(I,J),
100    NEXT J
110    PRINT
120  NEXT I
130  MAT B=INV(A)
140  PRINT
150  PRINT "THE INVERTED MATRIX IS :"
160  FOR I=1 TO 3
170    FOR J=1 TO 3
180      PRINT B(I,J),
190    NEXT J
200    PRINT
210  NEXT I
220  PRINT
230  PRINT "THE DETERMINANT IS",DET
240  END
```

Outputs:

```
THE ORIGINAL MATRIX IS:
 3                    1                    2
 4                    1                   -6
 1                    0                    1

THE INVERTED MATRIX IS:
-.1111111            .1111111             .8888889
1.1111111           -.1111111            -2.888889
 .1111111           -.1111111             .1111111

THE DETERMINANT IS  -9.00000056854
```

DOT returns the inner or dot product of two vectors. The two vectors must have the same working size. The type of the result is REAL. The following program uses DOT.

```
 10   OPTION BASE 1
 20   INTEGER Array1(3),Array2(3),Dotproduct
 30   MAT Array1=(6)
 40   MAT Array2=(3)
 50   Dotproduct=DOT(Array1,Array2)
 60   PRINT "Array1 IS:",Array1(*)
 70   PRINT
 80   PRINT "Array2 IS:",Array2(*)
 90   PRINT
100   PRINT "THEIR INNER OR DOT PRODUCT IS:",Dotproduct
110   END
```

Outputs:

```
Array1 IS:           6              6              6

Array2 IS            3              3              3

THEIR INNER OR DOT PRODUCT IS:       54
```

MAT...IDN establishes an identity matrix. An identity matrix is a square matrix in which all elements are 0 except those in the main diagonal which equal 1.

MAT...INV returns the inverse of a square matrix. If the matrix is singular (not invertible) no error message is given and a meaningless inverse is calculated. One way to check the inverse is to multiply the original matrix by the inverse; the result should be as close to the identity matrix as is possible with finite precision math. You can also now test the matrix to see if the determinant is 0 with the parameterless DET function, since the inverse has already been calculated with MAT...INV.

MAT...TRN returns the transpose of a matrix. The transpose of a matrix interchanges the rows and columns.

This program uses a matrix which is invertible:

```
 10  OPTION BASE 1
 20  INTEGER A(3,3),D(3,3)
 30  REAL B(3,3),C(3,3)
 40  READ A(*)
 50  DATA 3,1,2,4,1,-6,1,0,1
 60  !
 70  PRINT "THE ORIGINAL MATRIX IS :"
 80  CALL Printint(A(*))
 90  !
100  MAT B=INV(A)
110  PRINT "THE INVERTED MATRIX IS:"
120  CALL Printreal(B(*))
130  !
140  PRINT "THE DETERMINANT IS",DET
150  MAT D=TRN(A)
160  PRINT "THE TRANSPOSE OF A IS"
170  CALL Printint(D(*))
180  !
190  MAT D=IDN
200  PRINT "THE IDENTITY 3X3 MATRIX IS:"
210  CALL Printint(D(*))
220  !
230  MAT C=B*A
240  PRINT "THE INVERSE TIMES THE ORIGINAL MATRIX IS:"
250  CALL Printreal(C(*))
260  END
270  SUB Printint(INTEGER Array(*))
280    FOR I=1 TO 3
290      FOR J=1 TO 3
300        PRINT Array(I,J),
310      NEXT J
320      PRINT
330    NEXT I
340  SUBEND
350  SUB Printreal(REAL Array(*))
360    FOR I=1 TO 3
370      FOR J=1 TO 3
380        PRINT Array(I,J),
390      NEXT J
400      PRINT
410    NEXT I
420  SUBEND
```

Outputs:

```
   THE ORIGINAL MATRIX IS
 : 3                   1                  2
   4                   1                 -6
   1                   0                  1
   THE INVERTED MATRIX IS:
   -.111111111111         .111111111111       .888888888889
   1.11111111111       -.111111111111      -2.88888888889
   .111111111111       -.111111111111      .111111111111

   THE DETERMINANT IS  -9
   THE TRANSPOSE OF A IS
   3                   4                  1
   1                   1                  0
   2                  -6                  1
   THE IDENTITY 3X3 MATRIX IS:
   1                   0                  0
   0                   1                  0
   0                   0                  1
   THE INVERSE TIMES THE ORIGINAL MATRIX IS:
   1                   1.38777878078E-17  0
   -4.44089209850E-16   1                  -4.44089209850E-16
   5.55111512313E-17   1.38777878078E-17  1
```

The original multiplied by the inverse is not exactly the identity matrix because the internal binary representation of the inverse is not exactly the decimal representation.

This program uses a singular (non-invertible matrix):

```
 10  OPTION BASE 1
 20  INTEGER A(3,3),D(3,3)
 30  REAL B(3,3),C(3,3)
 40  READ A(*)
 50  DATA 1,-2,2,1,3,-1,2,1,1
 60  !
 70  PRINT "THE ORIGINAL MATRIX IS :"
 80  CALL Printint(A(*))
 90  !
100  MAT B=INV(A)
110  PRINT "THE INVERTED MATRIX IS:"
120  CALL Printreal(B(*))
130  !
140  PRINT "THE DETERMINANT IS",DET
150  MAT D=TRN(A)
160  PRINT "THE TRANSPOSE OF A IS"
170  CALL Printint(D(*))
180  !
190  MAT D=IDN
200  PRINT "THE IDENTITY 3X3 MATRIX IS:"
```

```
210  CALL Printint(D(*))
220  !
230  MAT C=B*A
240  PRINT "THE INVERSE TIMES THE ORIGINAL MATRIX IS:"
250  CALL Printreal(C(*))
260  END
270  SUB Printint(INTEGER Array(*))
280    FOR I=1 TO 3
290      FOR J=1 TO 3
300        PRINT Array(I,J),
310      NEXT J
320      PRINT
330    NEXT I
340  SUBEND
350  SUB Printreal(REAL Array(*))
360    FOR I=1 TO 3
370      FOR J=1 TO 3
380        PRINT Array(I,J),
390      NEXT J
400      PRINT
410    NEXT I
420  SUBEND
```

Outputs:

```
THE ORIGINAL MATRIX IS:
1                  -2                  2
1                   3                 -1
2                   1                  1

THE INVERTED MATRIX IS:
-1.80143985095E+15  -1.80143985095E+15   1.80143985095E+15
1.35107988821E+15   1.35107988821E+15  -1.35107988821E+15
2.25179981369E+15   2.25179981369E+15  -2.25179981369E+15
THE DETERMINANT IS  -2.22044604925E-15
THE TRANSPOSE OF A IS
1                   1                   2
-2                   3                   1
2                  -1                   1
THE IDENTITY 3X3 MATRIX IS:
1                   0                   0
0                   1                   0
0                   0                   1

THE INVERSE TIMES THE ORIGINAL MATRIX IS:
.5                  0                    .5
.25                  .5                  0
.25                 -1                   .5
```

Note that with finite precision math the determinant of this singular matrix is not exactly 0.

MAX and MIN

The functions MAX and MIN can be used with array arguments as well as with scalar arguments. If you use MAX and MIN with arrays, the calculations are performed by the computer in the precision declared for the array; however the result is coerced to REAL. If you use MAX and MIN with a list of scalar variables, the calculations are perfomed in REAL precision. The following program uses these functions with array and scalar arguments.

```
 10  OPTION BASE 1
 20  SHORT Jog_stats(6,2,2),Num1,Num2,Num3,Num4
 30  REAL Miles(6,2)
 40  DIM Name$(6)[5]
 50  READ Name$(*)
 60  DATA "HENRY","JOE","MARCY","SARAH","TOM","WENDY"
 70  READ Jog_stats(*)
 80  DATA 8.4, 71.2, 9.8, 82.4        !DATA FOR HENRY
 90  DATA 5.2, 45.6, 4.8, 41.3        !DATA FOR JOE
100  DATA 6.3, 48.1, 6.3, 47.2        !DATA FOR MARCY
110  DATA 2.1, 23.4, 2.2, 22.3        !DATA FOR SARAH
120  DATA 6.2, 36.3, 6.3, 35.4        !DATA FOR TOM
130  DATA 4.7, 41.3, 4.2, 42.6        !DATA FOR WENDY
140  !
150  PRINT ,"MILES    MINUTES     MILES    MINUTES"
160  FOR I=1 TO 6
170    PRINT Name$(I),
180    FOR J=1 TO 2
190      FOR K=1 TO 2
200        PRINT Jog_stats(I,J,K);"      ";
210      NEXT K
220    NEXT J
230    PRINT
240  NEXT I
250  FOR I=1 TO 6
260    FOR J=1 TO 2
270      Miles(I,J)=Jog_stats(I,J,1)    !Pick out miles
280    NEXT J
290  NEXT I
300  PRINT "The maximum miles run is:";MAX(Miles(*))
310  PRINT "The minimum miles run is:";MIN(Miles(*))
320  DATA 1.1,2.2,3.3,4.4
330  READ Num1,Num2,Num3,Num4
340  PRINT "The maximum of the first four is:";MAX(Num1,Num2,Num3,Num4)
350  PRINT "The minimum of the first four is:";MIN(Num1,Num2,Num3,Num4)
360  END
```

OUTPUTS:

```
             MILES      MINUTES    MILES      MINUTES
HENRY         8.4         71.2      9.8         82.4
JOE           5.2         45.6      4.8         41.3
MARCY         6.3         48.1      6.3         47.2
SARAH         2.1         23.4      2.2         22.3
TOM           6.2         36.3      6.3         35.4
WENDY         4.7         41.3      4.2         42.6
The maximum miles run is:   9.80000019073
The minimum miles run is:   2.09999990463
The maximum of the first four is:   4.40000009537
The minimum of the first four is:    1.1000002384
```

The digits to the right of the decimal point in the maximum and minimum values are because these values are calculated in REAL **finite** precision. Some decimal numbers do not have exact binary representations; refer to the "Bit Manipulations" chapter.

Array Statements

MAT performs a single operation on all the elements in an array. Instead of explicitly assigning a value to each element in your array, you have a tool to do this with a single statement. In the following examples A, B, and C are arrays with the same number of dimensions.

The legal operators are:

+	addition
−	subtraction
*	scalar or matrix multiplication
.	element by element multiplication
/	division
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to
=	equal to
<>	not equal to

Array operands are not enclosed in parentheses, but scalar operands must be enclosed in parentheses, for example:

```
50 MAT A=(3)        !Puts 3 into every element of array A
80 MAT A=B*(3)      !Multiplies every element of array B
90                  !by 3 putting the results in array A
```

The result and operand arrays must have the same number of dimensions for all operations except matrix multiplication. When using matrix multiplication (MAT A=B*C), the number of columns of the first operand (B) must equal the number of rows of the second operand, (C). The result array cannot be smaller than the operand array(s). The result array is redimensioned before the operation is performed so that it has the same working size as the operand array(s). Only the upper bound is modified, the original lower bound in each dimension is preserved. Type coercion is the same as for scalar variables; refer to the "Numeric Computations" chapter.

These are some of the forms of the MAT... statement:

MAT D=A*B performs a matrix multiplication of A and B, putting the result into D;

MAT B=A copies array A into the system redimensioned array B;

MAT D=A.B puts the element by element multiplication of arrays A and B into array D;

MAT D=A*(3) multiplies each element of the array A by 3 and puts the results into array D;

MAT D=A/(2.3) divides each element of the array A putting the results into array D;

MAT C=(45.99) initializes each element of array C to 45.99;

MAT E$=("OCTOBER 18, 1982") initializes each element of the string array E$;

MAT A=func(B) where func is any system supplied function taking a single numeric parameter. Legal functions are SIN, COS, TAN, ACS, ASN, ATN, EXP, LOG, LGT, SCREEN, SQR, ABS, SGN, INT, FRACT and BINCMP.

The following program uses the above forms of the **MAT** statement.

```
10   OPTION BASE 1
20   REAL Value,Angles(5,5)
30   SHORT A(5,4),B(4,5),C(4,4),D(5,5)
40   DIM E$(6,2)
50   READ A(*)
60   DATA  1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7, 8.8, 9.9, 10.1
70   DATA 11.1,12.2,13.3,14.4,15.5,16.6,17.7,18.8,19.9,20.2
80   PRINT "THE ORIGINAL MATRIX IS:"
90   PRINT A(*)
100  !
110  RESTORE 60                    !USE SAME DATA FOR B
120  READ B(*)
130  !
140  MAT D=A*B                     !MATRIX MULTIPLY
150  PRINT
160  PRINT "THE RESULT OF MATRIX MULTIPLY IS:"
170  FOR I=1 TO 5
180    FOR J=1 TO 5
190      PRINT D(I,J);"     ";
200    NEXT J
210    PRINT
220  NEXT I
230  !
240  MAT B=A                       !COPY A INTO REDIMENSIONED B
250  PRINT
260  PRINT "THE RESULT OF THE COPY IS:"
270  PRINT B(*)
280  !
290  MAT D=A.B                     !MULTIPLY ELEMENT BY ELEMENT
300  PRINT
310  PRINT "THE ELEMENT BY ELEMENT MULTIPLY IS:"
320  PRINT D(*)
330  !
340  MAT D=A*(3)
350  PRINT
360  PRINT "EACH ELEMENT OF THE ARRAY A TIMES 3 INTO D IS:"
370  PRINT D(*)
380  !
390  MAT D=A/(2.3)                 !DIVIDE EACH ELEMENT OF A
400  PRINT
410  PRINT "THE RESULT OF THE DIVIDE OPERATION IS:"
420  PRINT D(*)
430  !
440  MAT C=(45.99)                 !INITIALIZE EACH ELEMENT OF C
450  PRINT
460  PRINT "THE ARRAY INITIALIZED TO 45.99 IS:"
470  PRINT C(*)
480  !
490  MAT E$=("OCTOBER 18, 1982") !STRING INITIALIZE
500  PRINT
510  PRINT "THE STRING  ARRAY INITIALIZED IS:"
520  PRINT E$(*)
530  Value=3.45
540  MAT Angles=COS(D)
550  PRINT
560  PRINT "THE FUNCTION RESULT IS:"
570  PRINT Angles(*);
580  END
```

MAT SORT orders the data in an array. You tell the computer which vector(s) are to control the sort. Not only a primary vector can be indicated to provide the initial sort, but a secondary vector may be appended to provide sorting rules in cases where several array elements are equal. Only as many keys as are necessary to resolve ordering among elements are looked at by the sorting algorithm. Secondary keys are looked at only when they are needed and do not otherwise involve any extra time.

By default the sorting order is in ascending order for numeric arrays and current lexical order for string arrays. DES selects the descending numeric order or reverse lexical order.

If you do not want to modify your original array you can direct the subscripts for the sorted vector to a new one-dimensional pointer array.

```
 10   OPTION BASE 1
 20   INTEGER Numbers(3,4),Sortedcolumns(3),Sortedrows(4),Linarray(12)
 30   N=1
 40   FOR I=1 TO 3
 50     FOR J=1 TO 4
 60       Numbers(I,J)=((-1)^(I+J))*(2*I+J)
 70       Linarray(N)=Numbers(I,J)        !put into linear array
 80       N=N+1
 90     NEXT J
100   NEXT I
110   PRINT "THE ORIGINAL MATRIX IS:"
120   PRINT Numbers(*)
130   MAT SORT Linarray(*)
140   PRINT
150   PRINT "THE SORTED Linarray IS:"
160   PRINT Linarray(*);
170   PRINT
180   MAT SORT Numbers(*,1) TO Sortedcolumns
190   PRINT
200   PRINT "THE Sortedcolumns VECTOR OF ALL ROWS BY COLUMN 1 IS:"
210   PRINT Sortedcolumns(*)
220   PRINT
230   PRINT "THE SORTED RESULT IS:"
240   FOR I=1 TO 3
250     FOR J=1 TO 4
260       PRINT Numbers(Sortedcolumns(I),J),
270     NEXT J
280   NEXT I
290   END
```

Outputs:

```
THE ORIGINAL MATRIX IS:
3                   -4                   5                   -6
-5                   6                  -7                    8
7                   -8                   9                  -10

THE SORTED Linarray IS:
-10 -8 -7 -6 -5 -4  3  5  6  7  8  9

THE Sortedcolumns VECTOR OF ALL ROWS BY COLUMN 1 IS:
2                    1                   3

THE SORTED RESULT IS:
-5                   6                  -7                    8
3                   -4                   5                   -6
7                   -8                   9                  -10
```

MAT REORDER reorders an array. You tell the computer which vector (containing subscript pointers only) you want to sort the original array on, and the original array is reordered accordingly. The vector controlling the reordering can be specified by you or by the statement: MAT SORT...TO... The following program uses MAT SORT...TO... to calculate the controlling pointer vector and then uses MAT REORDER to reorder the array.

```
 10   OPTION BASE 1
 20   DIM Name$(4)[6],Names$(4)[6]
 30   INTEGER Score(4),Scores(4),Pointer(4)
 40   READ Name$(*)
 50   DATA "Arnold", "Betsy", "David", "Joseph"
 60   READ Score(*)
 70   DATA 95, 13, 99, 55
 80   PRINT "THE ORIGINAL MATRICES ARE:"
 90   FOR I=1 TO 4
100     PRINT Name$(I);Score(I);Pointer(I),
110   NEXT I
120   MAT SORT Score(*) TO Pointer
130   PRINT
140   PRINT "AFTER SORTING THE Scores TO Pointer:"
150   FOR I=1 TO 4
160     PRINT Name$(I);Score(I);Pointer(I),
170   NEXT I
180   PRINT
190   MAT Names$=Name$
200   MAT REORDER Names$ BY Pointer
210   MAT Scores=Score
220   MAT REORDER Scores BY Pointer
230   PRINT "AFTER REORDERING THE ORIGINAL MATRIX BY Pointer:"
240   FOR I=1 TO 4
250     PRINT Names$(I);Scores(I),
260   NEXT I
270   PRINT "THE ORIGINAL MATRICES ARE:"
280   FOR I=1 TO 4
290     PRINT Name$(I);Score(I),
300   NEXT I
310   END
```

Outputs:

```
THE ORIGINAL MATRICES ARE:
Arnold 95 0         Betsy 13 0        David 99 0       Joseph 55 0

AFTER SORTING THE Scores TO Pointer:
Arnold 95 2         Betsy 13 4        David 99 1       Joseph 55 3

AFTER REORDERING THE ORIGINAL MATRIX BY Pointer:
Betsy 13            Joseph 55         Arnold 95        David 99
THE ORIGINAL MATRICES ARE:
Arnold 95           Betsy 13          David 99         Joseph 55
```

Chapter 14
Introduction To I/O

The term I/O is a contraction of "Input and Output". The purpose of this chapter is to present three topics concerning I/O:

- To perform I/O, you need to understand something of the structure, or architecture, of the Model 520. This section discusses the I/O components of the Model 520 and the paths used in moving data between your program and an **I/O resource**.
- Once you determine which I/O resource is the destination to which you want to send data or the source from which you want to receive it, you must specify its identity. This section discusses the **addressing** of I/O resources.
- Before you use the resource, you need to be aware of its **characteristics**. This section discusses the **I/O attributes** of the Model 520 I/O resources.

Model 520 I/O Architecture

You can use the BASIC Language System to perform I/O with a wide variety of I/O resources. Before identifying these resources, an overview of the structure of the Model 520 is in order.

The block diagram on the next page shows the major components of a representative Model 520 computer system and its peripherals. This sample system is used in the examples of this and subsequent chapters. The configuration of an actual Model 520 can vary.

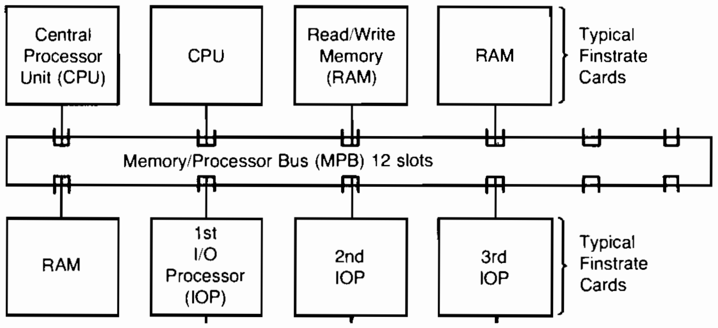
Memory/Processor Module

The heart of the Model 520 is the **Memory/Processor Bus** (MPB). The MPB resides within the Memory/Processor Module assembly of your Model 520. The Memory/Processor Module has card slots which can accommodate up to 12 circuit cards called **finstrates**.

There are three types of finstrates: Central Processor Unit, Memory and Input/Output Processor. Each Model 520 computer has at least one of each type. Although it is useful to know how many and what type of finstrates you have, you do not need to know which finstrate is in which MPB slot.

A **Central Processor Unit** (CPU) executes all program instructions other than I/O. The CPU "runs" your program. **Memory** finstrates contain the read/write, or Random Access Memory (RAM), in which both the BASIC Language System and your program reside.

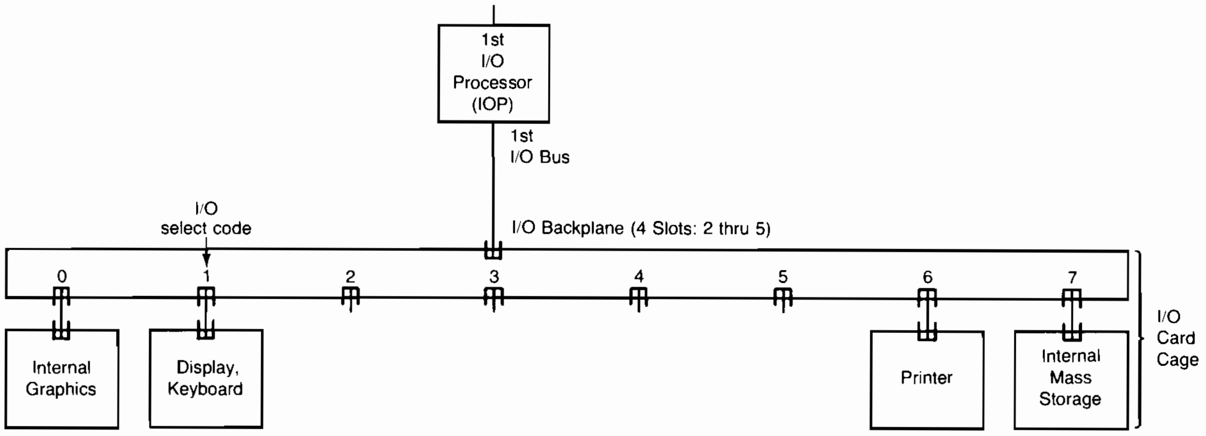
The **Input/Output Processor** (IOP) is a special-purpose processor designed specifically for I/O. An I/O statement in your program is not entirely executed by the CPU. It actually represents instructions which are sent to and executed by an IOP.



Memory/Processor Module

I/O Slots and Interfaces

Each IOP connects the MPB to an I/O card cage containing an **I/O Backplane** which has eight **I/O slots**. In the mainframe four of the first IOP's eight slots are used for the built in (internal) I/O devices. If your computer has a second or third IOP, each controls 8 additional I/O slots on the I/O backplane of an HP 97098A I/O Expander.

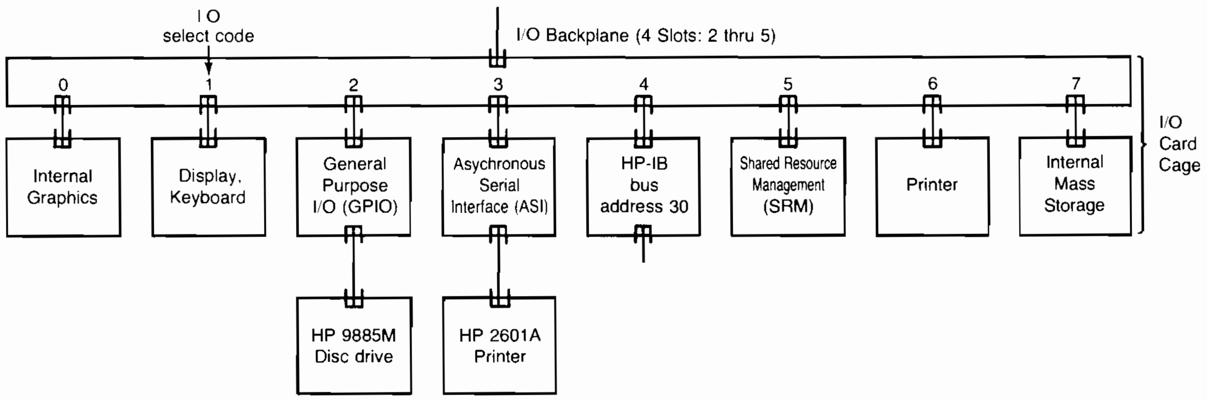


I/O Backplane

Each I/O slot can connect a single **I/O interface card** to the IOP's I/O Backplane. The I/O backplane is electronically and mechanically compatible with any HP-Channel Input/Output (HP-CIO) interface card. There are presently three types of HP-CIO interface cards supported by the BASIC Language System:

Model	Description	Abbreviation
HP 27110A	Hewlett-Packard Interface Bus	HP-IB
HP 27112A	16-bit General Purpose Input/Output	GPIO
HP 27128A	Asynchronous Serial Interface	ASI

An I/O interface card, or simply an interface, is the computer's link with the external world. An interface card connects an I/O backplane to an external **peripheral** device or devices.



I/O Interface Cards and Peripherals

Interface Select Codes

In the computer mainframe the I/O slots are numbered 0 thru 7. The number of an I/O slot is its **interface select code**. Interface select codes 0, 1, 6 and 7 are used by the internal peripheral devices. The following select codes are pre-defined in the Model 520:

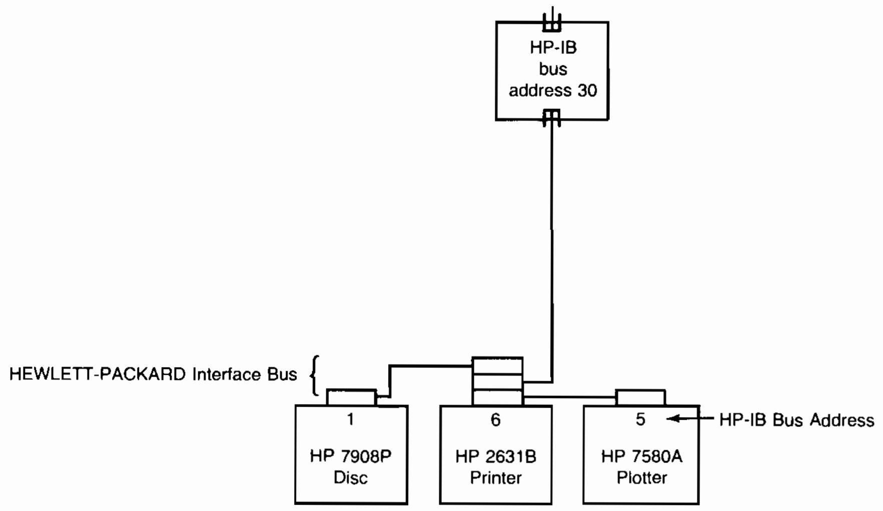
Select Code	Description
0	Graphics CRT (accessed only by GRAPHICS)
1	Model 520 CRT display
6	Model 520 Keyboard and internal printer
7	Model 520 Internal mass storage disc drive(s)

Interface select codes 2 through 5 are available for interface cards. There is a label on the side of the I/O card cage identifying each slot. When you install an interface card in an I/O slot, the card assumes the address of the slot. There are no switches on the card or the computer for this purpose.

If you require more than four I/O slots, you can have an HP Customer Engineer install one or two additional IOP finstrates. Each additional IOP supports one HP 97098A I/O Expander having eight additional I/O slots. The I/O slots in the first expander are interface select codes 8 through 15. The slots in the second are interface select codes 16 through 23.

HP-IB

HP-IB is the interfacing method most commonly used by HP computer systems. The ASI and GPIO cards are used less often and are typically connected to only one type of peripheral device at a time. An HP-IB interface card is frequently connected to several different peripheral devices simultaneously. Because of this, HP-IB effectively represents an additional "layer" in the I/O architecture of an HP computer.



Hewlett-Packard Interface Bus

The bus consists of a cable from the HP-IB interface to a standardized 24-pin connector. An HP-IB connector has both male and female pins. The interface cable is connected to a device's female connector. Additional HP-IB cable assemblies are attached to the interface's connector, or to each other, and connect more HP-IB devices to the bus.

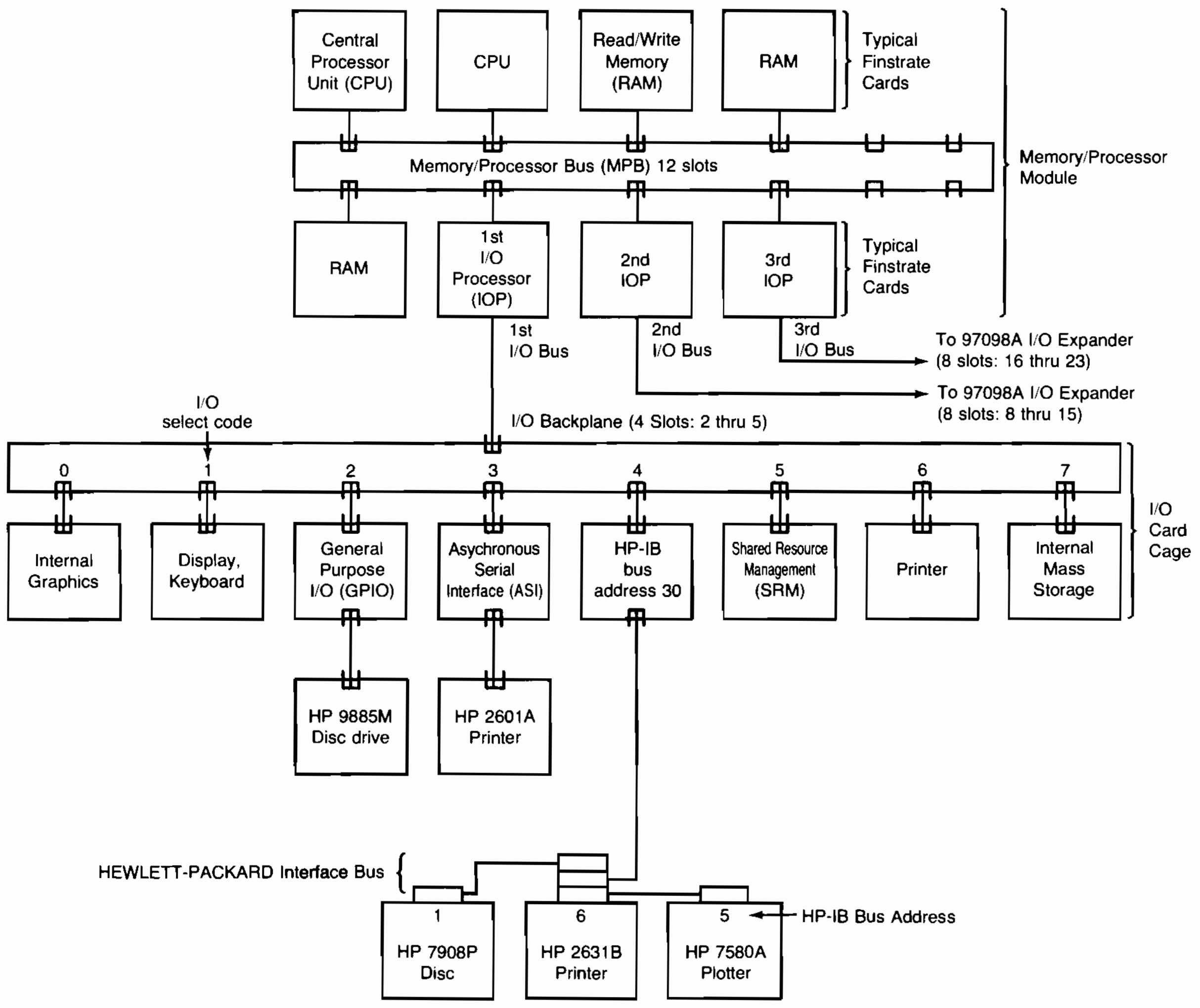
There are three types of bus devices: talkers, listeners and a controller. A **talker** is a device configured to transmit or send data on the bus. A **listener** is a device configured to accept data from the bus. A **controller** is a device which can **reconfigure** the bus by designating which devices are talkers and listeners.

A bus controller can be the **system** controller, **active** controller or both. A switch setting and bus programming establishes whether or not your HP-IB interface is system and/or active controller. Unless otherwise stated, this manual assumes that your computer's HP-IB interface is both system and active controller of the bus.

Each device on the bus is connected in parallel to all lines of the bus. To ensure that only the desired devices are configured as talker and listeners, each should be assigned a unique **bus address** in the range 0 through 30. In the sample system, the HP 7908P disc drive, HP 2631B printer and HP 7580A plotter have bus addresses of 1, 6 and 5 respectively. The HP-IB interface card also has a bus address, 30.

You must know the bus address of a device in order to communicate with it. The controller uses HP-IB **bus commands** specifying device bus addresses to reconfigure the bus.

The bus address of a device is generally set by an in-line or rotary switch on the device. Mass storage devices, such as disc and magnetic tape drives are usually limited to addresses 0 through 7. Many devices are preset by their manufacturer. For example, HP plotters are usually set to address 5. Some devices, such as the HP 7971A Magnetic Tape Drive, require removal of an access panel to set the address. Refer to the device reference manual.



Block Diagram of Sample Model 520 System

I/O Resources

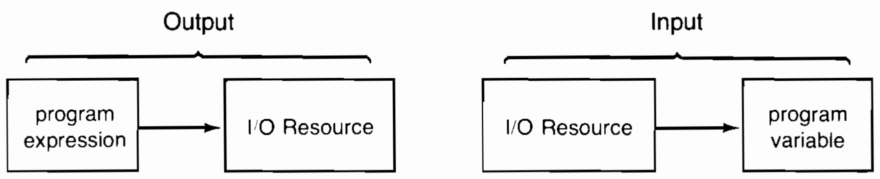
I/O involves either the transfer of **data** or I/O **control** from one computer **resource** to another. Data is the computer's representation of information. Variables and expressions are forms of data. I/O control is an action which affects I/O, but which does not necessarily involve the transfer of data. I/O control is discussed in the "Advanced I/O Operations" chapter.

In the context of I/O, a computer **resource** is any device or memory location of the computer which sends, stores or accepts data, or generates or responds to control. Examples of resources are: files, devices, memory buffers and your program.

Historically, **output** has referred to the process of moving data from computer memory to an external device, and **input** has referred to the reverse process. In contemporary computers the meaning of output and input has broadened, and in some cases the distinction between the terms is very fine.

In this manual output refers to the flow of data from a source within your program to another I/O resource of the computer. Input refers to the transfer of data from an I/O resource to your program. Note that the general terms output and input are in lower case to distinguish them from the BASIC statements OUTPUT and INPUT, which are always upper case.

The following diagrams show the direction of data movement during output and input.



Addressing I/O Resources

To communicate with a specific computer I/O resource, your program must have a means of designating the I/O resource. The BASIC Language System provides four ways to do this:

- device selector - a numeric expression which designates a specific interface card and/or device connected to that card;
- file specifier - a string expression which designates a named mass storage file and optionally password(s), a directory name volume label and device selector by which the file is accessed;
- file number - an identifier consisting of the ASCII "#" character followed by a numeric expression which evaluates to a number from 1 to 10. The file number represents a file specifier.
- I/O path name - an identifier consisting of the ASCII "@" character followed by a BASIC name. The I/O path name denotes a device selector, file specifier or a block of read/write memory known as a BUFFER.

Note

Many of the example program segments in this chapter require that the IO option be loaded in your computer. If you want to try the examples, load the file "IO"

Specifying Devices

Interface cards and the devices connected to interface cards are identified by a **device selector**. A device selector has three components: an interface select code, an optional primary address and zero or more secondary addresses. You can specify a device selector as a decimal constant or as a numeric expression. Device selectors are rounded to an integer when evaluated by the BASIC system.

The use of a variable as device selector makes your program easier to maintain. The actual constant or expression which denotes the device selector need appear only once in your program. All of the I/O statements then specify the variable name. Changing the input or output device requires changing a single statement, instead of every I/O statement in the program.

Specifying an Interface Select Code

The simplest form of device selector is the **interface select code**. It is a numeric expression which BASIC rounds to an integer. The evaluated result must be in the range 1 through 23.

In the example system, the device selector of the HP 2601A Printer is 3. To send the string "Sample text data" to the 2601A, use the following statement:

```
310   OUTPUT 3;"Sample text data"
```

To output this same string to the internal display of your computer you could use the statement:

```
310   OUTPUT 1;"Sample text data"
```

So that you do not have to memorize the device selectors of the internal display keyboard and printer, the BASIC Language System provides the numeric functions CRT, KBD and PRT. Each functions returns the device selectors of the corresponding peripheral. You should use these functions, especially if you want to transport your programs to the Series 200 computers. Using the CRT function, which returns the interface select code of the display, the previous example could also be written:

```
410   OUTPUT CRT;"Sample text data"
```

If you execute an I/O statement which specifies the interface select code of an HP-IB interface card, the I/O statement does not reconfigure the bus. This is referred to as **direct bus I/O**. If I/O can take place at all, it does so between the current talker and listener(s).

Specifying a Device Connected to an Interface

If an interface supports more than one device, simply specifying the interface select code cannot designate a specific device. Examples of such interfaces are internal select code 6 (keyboard and printer) and the HP-IB interface. To designate a device connected to one of these interfaces requires a form of device selector which includes both the interface select code and the **primary address** of the device. HP-IB primary addresses are the bus device addresses in the range 0 through 30. Other interfaces may support a different range of primary addresses. For example, the internal CRT, interface select code 1, has primary addresses in the range 0 through 99 for SCREENs.

This form of device selector is generated by multiplying the interface select code by 100 and adding the primary address. For example, to send the string to the HP 2631B shown in the sample system, execute the following statement:

```
410  OUTPUT 406;"Sample text data"
```

The device selector of the HP 2631B is: (4 * 100) + 6 = 406

When necessary, your program can compute a device selector, for example:

```
670  Device_selector=(Select_code*100)+Primary_address
680  PRINTER IS Device_selector
```

The table of pre-defined select codes noted that both the internal keyboard and printer are connected to interface select code 6. They are distinguished by primary addresses. The printer is primary address 0 and the keyboard is primary address 2. You can specify 600 or 602 to designate each device or you can use the PRT and KBD functions. To send the string to the internal printer use the following statement:

```
410  OUTPUT PRT;"Sample text data"
```

If you execute an I/O statement which specifies the device selector of an HP-IB device, the I/O statement **reconfigures** the bus by sending HP-IB commands. On output, the specified bus devices are addressed as listeners and the HP-IB interface card is addressed as a talker. On input, the first device is addressed as a talker. The computer (and any other devices specified) are addressed as listeners. This addressing is performed automatically by the BASIC system.

Some BASIC statements, such as ABORT, allow **only** an interface select code to designate the I/O device. In these statements, you cannot use a device selector with a primary address. In the BASIC Reference manual, such statements always use the term **interface select code** in their syntax diagrams.

Conversely, other BASIC statements, such as PASS CONTROL, require that the device selector include a primary address. In these statements you cannot specify an interface select code alone. This restriction is noted in the tables and semantics of these statements.

Specifying Secondary Addresses

Some HP-IB devices support secondary commands. When sent on the bus with a primary address, secondary commands are interpreted by bus devices as secondary addresses. Secondary addresses are in the range 0 through 31 and access locations **within** a device or cause a special device action not available through the device's primary address alone.

You can add one through six secondary addresses to a device selector which has a primary address. You cannot add secondary addresses to an interface select code alone. (The first secondary would be interpreted as a primary address.)

Secondary addresses are added to a device selector in the same fashion as the primary address; multiply the device selector by 100 and add the secondary. For example, the device selector for interface select code 11, device 4 and secondary 23 is 110423. Your program can "construct" device selectors in this fashion:

```
680  Device_selector=(Select_code*100)+Primary_address
690  Device_selector=(Device_selector*100)+First_secondary
```

Additional secondary addresses are added by the same method. Multiply the current device selector value by 100 and add the next secondary. The limit on the number of secondaries depends on the number of significant digits available in the variable used to represent the device selector. The recommended variable type for device selectors is REAL. The following table summarizes the limitations of each variable type:

Variable Type	Number of Secondaries	Comments
REAL	6	recommended variable type
DOUBLE	3	4 if select code is 2 thru 21
SHORT	2	3 if select code is 2 or 3
INTEGER	0	1 if select code is 2

Here is an example of the use of secondaries, drawn from the system illustrated at the beginning of the chapter. The HP 2631B printer returns a status byte on receipt of a secondary address of 14. To read this status byte from the printer use the statement:

```
530  Status=ENTERBIN(40614)
```

Specifying Files

When you first access a mass storage file, you must identify the file with a file specifier. A file specifier includes the file name and can include sub-directory name(s), volume label, media specifier and device selector. In addition, access to the file may be denied unless the proper file, directory and volume passwords are also specified. The following BASIC statements show typical file specifiers:

```
 530  CREATE ASCII "TEXT_FILE:INTERNAL",48
 640  PURGE "SOURCES/PROJECT_3/Sorter<Secret>:HP9895,904,1"
 790  Msi$="Dir_name:CS80,7,1"
1302  RENAME Name$&"<"&Password$&">" TO "New_name"
```

File specifiers are briefly introduced here because they are one of the four types of I/O resource specifiers. For more information refer to the "Mass Storage Organization" chapter.

Most of the I/O statements which can access files do not use the file specifier syntax. You only use the full file specifier when you open the file with the ASSIGN # statement. Thereafter you refer to the file in I/O statements with a **file number** or **I/O path name**.

File Numbers

A file number is an identifier beginning with the ASCII character "#" and followed by a numeric expression. The numeric expression, when rounded to an integer, must evaluate to a number in the range 1 through 10. The file number is assigned to a file specifier with the ASSIGN # statement, for example:

```
 950  ASSIGN #3 TO "ENG/Products/SMITH/Lister:CS80,401;LABEL P7908"
 960  !
1200  ! Later in the program ...
1210  READ #3;Data(*)
```

Assigning a file number to a file specifier is referred to as **opening** the file number. The file must exist. ASSIGN # does not create it. When your program has finished I/O operations on the file, you release the file number with the ASSIGN #...TO * form of the ASSIGN # statement. This is referred to as **closing** the file number. It closes the file. It also closes the file's directories and unlocks the disc door if no other files are open. Here is an example of closing a file number:

```
1430  ASSIGN #3 TO *
```

For more information on I/O using file numbers, see the "File Access and Data Transfer" chapter.

Specifying Any Resource - I/O Path Names

The most versatile means of identifying I/O resources is the I/O path name. An I/O path name is an identifier consisting of a literal ASCII "@" character, followed by a BASIC name. The following are examples of I/O path names:

@Io_path_name
@Data_file
@A
@Plotter_3

The I/O path name is the preferred method of specifying I/O resources in the BASIC Language System. Most I/O statements support the use of I/O path names, although some restrict the type of I/O resource to which the I/O path name can be assigned. Other I/O statements, such as TRANSFER, support only I/O path names.

Before using an I/O path name, you must designate the I/O resource it represents. You do this with the ASSIGN @ statement. An I/O path name is like a BASIC variable in some respects. It is only defined in the context which performed the ASSIGN @ statement. Like a variable, it can be declared in COM or passed as a parameter to a subprogram or function. Unlike a variable, it has I/O status and control registers, but does not have a Boolean or computational value.

Assigning and Cancelling I/O Path Names

Like the ASSIGN # statement, ASSIGN @ defines (opens) and ASSIGN@...TO * cancels (closes) the assignment of a particular I/O path name and an I/O resource. The following are examples of ASSIGN @ statements:

```
1196  ASSIGN @Printer TO 406
1720  ASSIGN @File TO "Directory/File_name<Password>:CS80,7"
3204  ASSIGN @Temp TO BUFFER [4096]
```

The precise syntax and semantics of this statement are described in the BASIC Language Reference Manual under the keyword ASSIGN @. You may want to consult that section of the manual during the remainder of the discussion of I/O path names.

You can assign an I/O path name to any of the following I/O resources:

- an interface select code;
- one or more device selectors;
- a file specifier;
- a block of memory known as a BUFFER.

Executing an ASSIGN @ statement is referred to as **opening** the I/O path name. Once assigned, you use the I/O path name just as you would use a device selector or file specifier, for example:

```
1230   OUTPUT @Printer;"Sample text data"
```

When you no longer need the I/O path name, you delete it with the ASSIGN @...TO * form of the statement. This is referred to as **closing** the I/O path name. No further I/O operations can specify the I/O path name. You can also re-assign the same I/O path name by specifying a new I/O resource. This opens the new assignment and closes the old one. The following example closes I/O path name @Printer:

```
1390   ASSIGN @Printer TO *
```

Assigning I/O Path Names to Devices

You can specify any valid form of a **device selector** in an ASSIGN @ statement. Executing the ASSIGN @ statement has no effect on the device. No addressing or I/O occurs until the execution of another I/O statement specifying the I/O path name.

For devices which are connected to the same HP-IB interface, you can assign a single I/O path name to two or more of those devices. For example, if there were two HP 2631B printers, at addresses 6 and 7 on the bus, the following example would print the string to both simultaneously:

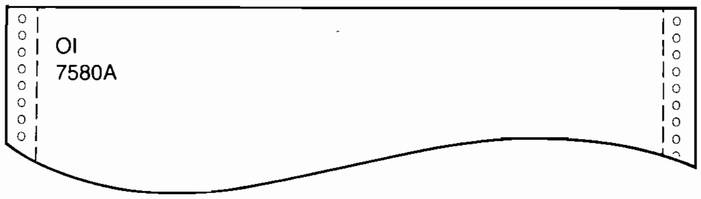
```
310  ASSIGN @Printer TO 406,407          !One name for 2 printers
320  OUTPUT @Printer;"Sample text data"  !Print on both printers
```

The effect of specifying multiple device selectors on **output** is that the I/O statement (OUTPUT) addresses the interface card as a talker and **both** devices as listeners. The string is sent to the bus only once, but is accepted by both devices.

The effect of multiple device selectors on **input** is different. The computer is addressed as a listener. Only the **first** specified device is addressed as a talker. The **second** and subsequent specified devices are addressed as **listeners**. The data sent on the bus by the first device is read by the computer's HP-IB interface, and is also read by the other listeners. For example, HP-GL plotters such as the HP 7580A return their model number when they receive the HP-GL command "OI" (output identification). Suppose you execute the following statements:

```
510  ASSIGN @Devices TO 405,406  !One name for two devices
520  OUTPUT @Devices;"OI"        !Output to both
530  ENTER @Devices;Identity$    !Read from 1st, log on second
```

The effect of these statements is that the OUTPUT sends the string "OI" to both the 7580A (address 5) and the 2631B (address 6). The input statement ENTER causes the 7580A to send its response, the string "7580A", to the computer and to the 2631B. When this program completes, the string "7580A" is stored in the variable Identity$ and the 2631B has printed the following two lines:



This mode is useful for debugging or obtaining a "log" of data sent to and from a bus device such as an instrument.

Assigning I/O Path Names to Files

You can specify any valid form of file specifier in an ASSIGN @ statement. As with ASSIGN #, the file must already exist and you must specify the appropriate passwords. Here are some examples:

```
740   ASSIGN @Data_file TO "DATA_FILE:INTERNAL"
750   ASSIGN @Scratch TO "CALLAHAN<Invert>/TEMP:CS80,402"
760   ASSIGN @Data_file TO *
```

Unlike specifying a device selector, specifying a file does result in I/O operations. When executed, the ASSIGN @ statement opens the file and all of its parent directories. When the file is closed with an ASSIGN @...TO * statement, the directories are closed if no other files are open.

For more information on file access, refer to the "File Access and Transfer Methods" chapter.

Memory as an I/O Resource - BUFFERs

A BUFFER is a block of memory used to temporarily store I/O data. You can output data from your program to a BUFFER, and later (or simultaneously) send it to a file or device with a TRANSFER statement. Or, you can read the data back into your program from the BUFFER, perhaps in a different format. The varied uses of BUFFERs are detailed in the "Advanced I/O Operations" chapter. Some semantics of the ASSIGN @TO BUFFER are briefly summarized here to complete the discussion of the types of I/O path names.

You can use a numeric array or string as a BUFFER by declaring it to be a BUFFER type variable. Such BUFFERs are referred to as **named** BUFFERs. The following are examples of BUFFER variable declarations and assignments:

```
 10   COM Global(16384) BUFFER,String$[1024] BUFFER
 20   DIM Local$[8196] BUFFER
 30  !
350  ! later in the program
360   ASSIGN @Destination TO BUFFER Global(*)
370   ASSIGN @Local TO BUFFER Local$
```

Refer to the "Declaring Variables" chapter for a discussion of BUFFER type variables. When the BUFFER is opened by the ASSIGN @ statement, any data it contains is not disturbed. Registers referenced by I/O statements, however, are initialized to indicate that the BUFFER is "empty". When you close the I/O path name of a named BUFFER with the ASSIGN @...TO * statement, I/O operations can no longer specify that BUFFER, but any data it contains is not lost and is still accessible via the variable name.

You cannot create BUFFER type variables with the ALLOCATE statement. To create a BUFFER of variable size, or which consumes memory only while it is defined, use the form of the ASSIGN @ statement which specifies the **buffer size in bytes**. The BASIC system allocates the necessary memory for this BUFFER, which is referred to as an **unnamed** BUFFER. Here is an example:

```
 390  ASSIGN @Temp TO BUFFER [Size] !Create a BUFFER
 400  !
1360  ! Later in the program....
1370  ASSIGN @Temp TO *            !Delete BUFFER and free memory
```

You can access an unnamed buffer only via I/O statements. When you close the I/O path name of an unnamed BUFFER, the memory is released for other use. Any data still in the BUFFER is lost.

A common error when assigning an I/O path name to a string BUFFER is to omit the secondary keyword BUFFER. If you do this, BASIC evaluates the string, expecting it to contain a file specifier. This usually causes a file error.

You can also treat memory as an I/O resource by creating a mass storage volume with the media specifier ":MEMORY". Refer to the "Mass Storage Organization" chapter.

Detecting Errors in ASSIGN # and ASSIGN @

There are a variety of errors which can occur on execution of the ASSIGN statements. For example, you may assign the file number or I/O path name to a file and not be certain whether the file exists or needs to be created. Although you can "trap" such an error with an ON ERROR statement, it is inconvenient to execute ERRN, and ERRL functions to determine that an error 56 did occur and was in the program line of the ASSIGN statement.

If an error is likely or even possible in your ASSIGN statement, you can specify a RETURN expression in the statement. If an error occurs during the ASSIGN, your program does not pause or take an ON ERROR branch. The ASSIGN stores the error number in the variable specified in the RETURN expression, and your program continues executing normally. If there is no error, the RETURN expression is zero. Here is an example:

```
510  Not_open=51                          !File initially not open
520  WHILE Not_open                       !Until opened.....
530    ASSIGN @Text TO File$;RETURN Not_open  !Attempt to open
540    IF Not_open THEN                   !Open failed, so...
550      CREATE ASCII File$,Sectors;RETURN Not_open
560      IF Not_open THEN  CALL Help      !Serious prob, can't create
570    END IF                             !
580  END WHILE                            !Try again
```

I/O Resource Characteristics

Consider the following example. When you output to an HP-IB device with a simple PRINT or OUTPUT statement, the line of data is terminated with the ASCII carriage-return (CR) and line-feed (LF) characters. This end of line (EOL) sequence is not part of your data. It is supplied by the BASIC Language system and is a characteristic, or attribute, of the I/O resource.

If you are sending your output to an HP 2631B Printer, this is probably how you want the system to work. The EOL causes the printhead to return to column one and the tractor advances the paper one line. On the other hand, if you are sending your data to an instrument which expects only LF at the end of a message and improperly handles the CR character, you need to change the EOL sequence.

In general, you should know what the "default" attributes of the I/O resource are, and if they are inappropriate, how to change them.

I/O Attributes

Each I/O resource supported by the BASIC Language System has default values for a variety of I/O attributes, of which EOL is only one. These attributes are listed for each resource in the "I/O Resources" Appendix of the BASIC Language Reference manual.

These attributes apply to all I/O involving the resource. If you use I/O statements which specify only device selectors or file numbers, you cannot change the defaults. You should, nonetheless, be aware of their values.

If you perform I/O via I/O path names, or to the PRINTER IS or PRINTALL IS device, you can change the default I/O attributes. The next section discusses how to change the I/O attributes of resources accessed via I/O path names, PRINTER IS and PRINTALL IS.

Changing I/O Attributes

To specify one or more I/O attributes of an I/O path name, you append the list of I/O attribute terms and expressions to the ASSIGN @ statement. For example:

```
30   ASSIGN @Prt_1 TO PRT;WIDTH 72
40   ASSIGN @Prt_2 TO 406;LIKE @Prt_1
50   ASSIGN @File TO File$&":INTERNAL";FORMAT OFF,RETURN Error
```

There are 17 I/O attributes which you can specify in the ASSIGN @, PRINTALL IS and PRINTER IS statements. You can specify the RETURN expression to detect errors.

There is also a LIKE expression. This expression, which you cannot specify in combination with any other I/O attributes or RETURN, copies all of the attributes of an open I/O path name. Line 40 of the previous example statements shows the use of the LIKE expression.

If the I/O path name is open (ASSIGN @ already executed), you can **change** the current value of most I/O attributes. You do this by executing an ASSIGN @ statement which specifies the I/O path name and the I/O attributes you want to change, but does **not** specify the I/O resource or the "*" term. For example:

```
930   ASSIGN @Prt_1;WIDTH OFF      !Cancel WIDTH on printer 1.
```

You cannot change **all** I/O attributes of an open I/O path name. To change BYTE/WORD BUFFERSIZE, DRIVER or MAXBUFFS you must first close the I/O path name. Then re-open it, specifying the new I/O attribute values.

Most of the I/O attributes are discussed in the "Advanced I/O Operations" chapter. Four that are frequently mentioned in the introductory I/O chapters are discussed here. They are EOL, WIDTH, WRAP and FORMAT.

End-Of-Line Sequence (EOL)

An EOL sequence consists of zero or more characters. The statements listed below automatically issue an EOL after the last output data byte of each line, and may issue an EOL in between data items, or even within a data item in some circumstances. The purpose of an EOL is to move the location at which output appears to an appropriate location for a new line. The default EOL sequence for all I/O resources is carriage-return (CR), line-feed (LF); the string expression CHR$(13)&CHR$(10).

The following statements automatically issue an EOL: CAT, DBSTATUS, DISP, DISP USING, DUMP ALPHA, LIST, LIST KEY, OUTPUT, OUTPUT USING, PARTITION STATUS, PRINT and PRINT USING.

If this sequence is inappropriate for your peripheral device or application, there are two ways you can change it:

- You can re-define the string expression used for the automatic EOL. This requires either an IMAGE specifier (see the "IMAGE Specifier" chapter) or the EOL I/O attribute.
- You may be able to suppress the automatic EOL and supply your own sequence as string data. Refer to each statement for details on how to suppress EOL.

The following examples show the use of the EOL I/O attribute. The "DVM" is an HP-IB instrument which expects only an LF as the data terminator. The printer is an older teleprinter which requires several character frames of time for the printhead to return to column one. This is accomplished by sending ASCII NUL characters after the CR-LF. In the third example, the EOL expression specifies a null string EOL. This disables the sending of the automatic EOL to the @Prt device.

```
40   ASSIGN @Dvm TO 421;EOL CHR$(10)          !EOL=LF for DVM.
50   Stall$=CHR$(13)&CHR$(10)&CHR$(0)&CHR$(0)&CHR$(0)
60   PRINTER IS 3;EOL Stall$,WIDTH 72,WRAP ON  !CR,LF,3-NULs
70   ASSIGN @Prt;EOL ""                        !Turn off EOL
```

Within the EOL expression you can specify a DELAY. If the device requires a delay to perform CR-LF, or if you merely want to regulate the rate at which you send data to the device, you can use DELAY. DELAY specifies the amount of time (in seconds) that must elapse between lines sent to the device. You can also specify an END in the EOL expression. The use of END is discussed the "File Access and Data Transfer" chapter.

Printer Wrap-Around - WIDTH and WRAP

There is no limit to the length of a line of data that you can output to a printer (or display). Of course, an actual printer and its paper has a finite width, usually expressed as some number of columns. If you send more data in one line than the printer can print, the results are unpredictable. The printer may print beyond the margin onto its platen, overstrike the rightmost column, discard the execess data or advance ("wrap around") to column one of a new line.

In the course of trying the PRINTER IS examples in this manual, you may notice that the internal printer wraps around when the line is longer than 80 columns. This is because the printer has default I/O attributes of WIDTH 80 and WRAP ON.

The WIDTH I/O attribute defines the width of the printer in columns. The WRAP I/O attribute determines how to handle lines longer than the WIDTH. WRAP ON causes the system to insert an additional EOL sequence every WIDTH columns. WRAP OFF causes the system to truncate the output to the number of columns specified in WIDTH.

The defaults for I/O path names are WIDTH OFF, WRAP ON. If you want an external printer to wrap around, you must specify the WIDTH I/O attribute. For example, for an HP 2631B printer using 14 inch wide paper, a typical assignment is:

```
35  ASSIGN @Prt TO 406;WIDTH 132
```

Data Representation - FORMAT

FORMAT determines the form of the data sent by an output statement or expected by an input statement. You can change FORMAT default for some I/O path names with the FORMAT I/O attribute. PRINTER IS and PRINTALL IS default to FORMAT ON and cannot specify FORMAT OFF. The OUTPUT statement discussed in the next chapter supports both FORMAT ON and FORMAT OFF.

If you are familiar with the FORTRAN programming language, do not confuse the BASIC FORMAT attribute with the FORTRAN FORMAT statement. They are not the same. The BASIC IMAGE statement is similar to the FORTRAN FORMAT statement.

Data Structures

FORMAT determines whether or not data is converted from internal form to the ASCII character representation of that form before it is output to the destination. It also determines in what form the data is expected during input. Conversion is necessary because the internal form is not suitable for many peripherals. The internal representation of data was discussed in the "Variables" chapter.

If you do not consider the implications of FORMAT ON/OFF, the I/O results can be unexpected and dramatic. The following example program illustrates the difference between FORMAT ON and FORMAT OFF and also shows the consequence of improper FORMAT OFF output.

```
 10  DOUBLE Bin
 20  Bin=-221                       !Typical DOUBLE value
 30  Disp_func_on$=CHR$(27)&"Y"     !DISPLAY FUNCTIONS on
 40  Disp_func_off$=CHR$(27)&"Z"    !DISPLAY FUNCTIONS off
 50  !
 60  ASSIGN @Prt TO PRT;FORMAT ON   !Open I/O path name, FORMAT ON
 70  OUTPUT @Prt;"FORMAT ON : ";    !Identify FORMAT ON line
 80  OUTPUT @Prt;"TEXT";2.7799;Bin  !Output String, REAL and DOUBLE
 90  !
100  OUTPUT @Prt;"FORMAT OFF: ";    !Identify FORMAT OFF line
110  OUTPUT @Prt;Disp_func_on$;     !Turn on DISPLAY FUNCTIONS
120  ASSIGN @Prt;FORMAT OFF         !Change path name to FORMAT OFF
130  OUTPUT @Prt;"TEXT";2.7799;Bin; !Output String, REAL and DOUBLE
140  !
150  ASSIGN @Prt;FORMAT ON          !Resume FORMAT ON
160  OUTPUT @Prt;Disp_func_off$     !Turn off DISPLAY FUNCTIONS
170  ASSIGN @Prt TO *               !Close I/O path name
180  END
```

This program prints the following result:

```
FORMAT ON : TEST 2.7799-221
FORMAT OFF: NUNUNUETTEXT@AK=<GD14EQ▒▒▒#
```

As you can see, when printed directly with FORMAT OFF, the internal four-byte string length header is output prior to the string as four meaningless characters. REAL and DOUBLE values are output as eight and four meaningless characters, respectively. A similar result would occur with INTEGER and SHORT data. The final ESC and Z characters turn off DISPLAY FUNCTIONS. If the program did not turn on DISPLAY FUNCTIONS, most of the characters in the FORMAT OFF output line would not be visible.

FORMAT ON

FORMAT ON causes output data to be converted from the internal form to a character representation of the internal form. **Strings** are sent as characters but without the length header. Numeric values are converted from the internal form to an ASCII string numeric form according to the prevailing FIXED, FLOAT or STANDARD mode or the applicable IMAGE specifier in DISP USING, PRINT USING and OUTPUT USING statements.

FORMAT ON is always in effect or required for CAT, DBSTATUS, DISP, DISP USING, DUMP ALPHA, LIST, LIST KEY, PARTITION STATUS, PRINT, PRINT USING and OUTPUT USING. If the output is to an I/O path name with FORMAT OFF, the attribute is forced to FORMAT ON. If the destination supports only FORMAT OFF, such as an ASCII file, an error occurs.

FORMAT OFF

FORMAT OFF causes the data to be sent either in unmodified internal form or in a modified internal form suitable for the destination. FORMAT OFF is always in effect for the PRINT# statement and is optional for the OUTPUT statement. You can output data with FORMAT ON to a BDAT file but not an ASCII, BCD or DATA file.

For ASCII files, each output statement builds and sends a single string preceded by a two-byte length header to the file. This "file string" is built from the value of string data and the STANDARD string numeric equivalents of numeric data. It is as if the output statement outputs to the file string with FORMAT ON, and then outputs the length header and file string to the file with FORMAT OFF.

For BCD files, string and INTEGER data types are output with a one-byte type header (SHORT and REAL types are typed by their exponent fields). Strings are also output with an additional two bytes of length header. INTEGER values are output unchanged. DOUBLE values are converted to, and sent as, INTEGERs. If the DOUBLE value is outside the range of an INTEGER, an error occurs. SHORT and REAL values are converted from floating point binary to HP 9800 Binary Coded Decimal (BCD). Because BCD notation has fewer bits to represent a floating point mantissa, some precision may be lost.

For DATA files, each data type is output with a one-byte type header. Strings are also output with an additional four bytes of length header. Numeric values are output in their internal form.

For devices, BUFFERs and BDAT files, string data is sent with its preceding four-byte length header. Numeric values are sent in their unmodified internal form. Because there are no type headers, as in BCD and DATA files, there is no way to determine what data type is represented by any given group (field) of data bytes in a BUFFER or BDAT file. You must take care that your program specifies the appropriate data types when reading FORMAT OFF data from a BUFFER, BDAT file or device, such as a magnetic tape drive.

FORMAT ON/OFF Defaults

The default value for each I/O attribute is described for specific I/O resources in the "I/O Resources" Appendix of the BASIC Language Reference manual. The general rules for the FORMAT attribute are summarized in the following table.

I/O Resource	Default	Comments
Interface/Device	FORMAT ON	FORMAT OFF allowed
BUFFER	FORMAT ON	FORMAT OFF allowed
BDAT file	FORMAT OFF	FORMAT ON allowed
ASCII, BCD, DATA files	FORMAT OFF	FORMAT ON disallowed

Chapter 15
Introduction to Output

The "Introduction to I/O" chapter defines **output** as the transfer of data from your program to other I/O resources. It also discusses how you address those resources. This chapter expands the definition of output and introduces the operations used to perform output.

The Output Operations

The following table summarizes the operations fully described in this chapter.

To perform this operation:	Use this statement:
Display a single-line message.	DISP
Define the standard printer.	PRINTER IS
Output data to the standard printer.	PRINT
List a program.	LIST
List a program's BASIC identifiers.	XREF
List one or more softkey typing aids.	LIST KEY
Copy the contents of the internal CRT display to another I/O device.	DUMP ALPHA
Output data to any I/O resource.	OUTPUT
Output to the tone generator.	BEEP

In order to provide you with an overview of the entire output capability of the BASIC Language System, the following operations are introduced in this chapter, but are only briefly discussed. They are covered fully in subsequent chapters of this manual.

To perform this operation:	Use this statement:
Output data to a numbered file.	PRINT#
List the contents of a file directory.	CAT
List a data base status report.	DBSTATUS[1]
List a partition status report.	PARTITION STATUS
Output data to the internal CRT using an IMAGE specifier.	DISP USING
Output to the standard printer using an IMAGE specifier.	PRINT USING
Output data to any I/O resource using an IMAGE specifier.	OUTPUT USING
Output binary data to any I/O resource.	OUTPUTBIN
Output data to a BUFFER, or from a BUFFER to another I/O resource.	TRANSFER
Output data and/or commands to an HP-IB device.	SEND

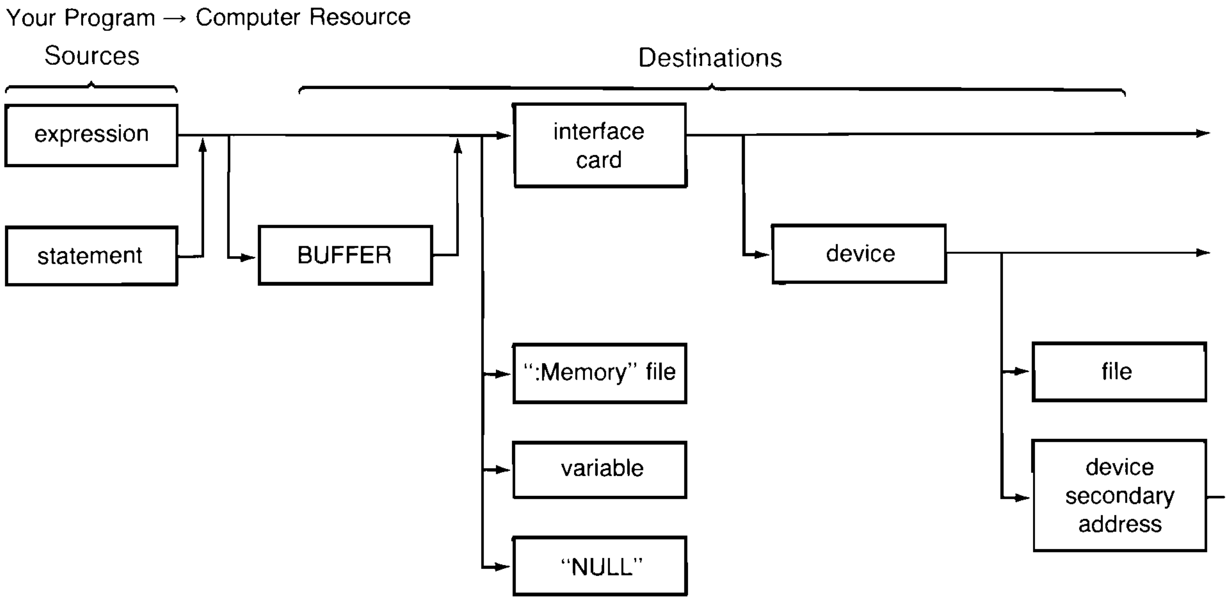
1 DBSTATUS is covered in the IMAGE Data Base Programming Techniques manual.

What is Output?

The output of data is the execution of any output I/O statement that results in the transfer of data from a program expression to an I/O resource. The I/O resource to which the data is output is referred to as the **destination**. The methods of addressing sources are covered in the "Introduction to I/O" chapter.

The following diagram summarizes the paths that data can take during output. As you can see, a destination can be in one of three locations:

- external to the computer;
- within the computer but external to your program (partition);
- within your program.



Displaying Single-line Messages - DISP

When you write an interactive program you generally need to communicate two kinds of information to the person using your program: messages and results. The most common method of displaying messages is to use an area of the internal CRT called the DISPLAYS SCREEN. The DISP statement outputs data to the DISPLAYS SCREEN. (SCREENS are described in the "Computer Operating Features" chapter.)

The CRT is divided into several individually addressable portions, or SCREENs. The system function DISPLAYS is always assigned to one of these SCREENs. At system power-up and after SCRATCH A, DISPLAYS is assigned to SCREEN 2 (device selector 102). SCREEN 2 defaults to line 25, is one line high and has no scrolling buffer.

The DISP statement specifies a list of expressions and output functions (items) separated by commas or semicolons. DISP always performs its output with FORMAT ON. That is, string items are output as characters and numeric items are output in their decimal character representation. In the following example DISP statement, the value of Time is 4793.43 seconds.

```
310  DISP "Elapsed time";Time;"seconds or ";TIME$(Time);" hours."
```

This statement displays the following message.

```
Elapsed time 4793.43 seconds or 01:19:53 hours.
```

Output from a DISP statement begins at the "current position" of the SCREEN. This is normally column one of a new line but can be any position on the SCREEN. When DISP completes, the current position is updated.

Note that because the default DISPLAYS SCREEN is only one line high and has no scrolling buffer, each DISP statement normally causes the previously displayed information to be lost. If the DISP statement's output is longer than one line, only the last line is displayed in the default DISPLAYS SCREEN. If you need to display more than one line, you can use the PRINT or OUTPUT statements directed to another, larger, SCREEN. You can also use the ASSIGN...TO SCREEN statement to re-assign the DISPLAYS function to another SCREEN, or the CREATE SCREEN statement to increase the size of SCREEN 2.

Displaying String Data with DISP

Strings are displayed with no leading or trailing blanks. String expressions are evaluated before output. Literals are displayed "as is", but without the enclosing quotes. If you need to display a quote character in a literal, use two quotes for each quote to be displayed. The following two statements have the same displayed result.

```
70  DISP "I said "&CHR$(34)&"Don't press that key"&CHR$(34)
80  DISP "I said ""Don't press that key"""
```

These lines display the following message.

```
I said "Don't press that key"
```

If the string would extend beyond the width of the SCREEN, the CRT automatically continues the output in column one of the next line. This is a feature of the "INTERNAL_CRT" I/O resource. This wrap is independent of the WIDTH I/O attribute of the internal CRT, which defaults to WIDTH OFF.

A string array is output as a sequence of individual strings. Arrays are displayed in row-major order, that is, the rightmost subscript varies the fastest.

Characters in your data that have codes in the range 160 thru 255 display as European or line drawing characters.

Characters in your data that have codes in the range 0 thru 31 and 128 thru 159 are control characters. If you output a control character to a SCREEN, it does not normally display. It has the effect listed for it in the "Control Character Effects" table in the Useful Tables section of the BASIC Programming Techniques manual. Character strings beginning with control character CHR$(27), the ASCII escape character (ESC), are called **escape sequences**, and have the effect listed in the "Escape Sequence Effects" table.

It is possible to display control characters in their graphical form. To do this you need to enable the DISPLAY FUNCTIONS mode of the SCREEN. DISPLAY FUNCTIONS is enabled by sending the escape sequence CHR$(27)&"Y" to the display. It is disabled by sending the sequence CHR$(27)&"Z". The use of control characters, escape sequences and DISPLAY FUNCTIONS is discussed in the "Advanced I/O Operations" chapter. You can also enable and disable DISPLAY FUNCTIONS mode with the STATUS and CONTROL statements,also discussed in the "Advanced I/O Operations" chapter.

Displaying Numeric Data with DISP

Numeric expressions, including constants and variables, are always displayed as strings of ASCII numeric characters. They are not output in their internal binary form. The form of the string numeric is governed the FIXED, FLOAT or STANDARD mode currently in effect. The examples in this chapter assume that STANDARD (the default) is in effect. The string has a leading blank if the sign of the number is positive. A trailing blank is output after each numeric field.

If the numeric string would extend beyond the edge of the SCREEN, DISP inserts an EOL **before** the number, causing it to be displayed on the next line.

A numeric array is output as a sequence of individual numbers. Arrays are displayed in row-major order. The rightmost subscript varies the fastest.

Positioning Displayed Items

If you output more than one item with a DISP statement, you must separate, or delimit, the items in the list with either a comma or a semicolon.

The **semicolon** delimiter causes DISP to output no additional characters between the displayed items. You must insert whatever blanks or other punctuation is required to make your output readable.

```
560  DISP "The result is ";-2*PI;"furlongs per fortnight."
```

The above example displays the following result:

```
The result is -6.28318530718 furlongs per fortnight.
```

The **comma** delimiter causes DISP to output as many blanks as needed to place the item following the comma in the next **default field**. Default fields are 20 columns wide and begin in columns 1, 21, 41, 61 and every 20 columns thereafter. EOLs are inserted as necessary after column 60. EOL wrap around should be avoided unless DISPLAYS is assigned to a SCREEN having more than one line. Otherwise only the last wrapped portion of the line is visible.

```
610  DISP 1,21,41,61,81,101
```

This example outputs the following two lines, of which only the second remains visible.

```
1                   21                  41                  61
81                  101
```

If you specify an array in the DISP list, the delimiter **after** the array specifier determines the positioning of the array elements displayed. If the array is the last item, and there is no trailing delimiter, a comma is presumed. The following example initializes and displays an array.

```
10   DIM Array(1:10)         !A 10 element array
20   FOR Element=1 TO 10     !For each aray element
30     Array(Element)=Element!Set the element to its number
40   NEXT Element            !Done
50   DISP Array(*)           !Print entire array
60   END
```

This program causes the following output, of which only the final line is visible.

```
1                   2                   3                   4
5                   6                   7                   8
9                  10
```

You can specify adjacent delimiters in the DISP statement with no display item between them. The delimiters have the same effect that they have when items are present. Adjacent semicolons output nothing. An adjacent comma and semicolon is treated as a comma. If you specify adjacent commas, the leading comma(s) output blank default fields and EOLs (if necessary). The last comma determines the positioning of the next item.

Displaying an End Of Line (EOL) Sequence

Like most output statements, DISP sends an EOL after the last output item, unless suppressed. The EOL sequence for DISP is the ASCII characters carriage-return (CR) and line-feed (LF), the string CHR$(13)&CHR$(10).

The CRT handles the EOL sequence in a special way. It does not scroll the last line after an EOL until data is output to the new line. Because the DISPLAYS SCREEN is usually only a single line, this avoids displaying a blank line after each output to the CRT.

Suppressing the EOL Sequence

If you want to append the output of a DISP statement to the contents of the DISPLAYS SCREEN, you must have prevented the previous DISP statement from sending its automatic EOL. In the DISP statement you suppress EOL by specifying a **trailing delimiter**. The following example "constructs" a single display line from different DISP statements.

```
10   DISP "The value is",
20   DISP SQR(PI);
30   DISP "hectares."
40   END
```

The result from this program is:

```
The value is        1.77245385091 hectares.
```

As you can see in the example, the choice of trailing delimiter still affects the positioning of the next display item, even though the item is displayed by a different statement.

DISP with no other items outputs an EOL. DISP with only semicolons outputs nothing. DISP with one or more commas outputs blank default fields for each comma but the last. This may cause EOLs to be issued as well. The next display item appears in the next default field.

Changing Default Positioning - LIN, SPA and tabbing

For each SCREEN, including the DISPLAYS SCREEN, the system maintains a record of the **current position** on the SCREEN. The current position is the SCREEN location at which the next item is displayed. The methods introduced so far allow you move the current position to the next default field (the comma), to column one of a new line (EOL) or to leave the current position at the end of the displayed line (suppressed EOL).

In addition to the choice of delimiter, there are five terms and expressions which you can use in the DISP statement to control the current position. These are the **output functions**: LIN, PAGE, SPA, TAB and TABXY. This section discusses these functions in the context of the DISP statement. These functions are also used with printers and are further described in the "Using a Printer" section.

These terms and expressions are called **functions** because you specify them in the display list like a numeric function and their effect on your output is similar to a string function. When invoked, they change the current position by adding characters to the line output by DISP.

The current position after an output function is determined by two factors:

- the effect of the output function;
- the effect of the delimiter preceding the output function. (The delimiter following the output function is ignored, except as a trailing delimiter.)

Forcing EOL - LIN

When the previous output to the DISPLAYS SCREEN suppressed EOL, your DISP statement **appends** its output to that line. If you want to ensure that your output begins on a new line, use the LIN function as the first item in the DISP statement, for example:

```
410  DISP LIN(1);"This message appears on a new line."
```

The LIN function specifies an end of line count. It adds the specified number of EOL sequences to your output. EOL count values higher than one are useful only if the DISPLAYS SCREEN has more than one line.

Unless you have used the ASSIGN...TO SCREEN or CREATE SCREEN statements to increase the size of the DISPLAYS SCREEN, the PAGE function has the same effect as the LIN statement.

Spacing to the Next Item - SPA

The DISP delimiters enable you to output either zero characters between items (;) or to place each item in the next default field (,). If you want to perform your own spacing, use the SPA function. In the following example, SPA is used to place short strings every 15 characters.

```
530  DISP SPA(15-LEN(A$));A$;SPA(15-LEN(B$));B$;SPA(15-LEN(C$));C$
```

The SPA function specifies a space count. This function adds the specified number of blank (CHR$(32)) characters to your output line. If the line exceeds the remaining width of the DISPLAYS SCREEN, the line wraps around to the next line.

Spacing to a Specific Column - TAB

There are three kinds of tab operations available on the internal CRT: TAB spacing, TABXY repositioning and tab stops. When you want to advance the current output position to a particular column, use the TAB function. In the following example, the prompt "Press CONT to continue" begins in column 59, regardless of the length of the string "Message$".

```
720  DISP Message$;TAB(59);"Press CONT to continue"
```

The TAB function specifies a tab column number. This function adds to your line the number of blank characters required to move the current position to the column specified. If you specify a column number less than the current position, TAB issues an EOL sequence before the tabbing spaces. This means that you cannot tab **backwards** with the TAB function. Use TABXY or tab stopping instead.

Positioning to a Specific Column - TABXY

If you want to move to **any** column, without regard for whether it is before or after the current position, use TABXY. If you position to a column which already contains text, you can overwrite that text, for example:

```
210  DISP "overwritten";TABXY(40,1);"right";TABXY(1,1);"left"
```

produces the following display:

```
leftwritten                            right
```

The TABXY function specifies both a CRT column number and a CRT row number. This function moves the display position to the row and column specified. It does this by adding to your line an HP cursor addressing escape sequence. A value of zero for either row or column signifies that the current row or column value remains unchanged. Unless the DISPLAYS SCREEN has more than one line, TABXY ignores the row number.

You can also move the current position backwards with the ASCII backspace (BS) control character, CHR$(8). If you position to a currently displayed character, subsequent output can replace that character.

Positioning with Tab Stops

Like a typewriter, the CRT display has tab stops. You can set these stops in two ways: with a keystroke and with an escape sequence. You can set any number of tab stops, up to one in each of the 80 SCREEN columns. These tab stops are part of the CRT hardware, so they affect all of the SCREENs of the CRT and all programs (partitions) which use those SCREENs.

To manually set a tab, move the input cursor to the desired column and press TAB SET. To position to a tab, press TAB. If no tabs are set, TAB simply causes a BEEP and does not move the cursor. You can also tab left, or back-tab, by pressing SHIFT and TAB. To clear a tab, position to the tab stop and press SHIFT and TAB SET (TAB CLEAR).

To set a tab (at the current position) in a program, output the escape sequence ESC "1". You can use the string expression CHR$(27)&"1". To position to a tab, output the ASCII Horizontal Tab (HT) control character (CHR$(9)) or the escape sequence ESC "I" (CHR$(27)&"I"). If there is no tab set to the right of the current position, the position is moved to column one of new line. To back-tab, output the escape sequence ESC "i" (CHR$(27)&"i"). To clear a tab, position to that tab stop and output the escape sequence ESC "2" (CHR$(27)&"2"). To clear **all** tabs, regardless of the current position, output the escape sequence ESC "3" (CHR$(27)&"3"). There is no distinction between tabs set with a keystroke and tabs set with an escape sequence.

The following program demonstrates tab stops.

```
10  Esc$=CHR$(27)                  !The ASCII escape character.
20  Ht$=CHR$(9)                    !ASCII horizontal tab
30  DISP Esc$&"3"                  !Clear all old tabs
40  FOR Column=1 TO 71 STEP 10     !For columns 1,11,21 etc...
50    DISP TAB(Column);Esc$&"1"    !  Set a tab stop there
60  NEXT Column
70  DISP 1;Ht$;2;Ht$;3;Ht$;4;Ht$;5  !Demonstrate tab stopping
80  END
```

The resulting output is:

```
1         2         3         4         5
```

Executing Expressions - Implied DISP

When you use the computer as a calculator (when you type in an expression and press EXECUTE), you are executing a DISP statement. This form of DISP is called "implied DISP". It has the same syntax as the explicit DISP, but without the keyword "DISP". Implied DISP output is sent to the RESULTS SCREEN rather than the DISPLAYS SCREEN. The differences between the two versions of DISP are summarized in the following table.

	Explicit DISP	Implied DISP
Keyword DISP	Required	Omitted
Output sent to	DISPLAYS SCREEN	RESULTS SCREEN
Programmable	Yes	No
Executable	Yes	Yes
Output functions	Yes	Yes

The Standard Printer

The standard printer is the default destination for several I/O statements. Each program (partition) has its own assignment for the standard printer. The standard printer is like a pre-defined I/O path name, except that it has no name. You can change its destination and attributes, but it always exists.

The PRINT and PRINT USING statements always send their output to the standard printer. The CAT, DBSTATUS, DUMP ALPHA, LIST, LIST KEY, PRINT, PRINT USING, PARTITION STATUS and XREF statements can specify a destination other than the standard printer.

Assigning the Standard Printer - PRINTER IS

The "standard printer" is not necessarily the thermal printer installed in Model 520. Indeed, your Model 520 may not have an internal printer. The standard printer does not need to be an actual printing device; at system power-up (or partition creation) the default standard printer is the CRT. To change the standard printer use the PRINTER IS statement.

The PRINTER IS assignment affects all contexts of your program. It does not affect other programs (partitions). You can assign the standard printer to three types of destinations: a single device selector, multiple HP-IB devices on the same bus or a mass storage file.

The standard printer also has all of the I/O attributes of an I/O path name. To specify or change any of these attributes you need to load the IO option.

CRT as Standard Printer

The default standard printer is the STD SCREEN. If you change the printer destination, you can reset it to the STD SCREEN with any of the following statements:

```
310  PRINTER IS PRT           !Recommended form
320  PRINTER IS 6
330  PRINTER IS 600
```

You can also assign the standard printer to any other SCREEN of the CRT. If DISPLAYS is assigned to SCREEN 2 (the default), you can assign the printer destination to the same SCREEN used by the DISP statement, for example:

```
310  PRINTER IS SCREEN(2)      !Assign printer to DISPLAYS
```

This causes the PRINT statement to act precisely like the DISP statement.

Regardless of the CRT SCREEN to which you assign the printer, the I/O attribute WIDTH is OFF. This normally does not require any special consideration because each SCREEN performs its own wrap at its right edge.

Internal Printer as Standard Printer

To make the standard printer the internal thermal printer of your Model 520, execute any of the following statements.

```
310  PRINTER IS PRT          !Recommended form
320  PRINTER IS 6
330  PRINTER IS 600
```

When the PRINTER IS assignment is the internal printer, or any destination other than the CRT, the default WIDTH I/O attribute is 80 and WRAP is ON. This means that for lines longer than 80 characters, the system outputs an extra EOL every 80 characters. The default EOL attribute depends on the destination: for devices it is CR,LF (CHR$(13)&CHR$(10)).

Although the internal printer is 80 columns wide, the WIDTH 80 attribute can produce unexpected results. Control characters and escape sequences which produce special effects on the printer are counted by the system as ordinary characters. TABXY is not counted and is the suggested tabbing method if WIDTH is not OFF.

```
380  Cr$=CHR$(13)                 !ASCII carriage return character
390  Esc$=CHR$(27)                !ASCII escape character
400  Ht$=CHR$(9)                  !ASCII horizontal tab character
410  PRINT RPT$("L",50);Cr$;RPT$("R",50) !Overprint Ls and Rs ?
420  PRINT Esc$&"3";TAB(75);Esc$&"1"     !Set tab in column 75
430  PRINT Ht$;"RRRRRRRRRR"              !10 Rs startin in col 75 ?
```

You might expect line 410 to print a single line of overprinted "L"s and "R"s. Instead, you get two lines. Even though the internal printer can overprint, BASIC counts the total characters (101) and inserts an EOL after the 80th character. Conversely, line 430 outputs only 11 characters, yet it also prints as two lines. This is because the printer itself performs a wrap in column 80.

If you are sending control characters or escape sequences to the internal printer, you should disable the wrapping performed by the system. You can do this in the PRINTER IS statement with the WIDTH attribute, for example:

```
350  PRINTER IS PRT;WIDTH OFF     !No system wrap
```

The internal printer still performs its own wrap, which cannot be disabled.

An External Printer as Standard Printer

To assign the standard printer to an external printer, you need to know the interface select code of the printer. If the printer is connected to an HP-IB interface, you need to know (or set) the printer's primary bus address.

For example, the HP 2631B Printer of the example system in the "Introduction to I/O" chapter is connected to the HP-IB interface at select code 4. The printer's primary bus address is 6. (The bus address of a 2631B is set on a switch on the back of the printer near the bus connector.) The device selector is therefore 406. To make this printer the standard printer, execute:

```
120   PRINTER IS 406               !2631B
```

If you expect to use wide paper, compressed fonts or other features of the 2631B, you should change the default WIDTH, for example:

```
120   PRINTER IS 406;WIDTH 132     !14-inch wide paper
```

The 2631B, like many HP-IB printers, also has programmable margins and internal wrapping. You can use the printer's own wrapping instead of the WIDTH/WRAP performed by BASIC. For example, you can program the 2631B to perform its own wrapping at column 132 with the following escape sequences:

```
120   PRINTER IS 406;WIDTH OFF     !No BASIC wrapping.
130   PRINT CHR$(27)&"&a132M";     !Set 2631 margin in col 132
140   PRINT CHR$(27)&"&s0C";       !Enable 2631 local wrap
```

If you have a large amount of output, or want a printed log of data output to an HP-IB device by a PRINT statement, you can assign the system printer to multiple HP-IB devices on the same bus. If there are printers at primary addresses 6 and 7 on the bus, execute:

```
120   PRINTER IS 406,407           !Two standard printers
```

Mass Storage File as Standard Printer

If a printer is unavailable, or available but too slow, you can direct all standard printer output to a file. Printing to files is only summarized here. It is fully discussed in the "File Access and Data Transfer" chapter. Major points relating to printing to files are summarized here.

- PRINTER IS output is only allowed to files of type BDAT.
- The PRINTER IS statement opens the file (and its parent directories). The first output starts at the beginning of the file even if there is already data in the file and even if the file is already open to another program. You should use only statements which output to the standard printer to write to the file. Two programs should not share the same PRINTER IS file.
- All statements which output to the standard printer add their output to the file. The file is closed when the standard printer is reassigned, the program is stopped, the partition deleted or SCRATCH A executed.
- The WIDTH attribute defaults to OFF. End of statement EOL sequences and output function data, such as TABXY escape sequences, are sent to the file as character data and do not reposition the file pointer(s).
- If the file is in a LIF or 9845 directory, an END error occurs when the file is full. If the file is in an SDF directory, it is automatically extended, as necessary, and an END error occurs when no additional file extents can be created.

Here is an example of a program which uses a file as the standard printer. It performs a CAT listing to the file. The second part of the program reads the file and prints its contents on the CRT.

```
 10   DIM Buf$[82],File$[32]       !Data buffer & file specifier
 20   File$="$STDLIST:INTERNAL"    !The list file
 30   CREATE BDAT File$,200,82     !Create list file, 82 byte records
 40   PRINTER IS File$             !Assign standard printer to file
 50   DISP "File created. Writing to it now....."
 60   CAT ":INTERNAL"              !List to the file
 70   PRINTER IS CRT               !Close file & re-assign list
 80   !
 90   ASSIGN @List TO File$;FORMAT ON   !Assign I/O path name to file
100   DISP "Reading from file now....."
110   ON END @List GOTO Close      !On EOF, exit program
120   LOOP                         !Until EOF...
130     ENTER @List;Buf$           !  Read a line
140     PRINT Buf$;                !  Print it on crt
150   END LOOP                     !etc.
160 Close:ASSIGN @List TO *        !Close list file
170   PURGE File$                  !Purge file
180   END
```

Using the Standard Printer - PRINT

When you need to output more than one line of text, or need a permanent copy of your output, you can use the PRINT statement.

The PRINT statement outputs to the standard printer (the current PRINTER IS device). The PRINT statement is very similar to the DISP statement. In fact, if you assign PRINTER IS to SCREEN 2, the syntax and effect of the DISP and PRINT statements are identical. All of the discussion about DISP applies to PRINT. This section is primarily concerned with the differences. It also provides more detail about how the output functions work, so that you can determine their effect on external printers.

The major difference between DISP and PRINT is that the PRINT statement can send its output to devices other than the CRT. The LIN, SPA and TAB functions have the identical effect on the internal printer that they have on the display. TABXY and PAGE have different effects.

Advancing to a New Page

When you want to move the printing position to the top of a new page, you first need to determine what methods your printer supports. The internal CRT and thermal printer of the Model 520 support the ASCII form feed (FF) character. The PAGE output function adds this character to your printed output. For example:

```
310   PRINT "These lines";PAGE;"appear on separate pages."
```

If the standard printer is an external device, you need to verify that the printer properly responds to the form feed. If PAGE is the last item in the print list, you should follow it with a trailing delimiter to suppress the extra blank line on the new page caused by the automatic EOL. All Hewlett-Packard HP-IB printers advance to a new page on receipt of a form feed. Most HP display terminals do not. Refer to the device reference manual.

The internal printer has programmable top and bottom margins. The default top margin is 13 mm. The PAGE function actually advances the paper to the top of a new form plus the top margin. If you want to advance only to the perforations, you must define the top margin to be zero with these escape sequences:

```
730   PRINTER IS PRT            !Internal printer
740   PRINT CHR$(27)&"&l0T";    !Top margin 0 mm.
750   PRINT CHR$(27)&"&l256G";  !Page length 256 mm between margins
```

Changing the Current Position on the Printer

When you want to add spaces to your printed output, use the SPA function. To advance to the next default field, use a comma delimiter. To tab to a specific column, use the TAB function. If WIDTH is OFF, or set to a high value, TAB does not wrap-around after every 80 columns as it does on the CRT. To output extra EOL sequences, use the LIN function.

The internal printer, and most HP external printers, support the same escape sequences for tab stops. Unlike the CRT, if you back tab or backspace, subsequent text does not replace previous text. The internal printer, and some external printers such as the HP 2631B, can overprint the current line.

The internal printer also supports TABXY. It moves the current print position to the column specified and ignores the row. This can result in overprinting.

Positioning to a Different CRT Line

For the normal DISPLAYS SCREEN and the internal printer, TABXY is useful only for positioning to a specific column without regard for the current position. For CRT SCREENs which have more than one line, TABXY can be used to change both the row and column. Try the following example.

```
10   PRINTER IS CRT                   !Use display
20   FOR Row=20 TO 1 STEP -1          !For each row...
30     Col=61-3*Row                   !  Use proportional column
40     PRINT TABXY(Col,Row);Col;Row   !  Position & show position
50   NEXT Row                         !Next row
60   END                              !
```

Unlike TAB and SPA, which move the print position with spaces, TABXY moves the print position with an escape sequence. It is a standard HP screen relative cursor addressing sequence. This sequence is described for the PRINT statement in the BASIC Language Reference manual, and in the "Escape Sequence Effects" table of the Reference manual.

Screen-relative cursor addressing is supported by all HP 238x, HP 262x, HP 264x and HP 27xx CRT terminals. Most HP 263x and HP 267x printers support the column number and ignore the row number. Any device which does not support this escape sequence may respond to it in unpredictable ways.

The TABXY escape sequence is not counted by PRINT in determining whether or not to issue a WRAP EOL at printer WIDTH. The escape sequence is counted if you send it as data rather than with the TABXY function.

Determining the Standard Printer

Good programming practice suggests that when you change a system resource, that you restore its state when you have finished with it. If you want to determine the current assignment of the PRINTER IS destination before you change it, use the SYSTEM$ function, for example:

```
20   DIM Printer_was$[285]     !Longest possible file specifier.
       .
       .
       .
510   Printer_was$=SYSTEM$("PRINTER IS")  !Identify std printer.
```

The string returned is either a file specifier or a list of one or more device selectors. The device selectors are in string numeric form separated by commas. When you restore this assignment, you can use the returned string in the PRINTER IS statement only if it is a file specifier.

If one of more device selectors were returned, you must count them and convert them to numeric form. You can convert a numeric substring to numeric value with the VAL function. Unless you know the device selector count to be fixed number, you also need a separate PRINTER IS statement for each count.

Summary of DISP and PRINT Functions

The following table shows the effect of the output functions.

Function	Effect
;	No spaces output between items.
:	Next item appears in next default field.
LIN	Outputs next item on new line.
SPA	Outputs the specified number of spaces.
TAB	Next item appears at the specified column. The column must be after the current column or a carrige return line feed occurs.
TABXY	Next item appears at the specified column an row. Row is ignored by the internal printer and some other printers.
PAGE	Moves paper on printer to top-of-form.

Listing a Program - LIST

To list a program with the LIST statement, the program must be loaded in memory. You may not be able to list all lines of the MAIN program, subprograms and DEF FN multi-line functions. The following table summarizes the listable parts of programs for the LIST statement.

Program code source	Form of code	Portion of code listable
Editor, ASCII, BCD BDAT or DATA file	Source	All lines.
PROG	Intermediate MAIN and SUBs	All lines.
PROG	Mixture of intermediate and compiled code	All intermediate lines. Only first line of a compiled MAIN program. Only SUB or DEF FN statement line in each compiled subprogram or function.
PROG	Compiled	Only first line of MAIN program. Only SUB or DEF FN statement line in each subprogram or function.
BIN[1]	Binary	None, use CAT_BOOT or SYSTEM$

If there is a program in memory, the LIST statement outputs the date and time, a blank line and one or more lines from the program. Regardless of whether a program is in memory, LIST also displays the available memory in the MESSAGES SCREEN. This is the memory available to your program. If your program resides in a partition for which PRIVATE MEMORY was specified, the available memory is the number of bytes remaining in the partition.

If the program is in source or intermediate form, one line is output for each program line. Here is typical output using a simple LIST command:

```
31 Aug 1982          15:42:24

     10  !Do-nothing program  [RJN] <820901.1519>
     20  FOR Index=1 TO 100000
     30    Nop=Nop
     40  NEXT Index
     50  END
```

If there are more than 80 characters in a program line, the line may be listed as two lines depending on the WIDTH and WRAP attributes of the PRINTER IS device.

The default destination for LIST is the standard printer. You can specify a different device selector if desired, by using the LIST TO syntax. The LIST # form is allowed for compatibility with older HP computers and should not be used in new programs.

The LIST commands can have several forms. For example:

```
LIST TO PRT
LIST Start,Finish
LIST TO 602;500
```

[1] Although you may load BIN option file for use with a specific program, the contents of a BIN file are not part of the program. The option becomes part of the BASIC system and is available to all programs.

The header consists of one line of date and time, and one blank line. If you specify a beginning line number or label, lines prior to that number are not listed. If the beginning line number greater than 999 999, only the compiled subprograms and functions are listed. If you specify an ending line number or label, lines (and compiled subprograms/functions) after that number are not listed.

Listing Compiled Code

If you list compiled code, only one line is output per context. The form of each line depends on the module type: main program, subprogram or function. Here is an example:

```
24 Jan 1983        08:35:12

>>>>>> PROGRAM MAIN !Pager: Report formatting program [HLC] <820812.1315>
>>>>>> SUB Environment !Get global commands           [RJN] <830111.0915>
>>>>>> DEF FNCommand(Text_line$) !Interpret a command [WLM] <820713.1633>
```

For a compiled **main program**, the line contains the first source line of the program, if it is a comment line. The line number is deleted and replaced by `">>>>>> PROGRAM MAIN"`.

For a compiled **subprogram**, the line is the "SUB" statement as it appeared in the original subprogram, except that the line number is replaced by `">>>>>> "`. If the SUB statement contains a comment, it is listed as well.

For a compiled DEF FN multi-line **function**, the line is the "DEF FN" statement as it appeared in the original function, except that the line number is replaced by `">>>>>> "`. If the DEF FN statement contains a comment, it is listed as well.

The example shows comments that identify the module's purpose, author, date and time. Because this comment may be the only way that you can identify a compiled module, it is suggested that you use a convention for the form of the comment.

Listing Control Characters and Escape Sequences

Unlike the EDITOR, LIST does not automatically output control characters as displayable characters. If the program contains a backspace character in a literal that was entered by pressing (CTRL) and (H), it does not appear in your output as a "BS" character. It probably causes overprinting in the listing of that program line. The internal CRT and printer and most external HP peripherals, can represent ASCII control characters only if their DISPLAY FUNCTIONS mode is enabled.

The same is true for display enhancements. If you blink literals with the BLINKING SFK, a printer cannot print this effect, and external peripherals cannot print the Series 500 control code BG even with DISPLAY FUNCTIONS enabled.

There are several ways to avoid these difficulties, but the safest is to express all control characters in programs as string expressions composed of CHR$ functions.

Listing Typing Aid Keys - LIST KEY

To list the typing aid function of one or more SFKs, use the LIST KEY statement. LIST KEY does not list the ON KEY definition of the SFKs. You can list an individual key or all of the keys. The default destination is the standard printer. Here is a listing of the default SFKs:

```
Key  8
▒Clear lineEDIT
Key 9
▒Clear lineFIND ""▒left arrow▒Insert character
Key 10
▒Clear lineCHANGE "" TO ""▒Left arrow▒Left arrow▒Left arrow▒Left arrow
▒Left arrow▒Left arrow▒Left arrow▒Insert character
Key 11
▒Clear lineSCRATCH
```

An underlined field beginning with a smudge (CHR$(255)) character, such as ▒Clear line represents the keystroke of an editing key or EXECUTE. The smudge denotes a keycode rather than a character code. The keycodes of editing keys and EXECUTE are entered in SFKs by pressing CTRL and the editing key.

All other characters represent ordinary character codes. Note that ▒Backspace is not the same as CTRL plus H, the ASCII BS character. ▒Backspace causes the cursor to backspace in the KEYBOARD SCREEN. CTRL plus H is displayed as the ASCII BS character.

As with the LIST statement, control characters in SFKs are always visible when editing the key, but they do not appear during LIST KEY unless DISPLAY FUNCTIONS is enabled on the list device.

The Reporting Statements

This section discusses the I/O implications of the CAT, DBSTATUS and PARTITION STATUS statements. These statements are discussed in detail in the "Working with Files" chapter, the IMAGE/Data Base Programming Techniques manual and the "Partitions and Events" chapter of this manual, respectively.

Each of these statements outputs one or more lines of text in the form of a report consisting of a header and lines of information about files, data sets or partitions. You can specify as the destination for the report a device selector or string array. The default destination is the standard printer.

Reporting to the Standard Printer

Each line of the report is output as a string of characters followed by an EOL sequence consisting of a carriage-return line-feed. You can only change the EOL sequence if you use the standard printer. You change the EOL sequence with the EOL I/O attribute in the PRINTER IS statement.

No additional blank lines or form-feeds are sent before or after the report. If you want the report to start on a new page, or form-feed after it is output, use a `PRINT PAGE;` or `OUTPUT Device_selector;CHR$(12);` statement.

Reporting to a String Array

In addition to specifying a device selector other than the standard printer, you can also send the output of these statements to a string array. Each line of the report is output to an element of the array. The dimensioned size of the array elements must be large enough for the longest line of the report, or an error 18 occurs. The report begins in the first element of the array. Any data in elements used by the report is lost.

Blank lines may be stored in the array as zero-length (null) strings.

The output is performed in row-major order, that is, the right-most subscript is incremented for each line until that dimension is full, then the next dimension to the left is incremented by one. No EOL sequence is sent with the data. Elements in the array beyond the end of the report are set the null string (LEN = 0).

Copying the CRT Display - DUMP ALPHA

If you are performing output to the internal CRT display and need a permanent copy of its contents, you can redirect your program's output. However, this only provides a permanent copy of the contents of **one** SCREEN. If you want a permanant copy of **all** currently displayed SCREENs, use the DUMP ALPHA statement.

DUMP ALPHA outputs the 30 displayable lines of the CRT to the standard printer or any specified device selector. This output includes the visible contents of all currently displayed SCREENs, such as the RUN LIGHT. It does not include scrolling buffer data for displayed SCREENs, nor any data from SCREENs not currently visible.

You cannot specify the device selector of the CRT or any SCREEN in the DUMP ALPHA statement as this might result in circular output to a SCREEN line being output. DUMP ALPHA cannot dump the editor's display; since executing a DUMP ALPHA statement exits the EDITOR. (Use the LIST statement instead.) DUMP ALPHA does not output the contents of the GRAPHICS display, even though this display may be ON and may contain alphanumeric graphics text. (Use the DUMP GRAPHICS statement.) DUMP ALPHA sends a carriage-return line-feed EOL sequence at the end of each line. You can only change this EOL sequence if "dumping" to the standard printer.

Dumping Control Characters

If the CRT contains displayed ASCII or Model 520 control characters, these characters can have the effect described in the "Control Character Effects" and "Escape Sequence Effects" tables in the "Useful Tables" section of the BASIC Language Reference manual. These characters are printed or displayed by the destination, if:

- the DISPLAY functions mode of the destination is ON. Control characters in the line, including EOL, are printed or displayed until an ESC-Z is output. An ESC-Z (CHR$(27)&"Z") turns DISPLAY FUNCTIONS OFF;
- the line contains an ESC-Y (CHR$(27)&"Y") sequence. This turns on DISPLAY FUNCTIONS on the internal and most external HP printers and displays. The ESC-Y itself is not displayed unless DISPLAY FUNCTIONS is already ON.

Dumping National and Line Drawing Characters

Characters output as Model 520 extension character codes in the range CHR$(160) through CHR$(255) are supported only by the internal printer and external devices that are configured for "eighth bit" mode print or display different characters.

External HP printers and terminals use different character codes for national characters. Those which have line drawing require an escape sequence to define the line drawing set as the alternate character set and use the ASCII SO and SI characters to enable the set. Ordinary displayable ASCII characters are then displayed as line drawing characters on a substitution basis.

Katakana characters have the same character codes as ASCII. These characters are displayed only on a computer configured for Katakana. Press (CTRL)(<) or (CTRL)(>) to change from (or to) Katakana to (or from) Roman characters. Katakana literals are output to the CRT or internal printer. They can cause unexpected results in printer which are not configured for them.

Dumping Display Enhancements and Color

Groups of characters displayed with the display enhancements inverse video, blinking, and underline are output with a preceeding extension control character code in the range CHR$(129) through CHR$(135). If the enhancements change, another extension control code is output. When the enhancements end, a CHR$(128) is output. The internal printer supports inverse video and underline. It ignores blinking. Some HP peripherals, such as terminals, interpret these codes as display enhancements, but the particular effect may not be the same.

Groups of characters displayed with colors are output with a preceding HP 9000 extension control character code in the range CHR$(137) through CHR$(143). If the color changes, another extension control code is output. When the color returns to white, CHR$(136) is output. The internal printer ignores these codes. No other HP peripheral presently supports these codes.

DUMP ALPHA always outputs an extension control code for enhancement and color effects, even though the effect was originally displayed on the CRT by an equivalent escape sequence. This is because these effects are always stored in the display memory as control codes. Escape sequences are converted to the equivalent code when stored in the CRT's display memory.

Output to any I/O Resource - OUTPUT

OUTPUT is the most versatile of the output statements. You can use it to output to any I/O resource. Unlike the DISP, PRINT and reporting statements, which always perform I/O with FORMAT ON, OUTPUT supports both FORMAT ON or OFF. There are significant differences between OUTPUT with FORMAT ON and FORMAT OFF. This section discusses each FORMAT mode separately after discussing semantics common to both modes.

You must specify the destination I/O resource in every OUTPUT statement. The types of resources you can specify are:

- device selectors, as one or more numeric expressions;
- the I/O path name of any I/O resource;
- a simple string variable, string array element or substring.

You cannot specify a file number. OUTPUT to the file is always via an I/O path name. If the destination is the DISPLAYS SCREEN of the CRT, or the standard printer, the DISP or PRINT statements may be more convenient to use.

OUTPUT to Devices

OUTPUT to a device or device(s) is performed with FORMAT ON unless you specify a device I/O path name with FORMAT OFF. As demonstrated by the example in the "Data Representation - Format" section of the previous chapter, output with FORMAT OFF is not useful for printing and display devices. It is a useful technique for writing compact data to mass storage devices such as magnetic tapes or to other computers. It is also faster than FORMAT ON because no data conversion is performed.

Here is an example of OUTPUT to a device selector.

```
540  OUTPUT SCREEN(2);"Sample text"       !Output to CRT
```

OUTPUT to a String

Each OUTPUT statement to a string starts at the beginning of the string or substring and updates the current string length on completion. The string must be long enough for the data and the EOL (if any). OUTPUT to an ordinary string is always FORMAT ON. You can output to a string BUFFER with FORMAT OFF via an I/O path name.

Here is an example of OUTPUT to a string.

```
    280   OUTPUT Array$(Line);Record_data    !Output to an array element
```

OUTPUT to a BUFFER

Each OUTPUT statement starts at the byte denoted by the current value of the **fill pointer** associated with the BUFFER's I/O path name. The fill pointer is set to one when the I/O path name is opened and is updated by each output to that I/O path name. It is set to the number of the byte following the last output data byte.

The BUFFER's I/O path name also has an **empty pointer**. This contains the number of the first byte of valid data within the BUFFER. Each statement which reads data from the BUFFER updates this pointer to the number of the byte of data not yet read from the BUFFER. For more information about BUFFERs, see the "Advanced I/O Operations" chapter.

The OUTPUT statement can write into a BUFFER circularly. If it reaches the end of the BUFFER, it continues writing data into the beginning of the BUFFER. It does not write past the empty pointer. It is possible to both output to and input from the same BUFFER simultaneously. This is normally done with the TRANSFER statement, and is discussed in the "Advanced I/O Operations" chapter. OUTPUT to a BUFFER can be either FORMAT ON or OFF.

OUTPUT to a File

File I/O is fully discussed in the "File Access and Transfer Methods" chapter. It is only briefly summarized here.

You can output randomly or serially. **Random** OUTPUT statements specify a **record number**; **serial** OUTPUT statements do not. An I/O path name assigned to a file has one or more variables which denote (point to) the current data byte. When the I/O path name is opened, these pointers are set to one, the first data byte in the file.

You normally use **serial** mode when writing to a newly created file, or adding data to the end of an existing file. A serial OUTPUT begins at the current byte in the file. This position may be within a record. If the data length is longer than the record length, OUTPUT automatically continues writing in subsequent records. The current position is updated to the next byte after each output. For ASCII files, an EOF mark is written in that byte.

You normally use **random** mode for updating records in an existing file. A random OUTPUT begins at the first byte in the specified record. If the data length is less than or equal to the record length, the current byte pointer is updated after the OUTPUT to reflect the first byte of the next serial output. If the data is less than the record length, an end of record mark is written after the data in a BCD or DATA file. If the data is greater than the record length, an END error occurs. You can use random mode only with BCD, BDAT and DATA files, not with ASCII files.

OUTPUT...;END

OUTPUT is unlike DISP and PRINT in that it has only one output function: END. END causes an action which depends on the type of I/O resource. Output END effects are summarized for each resource type in the BASIC Language Reference "I/O Resources" section.

You cannot specify a trailing delimiter after the END term, so END also acts as a trailing delimiter.

OUTPUT with FORMAT ON

OUTPUT with FORMAT ON is similar to DISP and PRINT, except for the meaning of the comma delimiter. String expressions are evaluated and output as characters. Numeric expressions are evaluated, converted per FIXED, FLOAT or STANDARD and output as string numerics. Unlike DISP and PRINT, there is no trailing space after a numeric. Arrays are output as individual elements.

The comma delimiter outputs a comma after a numeric item and a carriage return line feed (not an EOL) after a string item. If it is a trailing comma, both the CR,LF and EOL are suppressed. The semicolon delimiter outputs nothing. An EOL is output after the last data byte unless suppressed by a trailing delimiter or an END term.

OUTPUT with FORMAT OFF

OUTPUT with FORMAT OFF outputs the data in an internal form appropriate to the destination I/O resource. See the "Data Representation - FORMAT" section of the "Introduction to I/O" chapter.

The comma and semicolon delimiters serve only to separate the items in the OUTPUT list. They output nothing themselves. A trailing delimiter is allowed but has no effect. No EOL is output with FORMAT OFF.

Producing Audible Output - BEEP and BEL

Your Model 520 has a programmable tone generator, or beeper. The BEEP statement causes an audible tone of the specified frequency (pitch) and duration.

You can also generate a brief tone by sending an ASCII BEL (CHR$(7)) character to the display in an output statement. Many HP peripherals also generate an audible tone if you output this character to them. The tone caused by the BEL character has the frequency and duration of the simple BEEP statement: `BEEP 500,0.1`

Two typical uses of BEEP are:

- alerting the user to errors or unexpected conditions. For example, the BASIC system executes a simple BEEP when an error occurs;
- sending messages to a user whose attention may not be directed to the CRT. You can use different frequencies or combinations of frequencies to signify different computer responses.

The following program waits for you to press SFK 1 or SFK 2 and gives you an audible indication of which SFK you pressed:

```
10  ON KEY 1 LABEL "PHASE 1" CALL Phase_1
20  ON KEY 2 LABEL "PHASE 2" CALL Phase_2
30 Wait:WAIT
40  GOTO Wait
50 !
60  SUB Phase_1
70    BEEP 888,.1
80    BEEP 555,.1
90    SUBEXIT
100 !
110  SUB Phase_2
120    BEEP 2000,.09
130    BEEP 200,.02
140    BEEP 2000,.09
150    SUBEXIT
160 !
170    END
```

Although a conversion table of frequencies and notes of the chromatic scale is in the Useful Tables Section of the BASIC Language Reference manual, the tone generator is not designed to produce music. It generates a single (square wave) frequency at a time and has no amplitude (volume) control. The duration of each cycle is an integer multiple of 1/50 000 of a second. Nonetheless, you are invited to try the following example:

```
10  A=440
20  Duration=.2
30  FOR Note=0 TO 12
40    Sharp_flat=(Note=1 OR Note=3 OR Note=6 OR Note=8 OR Note=10)
50    IF NOT Sharp_flat THEN
60      Frequency=A*2^(Note/12)
70      BEEP Frequency,Duration
80      WAIT .05
90    END IF
100  NEXT Note
110  END
```

Chapter 16
Introduction to Input

The "Introduction to I/O" chapter defines **input** as the transfer of data from an I/O resource to your program. It also discusses the methods of addressing I/O resources. This chapter expands the definition and introduces the BASIC operations that you can use to perform input.

The Input Operations

The following table summarizes the input operations fully discussed in this chapter.

To perform this operation:	**Use this statement:**
Read alphanumeric keyboard input into string variables or into numeric variables, using the number builder.	INPUT
Read alphanumeric keyboard input "unparsed" into a string.	LINPUT
Modify the current value of a string from the keyboard.	EDIT...$
Read alphanumeric keyboard input as numeric and string values, using the number builder.	ENTER KBD
Read keyboard keystrokes as keycodes; perform keyboard input without wait; redefine keyboard.	ON KBD, KBD$
Input data from a DATA statement within your program.	READ
Change the DATA statement used by the next READ statement.	RESTORE
Read data from any I/O resource with FORMAT ON (number builder) or FORMAT OFF.	ENTER

The following input operations are introduced in this chapter to provide you with an overview of the entire input capability of BASIC, but are not fully discussed in this chapter. Most are covered in subsequent chapters of this manual.

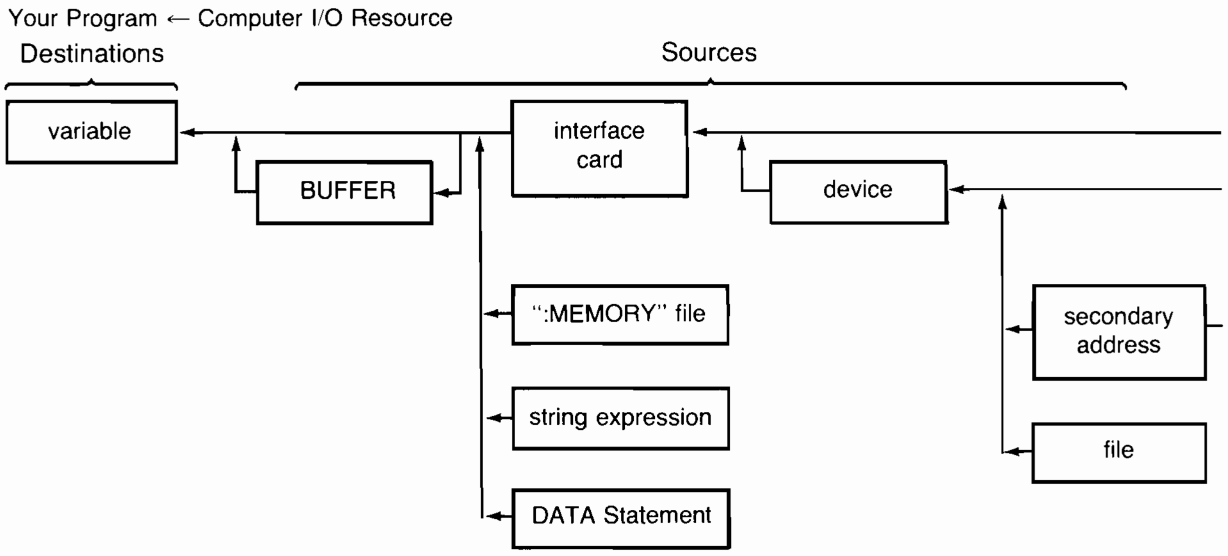
To perform this operation:	**Use this statement:**
Read file data via a file number.	READ#
Input data from any resource using an IMAGE specifier.	ENTER USING
Input binary data from any resource.	ENTERBIN
Input data to a BUFFER from an I/O resource.	TRANSFER

What is Input?

The input of data is the execution of any input I/O statement that results in the transfer of data to a program variable or BUFFER from an I/O resource. The I/O resource from which the data is input is referred to as the **source**. The methods of addressing sources are covered in the "Introduction to I/O" chapter.

The following diagram summarizes the paths that data can take during input. As you can see, a data source can be in one of three locations:

- external to the computer;
- within the computer but external to your program (partition);
- within your program.



Input with FORMAT ON/OFF

In the "Introduction to Output" chapter you saw that output was affected by whether or not the FORMAT I/O attribute was ON or OFF. This also applies to input. Each I/O source has its own default value for FORMAT. If the source is accessed via an I/O path name, you may be able to change the FORMAT attribute.

During output, FORMAT affects the form of the data at the destination. During input it affects the form of data expected from the source. If FORMAT is OFF, the form expected is the same as the form output with FORMAT OFF. This is not true of FORMAT ON.

Input with FORMAT ON

As in FORMAT ON output, the data is a string of characters. String data can include any character. Numeric data must be ASCII string numerics. Numeric values output with FORMAT ON are sent in the current STANDARD, FIXED or FLOAT representation. Input numeric values need not conform precisely to these notations. FORMAT ON input allows considerable freedom in the representation of numbers. For example, if you execute the following program segment:

```
10   DISP "Enter a number"
20   ENTER KBD;Value
30   DISP Value
40   END
```

and you enter the following response:

```
ABCD000.001234E-09.0XYZ RETURN
```

the ENTER statement ignores the "ABCD", accepts and ignores the leading zeros before the decimal point, accepts and evaluates the leading zeros after the decimal point, accepts the "E" as an exponent indicator and ignores the leading zero in the exponent. The second decimal point is an item delimiter. ENTER ceases accepting numeric characters for the variable `Value`. ENTER ignores subsequent characters, evaluates what you have entered, and "builds" the value of the variable, "Value":

```
1.23400000000E-12
```

This kind of process is known as "parsing". It involves decisions by the computer of what is a number and what is not, and how one item in a list is distinguished (delimited) from another. The rules of parsing apply to the READ and INPUT statements and ENTER with FORMAT ON. Although not strictly input related, parsing also applies to the VAL statement and the entry of constants in the Editor.

The Number Builder

The parsing of data from string numerics to INTEGER, DOUBLE, SHORT and REAL values is performed by the input statement using a system algorithm known as the "number builder".

In general the number builder expects well formed numbers. The number may begin with a sign (+, −), followed by a mantissa containing at least one digit (0,1 thru 9) and zero or one decimal point. If more digits are received for the mantissa than a REAL data type can represent, the extra digits are included in the calculation of the magnitude of the exponent, but are otherwise ignored.

An exponent field can follow the mantissa. It must begin with an exponent letter (e,E,D,L) and is followed by at least one digit of the exponent magnitude. The magnitude field can begin with a sign and must consist of one, two or three digits. A decimal point, another sign or any non-numeric character in the exponent terminates (delimits) the entire number.

The following table summarizes the differences in three types of number builder operations. **Evaluated** means that BASIC evaluates the non-numerics as variable or function names rather than send them to the number builder. **Delim** means that the conversion of the substring stops at that character.

Characters input	**Statements Affected**			
	INPUT, editor	**VAL**	**READ**	**ENTER**
Leading non-numeric	Evaluated	Error	Error	Ignored
Extra decimal point, sign or exponent character	Error	Delim	Error	Delim
D and L in exponent	Error	OK	OK	OK
Leading and trailing blanks	Ignored	Ignored	Ignored	Ignored
Embedded blanks with SPACE DEPENDENT ON	Error	Ignored	Ignored	Ignored
Embedded blanks with SPACE DEPENDENT OFF	Ignored	Ignored	Ignored	Ignored

Numeric Input Parsing

Keyboard Input

There are seven statements that you can use to perform data input from the keyboard. They are INPUT, LINPUT, EDIT...$, ENTER, ENTERBIN, ENTER USING and ON KBD. Each offers different capabilities.

Prompting

Although keyboard input might consist of whole paragraphs of text, more frequently the program needs entry of distinct lines of data. Good programming practice suggests that you display a message (a prompt) alerting the operator that input is required, identifying the kind of data required and perhaps identifying the program. Here is an example of one way to perform prompted keyboard input:

```
570   INPUT "PROGX: Enter your first name";First_name$
```

prompt

For prompted input, the INPUT, LINPUT and EDIT...$ statements are generally the best choice. They have the advantage that a single statement issues the prompt and reads the input data. You can issue the prompt and perform the read in separate operations, as in the following example.

```
570   DISP "PROGX: Enter your first name";
580   ENTER KBD;First_name$
```

A potential disadvantage of this method is that your program could be interrupted after displaying the prompt, but before executing the read. This could happen as a result of an ON condition or the execution of a higher priority program in another partition. If this happens, the prompt is misleading because the computer is not actually ready for input.

Entering Numeric Data

To assign keyboard input to numeric variables the string numerics entered must be converted to numeric values. This conversion can be performed in three ways.

- The input statement can convert the data. The INPUT, ENTER and ENTER USING statements automatically invoke the number builder for numeric items in their input lists. INPUT expects numbers to be delimited by a comma. ENTER allows any non-blank non-numeric character.
- You can convert the data fields in your program after input. You can enter the keyboard data as a string with EDIT...$, ENTER, ENTER USING or LINPUT and convert its substrings with the VAL, IVAL or DVAL functions. There is an example of this type of conversion in the "ON... Conditions" chapter in the "Skipping Program Lines - ON...GOTO" section; IVAL and DVAL are useful if the data is entered in binary, octal or hexadecimal notation. You can also convert substrings by "entering" from the substrings with the ENTER and ENTER USING statements.
- You can convert the data as a character stream in your program after input. You can enter the data as a string and interpret its substrings with character manipulations. This is not simple. Before considering this approach, make certain that the conversion you require cannot be performed by a string function or IMAGE specifier.

Entering String Data

All of the input statements except ENTERBIN support keyboard entry into strings. The EDIT...$, LINPUT and ON KBD/KBD$ statements assign data **only** to strings. Although the number builder does not apply to string input, there are variations in the string parsing performed by each input statement.

Here are some general guidelines for selecting a string input statement.

- Use INPUT for entry of multiple items, especially a mixture of numeric and string data. INPUT is also the only input statement which allows entry of numeric expressions, such as variable names. These are evaluated, and the resulting value is assigned to the input list variable. Here is an example of INPUT.

```
120   INPUT "INPUT: Name,Age",Name$(Index),Age(Index)
```

- Use LINPUT for assignment of single strings to a variable whose current value is not needed, such as the null string. LINPUT is especially useful when you need to treat commas, quotes, delimiters and control characters as data. It is also useful when you expect either string or numeric entry, for example:

```
120   LINPUT "Enter a number or string",Entry$   !Parse later
```

- Use EDIT...$ in place of LINPUT when the string variable may have a current (default) value which should be displayed along with the prompt. You can leave the value unchanged, modify (edit) it or completely replace it, for example:

```
110   Specifier$="Scratch:CS80,401"                   !Set default
120   EDIT "Enter work file specifier",Specifier$     !Modify?
```

- Use ENTER KBD in place of INPUT for entry of numeric data with less restrictive number builder parsing rules. Use ENTER...USING when you want to precisely control the parsing rules, for example:

```
100   DISP LIN(1);"Enter part number" !Issue prompt
110 Field: IMAGE DDDDDXDDDDD          !Typical#= 97050-90000
120   ENTER KBD USING Field;Part_no   !Enter # with embedded minus
```

- Use ON KBD for single-character input, and input without-wait. There is an example of ON KBD input later in this chapter.

The following table summarizes the parsing rules for INPUT, EDIT...$, ENTER and LINPUT. The other input statements are sufficiently different that they are discussed separately.

Feature	INPUT	EDIT...$	Enter	LINPUT
delimiter for multiple items	comma	n.a.	none	n.a.
implied TRIM$ of unquoted strings	YES	NO	NO	NO
trailing blanks removed from unquoted strings	YES	NO	Only after last item	NO
quotes removed from around quoted strings	YES	NO	NO	NO
enter quotes and commas only within quotes	YES	NO	NO	NO
enter quote character as	""	"	"	"
input > length of string variable	ERROR 18	ERROR 18	Excess held in buffer	ERROR 18

String Parsing Summary

The parsing rules for ENTER USING are controlled by the IMAGE specifier, which is discussed in the next chapter.

An ENTERBIN function specifying the keyboard returns the numeric character code of the first character entered. The second and succeeding characters are lost. The entry is terminated by pressing RETURN or STEP. Because the system program which supports the KEYBOARD removes trailing blanks, the space character (CHR$(32)) must be entered followed by at least one other character before pressing RETURN.

There is no parsing associated with the ON KBD statement and the KBD$ function. Their use is discussed in the next section.

Input Without Wait - ON KBD

The ON KBD statement defines a program branch which can occur when any key is pressed. While an ON KBD is active, keystrokes are stored in a temporary buffer. You can retrieve them with the KBD$ function. The combination of ON KBD and KBD$ provides you with three powerful capabilities.

- You can read single characters. Your program can read characters entered one at a time without waiting for RETURN.
- You can perform keyboard entry without wait. Your program can perform other operations while keyboard data entry is in progress.
- You can re-define the keyboard. You can turn part, or all, of the keys into SFKs. Your program can apply new meanings to keys or deny access to their normal function.

Keyboard Input Timeouts

The ordinary input statements EDIT...$, ENTER, ENTER USING, INPUT and LINPUT cause your program to suspend until you press RETURN or STEP. If the operator never presses these keys, the program can be suspended indefinitely. You cannot define an ON TIMEOUT condition for the keyboard device selector or I/O path name assigned to KBD.

The following program and subprogram show one method for using ON KBD to perform input without suspension (wait) and with a timeout. It uses ON DELAY to establish the timeout. If input does not complete in 30 seconds the "GOTO Timeout" branch occurs. The main program initiates the I/O by setting up the ON KBD branch and displaying the prompt. It performs some unrelated activity (the loop) and then enters a general WAIT state if the input has not yet completed.

Each keystroke is processed by the subprogram "Kbd_input". All keystrokes are echoed back to the keyboard (and KEYBOARD SCREEN) except RETURN or STEP. Either of these keys cause the subprogram to read the contents of the KEYBOARD SCREEN (3), clear that SCREEN and alert the main program with SIGNAL 6.

Without the ON DELAY/timeout feature, this is also an example of simple input without wait.

```
  10 !Program to demonstrate KBD input with timeout & without wait.
  20  COM /Kbd/ Kbd$[80]                 !Input string
  30 !
  40  More=1                             !Input not finished flag
  50  ON DELAY 30,15 GOTO Timeout        !Timeout in 30 sec
  60  ON SIGNAL 6,13 GOSUB Data_input    !RETURN or STEP pressed
  70  ON KBD ALL,14 CALL Input_Kbd       !Get a keystroke
  80  DISP LIN(1);"Enter something";    !Issue prompt
  90 !
 100  FOR Index=1 TO 100000              !Make-work loop
 110    Nothing=Index*PI
 120  NEXT Index
 130 !
 140 Wait: IF More THEN WAIT             !Wait if input not finished
 150  IF NOT More THEN STOP              !Input finished, stop
 160  GOTO Wait                          !Loop after each ON KBD
 170  !
 180 Timeout: !RETURN was not pressed within 30 seconds
 190  BEEP                               !Honk
 200  DISP LIN(1);"Keyboard input timeout. Program aborted.";
 210  STOP
 220 !
 230 Data_input:OFF KBD
 240  PRINT "[";Kbd$;"]"                 !Show input
 250  More=0                             !Flag that input is done
 260  RETURN                             !Return to mainline code
 270 !
 280  END

1000  SUB Input_Kbd !Keyboard entry without wait
1010    COM /Kbd/ Kbd$[80]                    !Entered data buffer
1020    DIM Input$[80]                        !Local data buffer
1030 !
1040    Input$=KBD$                           !Get entry
1050    OFF KBD                               !Hold off keys
1060    WHILE LEN(Input$)                     !Until buffer empty...
1070      Chr=NUM(Input$[1;1])                !Get keycode
1080      IF Chr=255 THEN                     !If a non-character...
1090        Flush=2                           !Send back 2 chars
1100        Chr=NUM(Input$[2;1])              !Get second code
1110        IF (Chr=69) OR (Chr=83) THEN      !If RETURN or STEP
1120           OUTPUT SCREEN(3);CHR$(27)&"G";  !Cursor to col 1
1130          ENTER SCREEN(3);Kbd$            !Read KEYBOARD SCREEN
```

```
1140          OUTPUT SCREEN(3);CHR$(27)&"H"&CHR$(27)&"J";!Clear display
1150          SIGNAL 6                         !Signal MAIN program
1160        ELSE                               !Not RETURN or STEP
1170          IF Chr=255 THEN Flush=3          !If a CTRL Keycode...
1180          OUTPUT KBD;Input$[1;Flush];      !Send back to KBD/CRT
1190        END IF                             !
1200        Input$=Input$[Flush+1]             !Delete keycode
1210      ELSE                                 !Was a character...
1220        OUTPUT KBD;Input$[1,1];            !Send back to KBD/CRT
1230        Input$=Input$[2]                   !Delete from buffer
1240      END IF
1250    END WHILE
1260    SUBEXIT
1270 !
1280  SUBEND
```

Any key that does not represent a displayable character is stored in the KBD$ buffer as two or three characters. The first is the extension character code CHR$(255) and the rest are "Function Key Codes" from the "ON KBD Function Key Codes" table in the Useful Tables section of the BASIC Language Reference manual.

You may not want to process every keycode in your service routine. For example "Input_kbd" processes only RETURN and STEP. You can have the system process keycodes by returning them to the keyboard with OUTPUT KBD;... (be sure to suppress EOL). This is the easiest way to let the operator enter and **edit** information. On completion, simply read the KEYBOARD SCREEN, as in the example.

ON KBD Cautions

You do not have to execute an ON KBD branch for the contents of KBD$ to be valid. You can read KBD$ at any time, but reading it also empties it. OFF KBD also empties it. If you read it while your ON KBD branch is disabled and subsequently enable the branch, the service routine may find that KBD$ contains no characters.

If more than one key is pressed while the ON KBD statement is active but disabled, KBD$ returns more than one keycode when executed. That is why the "Input_kbd" example examines the KBD$ information in a WHILE loop. If more than 80 bytes of key codes and function key codes are entered, those after the 80th byte are lost.

ON KBD ALL is powerful but dangerous. You can use "ALL" to deny access to any key or all keys. A common use for ON KBD ALL is to prevent accidental pressing of STOP during critical parts of a program. The risk is that there is now **no way** for the operator to stop the program. If the program contains an error, such as an infinite loop, you must **turn off** the computer to stop it. Your ON KBD ALL service routine should check for STOP and PAUSE, or define an SFK which allows the operator to bring your program to an orderly termination.

Memory Input

Input statements normally transfer data from files or external devices to program variables in memory. There are times when the data is already in memory but you still need to access it with an input statement. Here are some typical reasons.

- The data may be in the form of DATA statements within your program.
- The data may already be in a program variable but is in the wrong form; for example, binary numeric data in a string variable.
- The data may be in a BUFFER. In particular, string data in a numeric array BUFFER is virtually useless until read from the buffer into a string variable with an ENTER statement.
- You may want to test an ENTER statement with data from a string expression or BUFFER before supplying it with real data from an external device.

The following sections describe memory-to-memory I/O input using DATA, READ and RESTORE, and data conversions after input using various BASIC statements and functions. Since ENTER from a BUFFER is generally preceded by a TRANSFER to the BUFFER, the input of data from a BUFFER is discussed in the "Advanced I/O Operations" chapter.

DATA, READ and RESTORE

Assigning values to variables was discussed in the "Declaring Variables" chapter. If you have written any programs in which variables needed to be initialized prior to use, you have probably used simple assignments like:

```
30   Factor=-4.89456
40   Prompt$="PROG_NAME: "
50   File_name$="<none>"
```

You can also assign values to variables with the READ statement. The following statements perform the same assignment as the previous example:

```
30   DATA -4.89456,"PROG_NAME: ",<none>
40   READ Factor,Prompt$,File_name$
```

Data specified in DATA statements is similar to data accepted by an INPUT statement except that the exclamation point character can only be in a quoted string, like the comma, quote and leading and trailing blanks. This is because the exclamation point would otherwise denote the start of a comment. The data, whether representing numbers or strings, is stored in string form.

DATA statements are read serially within each context by READ statements. Each succeeding READ starts with the first item not read by previous READ statements. DATA statements do not have to be adjacent, although having them all in one block eases program maintenance. DATA statements do not have to precede any READ statement which accesses them.

To re-read information from DATA statements which have already been read once or to explicitly control which DATA statement is accessed by a READ statement, use the RESTORE[1] statement. RESTORE specifies a line number or label. The next READ statement after a RESTORE statement accesses the DATA statement in the specifed line. If the RESTORE'd line is not a DATA statement, the READ accesses the next statement in the context that is a DATA statement.

1 The RESTORE statement is not the same as the RE-STORE statement which is discussed in the "Working with Files" chapter.

READ has several uses:

- Assignment with DATA and READ can require fewer program statements than simple assignment, especially if you are initializing an array. For numeric data, READ is slower than constant assignment because of the string to numeric conversion.
- Since the data is stored in string form within your program, you can READ numeric data into either numeric or string variables. The string variable must be long enough to store the string numeric.
- If the values assigned to the variables or arrays must vary depending on program conditions, you can select separate groups of DATA statements with the RESTORE statement. This would require several complete sets of simple assignments.
- The syntax of the READ statement is very similar to the READ # statement. You can simulate READ # statements with READ and later convert the READs to READ #s from a file.

Conversions

If you have data in a program variable that is in the wrong form, there are several ways to convert it to more appropriate forms. The simplest is through the use of the conversion functions such as: CHR$, DVAL, DVAL$, NUM, IVAL, IVAL$, VAL and VAL$.

Characters and Codes

You can only assign a character to a string variable if the character is in **string** form. You can only use a character in a mathematical operation if it is in numeric, or character **code** form.

For example, the test in the "Input_kbd" subprogram for RETURN and STEP tested the numeric value of the keycodes with the lines:

```
1100  Chr=NUM(Input$[2;1])             !Get second keycode number
1110  IF (Chr=69) OR (Chr=83) THEN     !Is it RETURN or STEP ?
```

These lines could also be written:

```
1100  Key$=Input$[2;1]                 !Get second keycode char.
1110  IF (Key$=CHR$(69)) OR (Key$=CHR$(83)) THEN
```

String Numerics

There are two principal cases in which it is customary to enter numeric data into a string and convert it to numeric form separately.

You may not be certain of the form of the data. The "Skipping Program Lines - ON...GOTO" section of the "ON... Conditions" chapter has an example of this technique using VAL. You can also read ordinary decimal string numerics from strings with the ENTER statement, for example:

```
630  ENTER "1.2345E7";Value
```

The form of the string numeric may be inappropriate for **any** BASIC input statement. For example, if you are entering or transporting ASCII data in hexadecimal notation, you might have lines of data similar to this:

```
" 7FDEA55F,  39CB2A6, 099906 ,AA67FDE3   "
```

You can enter each line of data into a string with an ENTER or INPUT statement and parse each string by finding the commas with the POS function and converting each string numeric field with DVAL, for example:

```
450  FOR Field=1 TO 4                         !For 4 number record
460    Stop=POS(Data$,",")-1                  !delimited by commas
470    IF Field=4 THEN Stop=LEN(Data$)        !
480    Hex$=TRIM$(Data$[1,Stop])              !Remove blanks
490    Decimal(Field)=DVAL(Hex$,16)           !Convert to decimal
500    IF Field<4 THEN Data$=Data$[Stop+2]    !Throw away hex
510  NEXT Field                               !
```

Input from Files

You can read file data with the READ#, ENTER, ENTERBIN, ENTER USING and TRANSFER statements. READ# accesses files via a file number. The other statements access files via I/O path names. READ#, ENTER and ENTER USING are discussed in the "File Access and Data Transfer" chapter. ENTERBIN and TRANSFER are discussed in the "Advanced I/O Operations" chapter.

Input from Devices

You can use the ENTER, ENTERBIN, ENTER USING and TRANSFER statements to input data from devices such as a CRT SCREEN, a terminal connected to an Asynchronous Serial Interface (ASI) card or an HP-IB instrument. This section discusses considerations relating to the use of ENTER with devices. It deals primarily with the HP-IB interface.

There are six questions that you should ask yourself when performing input from a source other than a file or keyboard.

- How do you address the device? The "Specifying Secondary Addresses" section of the "Introduction to I/O" chapter pointed out that that proper addressing not only establishes a data path from the device to the computer, but may also determine what data is returned.
- Is the device ready for input? Even with keyboard entry, the operator may never respond unless your program outputs a prompt. With some devices, if you do not send certain commands before performing the ENTER, the ENTER may suspend your program indefinitely waiting for data that may never arrive.
- What is the data format? The "Introduction to I/O" chapter demonstrated the effect of FORMAT OFF data output to a FORMAT ON device. The potential errors are similar for input.
- Did you get all the data? Was the computer ready for each byte or half-word? Did any bits or bytes get "lost"? It is not an issue in the examples in this chapter because HP-IB handshaking is virtually foolproof.
- How much did you get? There are consequences if the ENTER statement requests an amount of data that differs from what the device "wants" to send.
- When does it stop? How does the ENTER statement determine that the device has no more to send so that your program can resume execution?

This section addresses these questions by using the HP 2631B printer and HP 7580A plotter of the example system in the "Introduction to I/O" chapter.

Input Addressing

Because the ASI and GPIO interfaces generally support only a single device type at a time, their input addressing requirements are highly device-dependent. HP-IB, on the other hand, has a strict protocol for addressing devices for input. The form of device selector that you specify can significantly affect the input operation.

To input data from an HP-IB device, the bus must be properly **configured**. The device must be addressed as a talker, the computer's HP-IB interface must be addressed as a listener and the ATN (attention) line must be false. For some bus devices, such as test instruments, the bus control line REN (remote enable) must be true. If these conditions are already in effect, you can read data from the device simply by performing an ENTER from the interface select code of the HP-IB interface. This is referred to as "direct bus I/O". Here is an example.

```
240  ENTER 4;Data$          !Assuming the data is in string form
```

It is not generally the case that the bus is already correctly configured for the input you want to perform. No devices, including the HP-IB interface, are addressed until you execute an I/O statement specifying that bus. Most I/O statements which reconfigure the bus leave the bus in the new state on completion, but you should not rely on this. Another program (in a different partition) might reconfigure the bus between your input statements.

To re-configure the bus for your ENTER operation, you normally need only specify a device selector that contains a primary address. Here is the example from the "Introduction to I/O" chapter which requested the model number from the HP 7580A plotter. The code has been changed to show the addressing.

```
20  OUTPUT 405;"OI"           !Send "Output Identity" instruction
30  ENTER 405;Model$[1,5]     !Read response
40  DISP Model$               !Display result
```

The display result should be:

```
7580A
```

This ENTER statement addresses the 7580A as a talker, un-addresses all other devices as listeners, addresses the HP-IB interface as a listener and sets the ATN line false.

Ready for Input?

Simply addressing an interface or device source with an input statement does not insure that the source can actually send any data. For example, a typical HP-IB device requires that you send it instructions in the form of ASCII data or HP-IB bus commands to tell it to send data. Some devices require additional instructions to tell them what kind of data to send. Even simple keyboard input requires a prompt to alert the operator.

The ENTER statement of the previous example is preceded by an OUTPUT statement that tells the 7580A what data to return. The following program segments show additional examples of the "setup" required to read data from devices. These examples use the HP 2631B of the sample system. The 2631B returns a variety of information about its state when sent either escape sequences or secondary addresses.

Many HP-IB devices, like the 7580A, treat all text sent to them as instructions. You can output instructions to them as ordinary text. Other devices, like the 2631B, are designed to display or print character data that you send to them. To send instructions that are to be acted on without printing you have to use a reserved character. For the 2631B, as with most HP display and printing devices, this character is the ASCII control character **escape** (ESC), character code 27.

When the 2631B receives an ESC character it does not print it[1]. It examines one or more subsequently received characters and interprets the sequence as an instruction rather than data to be printed. Some escape sequences tell the 2631B to perform an action or set a mode. Others tell it to return information to the computer. The 2631B also uses the ASCII control character **Device control 1** (DC1), character code 17, as the instruction to "send the data now". As an example, like the 7580A, the 2631B can "self-identify". To accomplish this, use the following technique.

```
740   OUTPUT 406;CHR$(27)&"*rK"&CHR$(17);   !Self-ident now
750   ENTER 405;Model_no$[1,5],Rev_level    !Read response
760   DISP Model_no$;Rev_level              !Display result
```

The resulting display should look something like this.

```
2631B 2046
```

You can also program your 2631B and read information from it by using secondary addressing. The rough equivalent of the self-identify escape sequence is the "universal identify command" which returns two bytes. This command is part of an HP extension to the IEEE-488 specification known as "AMIGO Protocol". The identify command is an HP-IB un-talk (UNT) command followed by the device's primary address sent as a secondary address.

This sequence cannot be sent by an OUTPUT or ENTER statement, and for this reason BASIC provides a SEND statement which gives you virtually direct control of the bus. The following example reads the 2631B's identifying data bytes.

```
740   SEND 4;UNL;MLA;UNT;SEC 6                    !Config, bus, send cmd
750   ENTER 4;B1$[1,1],B2$[1,1]                   !Read 2 identity bytes
760   DISP IVAL$(NUM(B1$),2),IVAL$(NUM(B2$),2)  !Display bytes
```

The resulting display should be:

```
00100000             00001001
```

This response means that the device is a printer or display. Refer to the 2630B Family Reference Manual for more information on programming the 2631B.

1 The 2631B prints **all** control characters if DISPLAY FUNCTIONS mode is enabled.

Data Format

In the previous example you may have noticed that the DISP statement displayed the **binary** representation of the character **codes** of the returned bytes. It did not display them as characters. Although most of the data returned by a 2631B consists of meaningful ASCII characters, the definition of the identity bytes depends on which bits are one or zero in the binary representation of the bytes. The values returned in the example correspond to the ASCII characters SPACE and HT. If line 760 of the example were:

```
760  DISP B1$,B2$
```

the resulting display would be **blank**. All HP-IB devices send data as a stream of 8-bit bytes. For many operations, these bytes represent meaningful characters. If they are string numerics, you can assign them to numeric variables using the number builder parsing of the ENTER statement. The 2631B self-identify escape sequence example did just this.

Other devices, such as the HP 7971A magnetic tape drive, can return INTEGER, DOUBLE, SHORT and REAL data as sequences of bytes representing the internal binary or IEEE form of the values. Data of this kind is meaningless if assigned to a string variable and it cannot be assigned to a numeric variable if FORMAT is ON. The number builder requires string numerics. In this case you must read the data via an I/O path name with FORMAT OFF.

Data Length and Delimiters

The examples in this section all request exactly as much data as the device is sending. If you want to read only the model number from the 2631B and want to ignore the revision level code, what happens if you execute the following statement?

```
750  ENTER 405;Model_no        !Read only model no?
```

The result is that it works. ENTER's number builder treats the "B" in the model number as the end of the numeric field and then reads and discards the revision code. ENTER stops accepting data after the revision code because the 2631B sends a carriage-return character (CR) with the HP-IB EOI line asserted true. EOI is one of two **terminating** conditions recognized by ENTER as denoting the end of data. The other condition is the **DELIM** character.

Each I/O resource has an I/O attribute called **DELIM**. During ENTER, the detection of a DELIM character or EOI terminates an item. After the last item, it terminates the ENTER statement. There is a relationship between DELIM/EOI and the amount of data that your ENTER statement requests.

- The ideal situation is for the DELIM to occur just after the last data byte, or for EOI to occur with the DELIN. No error occurs and no data is lost.
- You can request less data than the device actually sends. After satisfying your last ENTER variable, ENTER continues reading and discarding data for up to 256 bytes waiting to detect the DELIM or EOI. If it does not detect either, an error occurs.
- If you request more data items than the device sends, the DELIM or EOI terminates the current item. If the device never sends any more data, ENTER may wait indefinitely.

For most devices, the default DELIM is the ASCII line-feed (LF, CHR$(10)) character. The 2631B delimits responses to escape sequences with an ASCII carriage-return (CR, CHR$(13)) character sent with EOI TRUE. Its responses to secondary addressing commands are sent with EOI asserted true with the final data byte. The 7580A sends both CR and LF. In the 2631B examples, ENTER detected EOI. In the 7580A example ENTER detects LF.

In those cases where the devices send both a CR and LF character, you need to avoid reading the CR as a data character. The 7580A example read the "Model_no$" as a substring, ignoring the CR. The 2631B escape sequence example read the "Rev_level" into a numeric variable. The number builder ignored the CR.

For the default DELIMs of other I/O sources see the "I/O Resources" appendix in the BASIC Language Reference manual.

If you perform ENTER via an I/O path name, you can change the default DELIM with a DELIM I/O attribute expression in the ASSIGN @ statement. The ENTER USING statement's IMAGE specifiers can change the way in which ENTER USING responds to DELIM. The ENTERBIN statement is effectively independent of DELIM conditions. The TRANSFER statement specifies its own. These statements are discussed in subsequent chapters.

Chapter 17
The Image Specifier

This chapter discusses the use of a string expression known as the image[1] specifier in input and output statements.

On output, the image specifier provides full control of the form of the data representation. You are not limited to the forms discussed in the "Introduction to Output" chapter, those provided by FIXED, FLOAT and STANDARD. The image specifier also provides some control over the EOL sequence.

On input, the image specifier allows you to specify your own parsing rules. You are not restricted to the semantics of the "number builder" used by the statements discussed in the "Introduction to Input" chapter. The image specifier also provides some control over item and statement terminating conditions.

Operations Using an Image Specifier

You can use an image specifier in the following I/O operations.

To perform this operation:	Use this statement:
Output to the DISPLAYS SCREEN.	DISP USING
Output to the standard printer.	PRINT USING
Output any I/O resource.	OUTPUT...USING
Output labels to a graphics plotter.	LABEL USING
Input from and I/O resource.	ENTER...USING

These statements are similar in some ways to their counterparts which do not have the secondary keyword "USING". They are sufficiently different that the manuals treat them as separate statements. For example, the syntax of a "USING" statement does not support any output functions, treats both comma and semicolon delimiters similarly and does not permit trailing delimiters.

1 The concept of specifying an "image" has no relation to the IMAGE/9000 Data Base Management option.

Expressing the Image Specifier

There are four ways to reference an image specifier in an I/O statement. In the following example, each PRINT USING statement displays the identical result. The braces identify the specifier.

```
420   PRINT USING """Code is"",X,ZZZ";NUM("@")    !In-line literal
      .
      .
      .
450   String$="""Code is"",X,ZZZ"
460   PRINT USING String$;NUM("@")                !In_line expression
      .
      .
      .
490   PRINT USING 500;NUM("@")                    !IMAGE line number
500   IMAGE "Code is",X,ZZZ
      .
      .
      .
530   PRINT USING Label;NUM("@")                  !IMAGE line label
540 Label:IMAGE "Code is",X,ZZZ
```

The displayed result is:

```
Code is 064
Code is 064
Code is 064
Code is 064
```

You can express the image specifier in the I/O statement as a string expression. If it is a literal, it is a **quoted** literal. As with any string literal, any quoted literals **within** a literal specifier must have an extra quote character for each quote desired. If the image specifier expression is not a literal, string variables in the expression must be defined by assignment statements executed prior to the I/O statement.

The advantage of using a string expression is that you can construct the image specifier while your program is running.

You can also reference an IMAGE statement containing an image specifier. The IMAGE statement is non-executable, so it can be in any program line of the same context[2]. It need not precede any referencing I/O statement. The image specifier in the IMAGE statement is an **unquoted** literal. Literals within the specifier need only single enclosing quotes. The specifier in an IMAGE statement cannot be a string expression.

The advantage of using a literal image or IMAGE statement is that it is compiled at pre-run, rather than during execution, resulting in better performance. Another advantage of the IMAGE statement is that you can place frequently used images all in one block of statements. This eases program maintenance if you have to change the specifiers.

2 Since an IMAGE statement is not executable, it cannot be in an IF...THEN statement.

Contents of the Image Specifier

The smallest unit of an image specifier is an item; an item specifies a single action. The following table summarizes the specifier items. The notation [n] indicates an item which allows a repeat factor, discussed in "Repeating Items and Fields".

Item	Purpose on output	Purpose on input
"	Encloses literals to output	Each character within quotes treated as X specifier
#	Suppress automatic EOL	End input when last variable is satisfied
%	Ignored	End input on END condition
[n]*	Leading "*" in numeric field	Same as D specifier
+	Automatic EOL is CR	Terminate only on END after last input item
–	Automatic EOL is LF	End input only on DELIM after last input item
.	Decimal point radix indicator	Same as D
/	Send CR/LF now	Skip input until next LF
[n]@	Send form-feed character now	Ignored
[n]A	Send a character	Demand a character
B	Send 8-bits of numeric value as one byte	Assign byte numeric value to numeric variable
C	Conditional comma in numerics.	Same as D on input
[n]D	Send a digit character	Demand a digit character
E	Same as ESZZ	Same as DDDD
ESZ	Send "E" and explicitly signed 1-digit exponent	Same as DDD
ESZZ	Send "E" and explicitly signed 2-digit exponent with conditional leading zero	Same as DDDD
ESZZZ	Same as ESZZ but with 3-digit exponent	Same as DDDDD
EZ	Send "E" and implictly signed 1-digit exponent	Same as DD
EZZ	Send "E" and implicitly signed 2-digit exponent with conditional leading zero	Same as DDD
EZZZ	Same as EZZ but with 3-digit exponent	Same as DDDD
H	Same as K but with comma as radix point	Same as K, but European notation
–H	Same as H	Same as H but without DELIM processing
K	Numerics sent in STANDARD string numeric form	String numerics parsed via ENTER's number builder
–K	Same as K	Same as K but without DELIM processing
[n]L	Send automatic EOL now	Ignored
M	Send space or minus sign	Same as D
P	European "." numeric digit separator	Same as D
R	European "," radix point	Same as D, but "," is radix
S	Send explicit numeric sign	Same as D
W	Send number as an integer in two 8-bit bytes; skip to next half-word if required to align	Assign two 8-bit bytes to a numeric variable as an INTEGER, skip if required
[n]X	Send space character	Skip one byte
Y	Same as W without skip	Same as W, but without skip
[n]Z	Leading "0" in numeric field	Same as D

Field Specifiers

Many specifier items can be combined into a **field specifier**, such as `3DC3D.DD` or `3AXA`, which specifies a set of operations for a single output expression or input variable. There are four types of fields.

- String fields specify processing of character data input to a string variable or output from a string expression.
- Numeric fields specify processing of character data input to a numeric variable or output from a numeric expression.
- Binary fields specify processing input of an 8 or 16-bit binary value to a numeric variable or output of a numeric expression as an 8 or 16-bit binary value.
- Control fields specify a mode for the entire operation, cause skipping of input data or provide a literal to be output directly from the image specifier.

Repeating Items and Fields

For some items which appear more than once within a field, you can literally repeat them, such as `DDDD`, or you can use a repeat count, such as `4D`. Items which support a repeat count are:

* / @ A D L X Z

You can repeat an entire field or set of fields with a repeat count by enclosing the field in parentheses, such as `3(4D)`. The parentheses amount to an implied delimiter between the repeated fields. For example `5D` is the same as `DDDDD` and is a **single** field.

`5(D)` is equivalent to `D,D,D,D,D` and is **five** fields. For items which are always individual fields, such as `/`, the method of repetition has no side effects. `5/`, `5(/)`, `/////` and `/,/,/,/,/` are identical.

Field Delimiters

Specifier fields must be separated from adjacent fields by a delimiter, normally a comma. For compatibility with earlier HP computers, the `/` and `@` items are also delimiters, but it is suggested that you use only the comma.

Literals do not need to be separated from adjacent fields by a delimiter. If a literal precedes a numeric field specifier without a delimiter, the literal can "float" into the numeric field. See the discussion of "Literals and Numeric Fields" in this chapter.

Syntax Errors

When entering or editing a program using the Editor, an image specifier is checked only to insure that it is an otherwise valid literal or string expression; it is not checked to insure that it is a valid specifier.

Image specifiers are checked as specifiers during execution. A BASIC error occurs if the specifier is illegal, such as `"DD.T"` or if a field is not the same data type (string/numeric) as the expression or variable to which it is matched.

Other kinds of errors, such as a numeric value too large for a numeric field, are processed without causing a BASIC error. These cases are discussed in the following sections.

Image Specifiers and FORMAT OFF

The output data sent by and input data expected by statements using an image specifier is normally a stream of 8-bit bytes, typically characters. These statements cannot output or input the internal data form required by some I/O resources. Therefore, I/O using an image specifier is always **FORMAT ON**.

To output to a destination which supports only FORMAT OFF, such as an ASCII file, requires a two-step operation:

1. Output the data to a string (FORMAT ON) with the OUTPUT...USING statement, for example:

```
410   OUTPUT String$ USING Specifier;Data,Value,Misc
```

2. Output the string to the FORMAT OFF destination, for example:

```
 20   ASSIGN @File TO "AFILE:INTERNAL"   !Open file, FORMAT OFF
      .
      .
      .
420   OUTPUT @File;String$               !Send string
```

Output Using an Image Specifier

Like the simple DISP, LABEL, OUTPUT and PRINT statements, the DISP USING, LABEL USING, OUTPUT...USING and PRINT USING statements output data in character (byte) form[3]. Like the simple statements, the "USING" statements send the current EOL sequence after the last image or data item. Unlike the simple output statements, the "USING" statements have no default format for their output. You must specify the form desired.

When you execute an output statement using an image specifier, the statement begins "scanning" the evaluated specifier from left to right. For each field which requires variable data, the statement accesses the next item in the output statement's list of expressions. If an image item requires no variable data, such as a literal, the statement acts on it or outputs it directly.

Output ceases on two conditions:

- when the end of the entire image specifier is encountered and all output list expressions have been accessed;
- when a field specifer which requires variable data is encountered after the output list is exhausted.

If the image specifier is exhausted before the output list expressions, the image specifier is **re-used** from the beginning. There is an example of this in the "Variable-Width Numeric Fields" section of this chapter.

3 The W and Y specifiers are exceptions; they processes 16-bit values.

A Word about the Examples

The examples in this section show each output byte in a separate box. If a box contains two or more characters, those characters are the ASCII abbreviation for a control character. For example, FF is the abbreviation for form-feed, CHR$(12). An exception is the abbreviation "eol" which stands for the current end-of-line sequence. EOL is normally CR LF, but its default value depends on the type of I/O resource and can be modified if the resource is an I/O path name, the standard printer or PRINT ALL printer.

Although the illustrations may require more than one line, the appearance of the output on a printer or display is generally a single line, depending on how the device responds to the control characters sent to it.

Numeric Output Fields

There are fixed-width and variable-width numeric fields. A **fixed-width** numeric field consists of at least one *, D or Z digit specifier item and can include additional specifiers for sign, digit separators, exponent, radix, blanks and floating literals. When a numeric field is processed by the output statement, it uses the value of the next numeric expression in the output statement's list. If the next expression is not numeric, an error occurs. If there are no more expressions in the list, the output statement terminates and your program continues execution.

Your output statement always sends one character for each fixed-width[4] specifier item, so the width of the field is a constant. The format of the data within the field depends on the specifiers you select and on the value of the numeric expression from the output list.

A **variable-width** field consists of a single K, -K, H or -H item. Each field represents a single output list expression. Variable-width fields are described in the section following the discussion of the fixed-width items.

Numeric Digits (Specifier D)

Each D specifier causes the printing of one significant digit of the numeric value. If the field specifier contains no * or Z specifiers, leading zeros are output as blanks. If the value is negative, the minus sign character occupies one of the D positions. The value of the output list expression is rounded to fit the number of digits specified in the field. Here are two examples using the D specifier.

```
120  PRINT USING "4D";1234.012
```

1	2	3	4	eol

```
130  PRINT USING "DDDD";-1.8
```

		–	2	eol

4 The E specifier is treated as ESZZ and sends four characters.

Leading Zeroes (Specifier Z)

If you want leading zeroes in your output fields, you can replace one or more leading `D` specifiers with a `Z` specifier. Each `Z` represents a character position which can be occupied by a significant digit, a minus sign, or a leading zero. Here is an example using the `Z` specifier.

```
234   PRINT USING "4Z";-3.7
```

–	0	0	4	eol

An Alternate Fill Character (Specifier *)

If you want to output numeric fields that cannot be altered on the printing medium, such as on a bank check, you can replace one or more leading `D` specifiers with a `*` specifier. Each `*` represents a character position which can be occupied by a significant digit, minus sign or leading "*" character. Here is an example using the `*` specifier.

```
560   PRINT USING "4*";23
```

*	*	2	3	eol

Output of Numeric Signs (Specifiers M and S)

In numeric fields beginning with `D`, `Z` and `*` specifiers, a negative number has a leading minus sign and a positive number has only a leading blank, digit, zero or asterisk. If you want positive numbers to have a leading blank or an explicit plus sign, use the `M` or `S` specifiers. Here are examples of both the `M` and `S` specifiers.

```
196   PRINT USING "M4Z.DD";12.99
```

	0	0	1	2	.	9	9	eol

```
673   PRINT USING "S5D";84.3
```

			+	8	4	eol

In the "`S`" example notice that a six character field was specified, but the plus sign was not output as the first character. Instead it appears as the fourth character. Signs "float" into numeric fields and the sign is always output to the immediate left of the first digit of the number, unless "blocked" by a `*` or `X` specifier or a literal. Literals can also "float" into numeric fields. See the "Literals and Numeric Fields" section of this chapter.

Output of Radix Indicators (Specifiers . and R)

As you can see from the previous examples, numeric fields composed solely of `D`, `Z` and `*` specifiers can output only the integer representation of numeric values. To output any fractional part of a SHORT or REAL value, you must specify a radix indicator.

The radix indicator specifies both the location and the appearance of the decimal point within a numeric field. The period outputs a period. The `R` outputs a comma, which is frequently used in European numeric notation. Here are examples of the `.` and `R` specifiers.

```
324   PRINT USING "4Z.DD";23.469
```

0	0	2	3	.	4	7	eol

```
450   PRINT USING "****RDDD";8.123456
```

*	*	*	8	,	1	2	3	eol

Only one radix indicator can be specified in a single numeric field.

Output of Digit Separators (Specifiers C, P and X)

In the display of numeric values which contain many digits it is customary to break the number into groups of three digits separated by a non-numeric character. In the United States this separator is normally a comma or space. In Europe it is frequently a period, particularly if the comma is already being used as the radix indicator.

The `C` specifies a conditional comma. The `P` specifies a conditional period. `X` specifies a space. The `C` and `P` are conditional in that they are not output unless there is a digit in both adjacent character positions; a space or "*" is output. Here are examples of the `C`, `P` and `X` specifiers.

```
430   PRINT USING "3*C3*C3*.DD";1764.18
```

*	*	*	*	*	*	1	,	7	6	4	.	1	8	eol

```
690   PRINT USING "3DP3DP3DR3D";34982.1757
```

					3	4	.	9	8	2	,	1	7	6	eol

```
110   PRINT USING "3*X3*X3*.DD";-9276.31
```

–	*	*		*	*	9		2	7	6	.	3	1	eol

The `X` is not conditional. Each `X` always outputs one space. `X` also "blocks" characters or fields which would otherwise float into the numeric field.

Output of Numeric Exponents (Specifiers E, ESZ..ESZZZ and EZ..EZZZ)

To output numeric values in scientific notation, add an exponent specifier to the end of a numeric specifier. At least one `D`, `Z` or `*` specifier (for the mantissa) must precede an exponent specifier. The exponent specifier can describe an exponent having one, two or three digits with an implicit or explicit sign. As with the numeric specifier for the mantissa, if you do not use an explicit `S` exponent sign specifier, there must be enough digits in the exponent field for a minus sign. You cannot specify an `M` item in an exponent field.

The simple `E` specifier is provided for compatibility with earlier implementations of HP BASIC. It is equivalent to `ESZZ`. Here are two examples of exponent specifiers.

```
780   PRINT USING "DDD.3DEZZZ";PI
```

3	1	4	.	1	5	9	E	–	0	2	eol

```
610   PRINT USING "SZ.5DESZZ";PI*1E11
```

+	3	.	1	4	1	5	9	E	+	1	1	eol

Numeric Field Overflow

With any fixed-width field, if the value cannot be represented in the field, the output statement sends the field filled with asterisks followed by an ASCII BEL character, three spaces, the value in STANDARD format and an EOL. This is in addition to the normal automatic EOL. Subsequent fields from the same statement appear on a new line. The ASCII BEL character beeps the tone generator to alert the operator to the error. No other error processing occurs. Here is an example of overflow.

```
140   PRINT USING "3D";-123.400
```

*	*	*	BEL				–	1	2	3	.	4	eol	eol

A typical cause of overflow is the failure to include an extra `D`, `Z` or `*` item for the minus sign when no `M` or `S` is specified.

Variable-width Numeric Fields (Specifiers K,-K,H and -H)

These specifiers are single item fields. The numeric value output is sent in STANDARD representation. They are useful for numeric values which may have unpredictable values. You cannot control the width of these fields since they depend on the value of the numeric expression.

The following example shows both the `K` specifier and the concept of re-use of the image specifier.

```
980   PRINT USING "K";123,-4.5600,-7E89
```

1	2	3	–	4	.	5	6	–	7	.	0			
0	0	0	0	0	0	0	0	0	0	E	+	8	9	eol

If `K` is the most appropriate specifier for your data, consider using the simple output statements. The output from the above example is identical to that from the following statement, except that the simple PRINT sends a space after a numeric field and before a positive numeric field.

```
990   PRINT 123;-4.5600;-7E89
```

	1	2	3		–	4	.	5	6		–	7	.		
0	0	0	0	0	0	0	0	0	0	0	E	+	8	9	eol

The `-K` specifier is identical to `K` on output. The `H` and `-H` specifiers are identical to `K` on output except that the decimal point is sent as a comma (European notation).

String Output Fields

You can output strings as literals in the specifier or as string expressions using an `A` or `K` specifier. You can also output single numeric codes as characters using the `B` specifier. `B` is discussed in the "Binary Output Fields" section.

Output of Literals

You can output literals as data or as a specifier field. The following PRINT USING statements have identical effect.

```
110   PRINT USING """Text"""        !Literal in specifier
120   PRINT USING "4A";"Text"       !Literal as data
```

T	e	x	t	eol

Literals and Numeric Fields

If you specify a literal **in** the image specifier (not in the output list) and the literal precedes a D specifier without an intervening delimiter, the literal can float into the numeric field. Here are examples of both floating and non-floating literals.

```
10 Floating:IMAGE "$"6D.DD
20  PRINT USING Floating;419.34
30  !
40 Non_float:IMAGE "$",6D.DD
50  PRINT USING Non_float;419.34
```

			$	4	1	9	.	3	4	eol
$				4	1	9	.	3	4	eol

Fixed-width String Output Fields (Specifier A)

The `A` specifies a character position which can be occupied by a data character or a statement-supplied blank. The evaluated string expression is left-justified in the `A` field. If the string is shorter than the field, the field is blank filled (CHR$(32) character). If the string is longer than the field, the trailing (rightmost) characters of the string are not output and no error occurs. The following example shows all three cases.

```
430   IMAGE 2A,"-"
440   PRINT USING 430;"","X","XY","XYZ"
```

		–	X		–	X	Y	–	X	Y	–	eol

Variable-width String Output Fields (Specifiers K, -K, H and -H)

You can use these specifiers for string as well as numeric data. They are treated identically for string data. Strings are sent exactly as evaluated with no leading or trailing blanks. This is similar to the simple print statement. The following two statements have the same effect.

```
420   PRINT USING "K";"Text"&CHR$(32),"sample"
430   PRINT "Text"&CHR$(32);"sample"
```

T	e	x	t		s	a	m	p	l	e	eol
T	e	x	t		s	a	m	p	l	e	eol

Binary Output Fields

The numeric field specifiers imply that numeric values are converted to their string decimal character representations. If you are sending an INTEGER value to a GPIO interface you may not want this conversion to occur. Likewise if you have a numeric variable containing a character code and you want to output the character, it is an extra step to convert it to string form with CHR$ and then output it in an A field.

This section discusses the B, W and Y specifiers, which allow you to send 8-bit byte or 16-bit INTEGER numeric values without conversion to character. If you need to output only a single value, and without eol, you can also use the OUTPUTBIN statement.

Binary Byte Output (Specifier B)

This specifier is intended primarily for use with 8-bit interfaces or where the data already represents Series 500 character codes. As such, the data should be in the range 0 through 127 for the ASCII character set, 0 through 255 for the Series 500 character set, and −127 through 255 for an interface.

Values less than −32 768 are set to 0. Values greater than 32 767 are set to 255. After this check the result is evaluated BINAND(Result,255) and sent to the destination. External printers and displays may interpret values in the range 128 through 255 as characters from an extension character set, as ASCII characters with parity or as blanks.

If the numeric value is in the range −127 through 255, output of the value using a B specifier is equivalent to the output of CHR$(Value) using an A specifier, as in this example.

```
660  IMAGE "Code ",3Z," = ",B
670  PRINT USING 660;38,38
```

C	o	d	e		0	3	8		=		&	eol

Binary INTEGER Output (Specifiers W and Y)

These specifiers are intended for use with 16-bit interfaces such as the HP 27112A GPIO card, or other destinations such as BUFFERs and BDAT files, which can accept data in 16-bit INTEGER units. If the numeric value sent with a W or Y specifier is less than −32 768, it is set to −32 768. If it is greater than 32 767, it is set to 32 767.

The Y specifier sends the value as two 8-bit bytes. The W specifier sends the value as a single 16-bit INTEGER (most significant byte first). If an odd number of bytes have been output by the statement since it started or since the last EOL within the statement, W "aligns" the output on a even byte boundary by padding the output with a preceding (all-zeroes) byte, an ASCII NUL character.

```
660  DIM Buf$[10] BUFFER
670  ASSIGN @Buf TO BUFFER Buf$
680  OUTPUT @Buf USING "Y,X,W,W";17970,18260,21070
690  PRINT Buf$[1,8]
```

F	2		NUL	G	T	R	N	eol

Output Control Specifiers

The DISP USING, LABEL USING, OUTPUT USING and PRINT USING statements do not have the output functions of their simple counterparts, such as SPA, LIN and PAGE. There are instead image specifiers that provide similar, if not identical, control of the current output position.

Horizontal Forms Control

There are three ways to move the current output position in the same line; spaces, tabbing and direct cursor control. In the simple output statements this control is provided by the SPA, TAB and TABXY output functions. Output statements using an image specifier must use the X specifier, ASCII control characters and escape sequences.

Output Spacing

You can output any number of spaces with literal spaces, the CHR$(32) expression or the X image specifier. With some destinations, such as the internal printer and display, you can backspace by sending ASCII BS (CHR$(8)) characters. For multiple spaces you can use a repeat factor with the X specifier. The following statements illustrate spacing.

```
930   PRINT SPA(4);"Text"
940   PRINT USING "4X,4A";"Text"
```

				T	e	x	t	eol
				T	e	x	t	eol

If you send backspace characters to the ALPHA display of the internal CRT, later characters entirely replace earlier ones. If you send backspace characters to the internal printer or GRAPHICS display, later characters overprint earlier ones. External devices may exhibit either behavior, perform no operation or print a visible character. This example uses the internal display.

```
590   DISP USING "DDD,A,DDD";123,CHR$(8),456
```

Output sent by statement:

1	2	3	BS	4	5	6	eol

Resulting display:

1	2	4	5	6

Output Tabbing

The TAB function of the simple output statements does not actually perform a tab operation in the same sense as a typewriter. It does not move the current position to the location of a tab "stop". It simulates tabbing by sending enough spaces to move the current position to the designated column.

There is no image specifier which is the direct equivalent to the TAB function. You can use spacing or you can use tab stops. Setting and clearing these tabs was discussed in the "Introduction to Output" chapter. Most external HP printers and terminals also have tab stops and support the same programming escape sequence to set and clear the tabs.

If you set tab stops, you can move the current position to any tab stop by output of the appropriate number of forward or reverse tab sequences. The current position advances to the next tab position when you output an ASCII HT character, CHR$(9). The position moves to the previous tab stop when you output the escape sequence CHR$(27)&"i". If there are no tabs set, the result is device dependent.

The following example clears all the tabs on the internal display, sets a tab in column 7 and illustrates output using a tab stop.

```
10   Esc$=CHR$(27)                     !ASCII escape character
20   Ht$=CHR$(9)                       !ASCII tab character
30   IMAGE A,"3",6X,A,"1"              !Clear all tabs, set one
40   OUTPUT CRT USING 30;Esc$,Esc$;    !Send the escape sequences
50   IMAGE ZZ,A,DDD                    !
60   OUTPUT CRT USING 50;4,Ht$,123     !Send 4,Tab,123
70   END
```

Output sent by DISP:

0	4	HT	1	2	3	eol

Resulting display:

0	4					1	2	3

Direct Cursor Control

The TABXY function of the simple output statements sends an HP standard escape sequence to change the current position. In an output statement using an image specifier, you can send this same escape sequence as a string expression. You can also send a large variety of other escape sequences which provide various cursor movements. See the "Escape Sequence Effects" table in the Useful Table section of the BASIC Language Reference manual.

Here is an example of a string function which produces escape sequence data that can move the CRT cursor in the same way that the TABXY output function does. FNTabxy$ is not identical to TABXY because it does not range-check the parameters, and FNTabxy$'s output is counted in WIDTH calculations; TABXY's is not.

```
1000  DEF FNTabxy$(Column,Row) !Return TABXY Escape Sequence
1010    Sequence$=""                              !Assume column & row = 0
1020    Prefix$=CHR$(27)&"&a"                     !ESC & a sequence
1030    IF Column THEN Column$=VAL$(Column-1)&"c" !##c column portion
1040    IF Row THEN Row$=VAL$(Row-1)&"y"          !###y row portion
1050    IF Row OR Column THEN                     !If at least one <>0...
1060      Sequence$=Prefix$&Row$&Column$          !Build escape sequence
1070      Last=LEN(Sequence$)                     !Find last character
1080      Sequence$[Last;1]=UPC$(Sequence$[Last]) !Make it uppercase
1090    END IF                                    !
1100    RETURN Sequence$                          !Return null or sequence
```

Here is an example of how to use this function.

```
400   IMAGE K,"Column",3D,", Row",3D
410   OUTPUT CRT USING 400;FNTabxy$(30,12),30,12
```

Vertical Forms Control

There are four ways to move the current output position to a different line: the EOL specifiers, vertical tabbing, control characters and escape sequences. The image specifiers related to vertical forms control are `#`, `+`, `-`, `@`, `/` and `L`.

Note that `#`, `+`, `%` and `-` override each other. The last (rightmost) specified prevails. You should specify these terms at the beginning of the image.

The EOL Sequence

All of the output statements which use an image specifier automatically send the current EOL sequence after the last data item. You can find the default EOL for each I/O destination in the "I/O Resources" appendix of the BASIC Language Reference manual. If the destination is accessed via an I/O path name or is the standard printer or the PRINT ALL printer, you can change the default EOL. Regardless of the value of the EOL, some of the image specifiers can suppress it, explicitly send it or send another sequence in its place.

Suppressing EOL

Unlike the simple output statements, the output statements which use an image specifier do not allow a trailing delimiter in their syntax. To suppress the automatic EOL sequence, use the `#` specifier, as in the following example.

```
190  PRINT USING "#,4A";"Text"
```

T	e	x	t

Changing the EOL Sequence

The image specifier provides a limited amount of control over the content of the EOL sequence. The `+` and `-` specifiers change the current EOL to a simple CR or LF.

The `+` specifier is generally appropriate for printers and terminals that are configured for "auto line-feed" mode. Such devices supply their own LF character whenever they receive a CR. The `-` specifier is frequently appropriate for HP-IB instruments that recognize LF as an end-of-message and might treat the preceding CR as an invalid instruction. The following examples show both `+` and `-`.

```
220  PRINT USING "4A";"Text"
230  PRINT USING "+,4A";"Text"
240  PRINT USING "-,4A";"Text"
```

T	e	x	t	eol
T	e	x	t	CR
T	e	x	t	LF

Extra EOL Sequences

The `/` and `L` specifiers output an EOL sequence as they are encountered during the execution of the statement. The `L` specifier issues the current EOL sequence and is equivalent to the LIN function of the simple output statements. The `/` specifier issues a CR LF, regardless of the current EOL definition.

The following example re-defines the default EOL sequence to show the difference between L and /.

```
30   ASSIGN @Prt TO PRT;EOL CHR$(30) !EOL is ASCII Record Separator
40   OUTPUT @Prt USING "2A,2A";"ab","cd"
50   OUTPUT @Prt USING "2A,L,2A";"ab","cd"
60   OUTPUT @Prt USING "2A,/,2A";"ab","cd"
70   OUTPUT @Prt USING "#,2A,/,2A";"ab","cd"
```

a	b	c	d	RS

a	b	RS	c	d	RS

a	b	CR	LF	c	d	RS

a	b	CR	LF	c	d

You can also skip lines by sending control characters as string data. For the internal CRT and printer see the "Control Character Effects" and "Escape Sequence Effects" tables in the Useful Tables section of the BASIC Language Reference manual.

Page Control

You can move the output position to a new page with the @ specifier. This is the equivalent of the PAGE function of the simple output statements. The following example shows the typical use of @.

```
730   PRINT USING "+,K,@";"End of report"   !CR only as eol.
```

E	n	d		o	f		r	e	p	o	r	t	FF	CR

If you want the output to stop at the beginning of a new page, suppress EOL so you won't get a new page plus one line. If printing on an external non-HP printer, it is a good idea to send a CR after the FF to insure that the printhead (output position) is returned to column one.

Some external printers support control instructions that provide more extensive vertical forms control (VFC). The HP 2631B, for example, has VFC escape sequences for partial page skips and vertical pitch.

Output END

No image specifiers provide any control over output END conditions, such as EOI on HP-IB. Use the END syntax element to assert an END condition in the OUTPUT...USING statement. You can also specify that an END is to be asserted with EOL via the EOL...END I/O attribute. Place END at the end of the image specifier.

Input Using an Image Specifier

Of the numerous input statements available, only the ENTER...USING statement supports the use of an image specifier. Here are some applications in which you can use ENTER...USING.

- The input data consists of numerous fields of characters and has no delimiters.
- The numeric data is in European notation or has other non-numeric characters.
- You can't use a simple ENTER statement because it reads and discards valuable data while scanning for the DELIM or END.
- The data contains no DELIM character.
- You want input to terminate on an END condition only.
- The data is in a binary form which cannot be assigned to a string or converted to a numeric value.

Item and Statement Termination

The ENTER...USING statement normally ceases processing data for the current variable in the ENTER list (item termination) on one of three conditions:

- characters are received beyond the width of the variable's fixed-width field specifier;
- detection of the current DELIM character for a variable-width field;
- detection of an END condition for a variable-width field;
- entry of characters beyond the length of the variable during variable-width field entry.

ENTER...USING normally ceases input altogether (statement termination) when the last variable in the list has had a value assigned to it and an END or DELIM is detected within 256 bytes of the last byte required by the image specifier. See the "Input without Terminator" section.

Numeric Input Fields

You can use any of the output numeric specifiers in a numeric input field. Except for the `R` radix indicator, the `X` and literals, the numeric specifier characters are generally treated like the `D` specifier. This means that you can generally use the same image specifier for both input and output of a given data item.

The ENTER...USING statement expects one byte or character for each character in the numeric field. It uses a number building algorithm similar to that used by the simple ENTER statement.

In the following discussions the term "skipped" means that the ENTER...USING statement reads and counts but otherwise ignores the character.

Fixed-width Numeric Input Fields (Specifiers D, Z, *, M, S and Exponent)

Like a numeric output field, a numeric input field consists of at least one `D`, `Z` or `*` character. The sign and exponent specifiers are all treated as `D` by the ENTER...USING statement. If the entire field is composed of the above specifiers, the choice of specifiers affects only the width of the field. Radix, separator, space and literal specifiers can affect the interpretation of the entered data. The field must have at least one `D`, `Z` or `*`.

A `D` (or equivalent) item in an input field "demands" a digit character. The ENTER statement expects the entry of one numeric character. Numeric characters are the digits, 0 through 9; the decimal point, .; signs, − and +; and the exponent letters, E, D and L. The character is used to build the value of the numeric variable. Non-numeric characters are ignored. A sign is ignored unless it precedes all digits.

Here is an example of simple numeric input.

```
510  ENTER Source USING "SDDD";Value
520  PRINT Value
```

The following are examples of the data which "Source" might supply in response to the ENTER...USING:

+	1	2	3	delim

1	2	3	E	0	4	delim

1	2	3		delim

	1	2	3	4	delim

1	2		3	delim

1	,	2	3	delim

1	2	3	.	delim

1	2	R	3	delim

In each case the numeric value "built" from the responses is the same and the PRINT statement outputs the following result.

	1	2	3	eol

The SDDD specifier uses exactly four characters of the response to build the value. Non-numeric characters are skipped. The radix indicator, if any, must be a decimal point. Excess radix indicators are skipped.

Radix and Separator Input Specifiers (Specifiers ., R, C and P)

These specifiers are treated as D. The R and P specifiers force the interpretation of the string to European notation. (Periods are skipped, and the comma becomes the radix indicator.)

Skipping Characters within Numeric Fields (Specifier X and literals)

Each entered character in a position corresponding to an X or literal character is skipped. Here is an example using X.

```
380   ENTER KBD USING "DXDXD";Number
390   PRINT Number
```

The following keyboard input...

1	2	3	4	5	RETURN

...prints the following output.

	1	3	5	eol

Variable-width Numeric Input Fields (Specifiers K, -K, H, -H)

These are the "free-field" specifiers. They signify that the numeric variable is to be satisfied by input and conversion of a variable number of characters. The number building rules are the same as for the simple ENTER statement. In fact, the following statements have identical results.

```
910   ENTER Source;Value,Number
920   ENTER Source USING "K";Value,Number
```

The H specifier is the same as K except that European notation is used. Periods are item terminators, and the comma is the radix indicator.

The -K and -H specifiers turn off DELIM processing during input for the associated string variable. This generally has no effect on the number builder unless you specified a numeric character as the DELIM character in the DELIM I/O attribute.

String Input Fields

You can input characters and assign them to a string variable with the `A`, `K` and `-K` (or `H`, `-H`) specifiers. You can skip entered characters with the `X` specifier or a literal.

Fixed-width String Input Fields (Specifier A)

The `A` specifier signifies that one input character is assigned to a position in a string variable. An `A` field should be the same size or smaller than the length of the string. It can be wider than a substring in which case the extra characters are input and discarded. If the DELIM character is entered during an `A` field, it is treated as data. You can also get other unexpected characters in `A` fields, as in the following example.

```
330   ENTER KBD USING "AX4A";Char$[1,5]
340   PRINT Char$
```

The following keyboard input

a	b	c	d	e	RETURN

prints the following result.

a	c	d	e	CR	eol

The keyboard's RETURN key simulates a CR and LF. Because only five data characters were entered when six were expected, the ENTER...USING statement reads the CR as a data character.

Skipping Characters in String Input

As with numeric fields, any `X` specifiers or literals in a string field cause the specified number of characters to be read, counted and ignored by ENTER...USING. The effect of an `X` is shown in the previous example.

Variable-width String Input Fields (Specifiers K, -K, H, -H)

Like numeric entry with `K`, string entry with `K` effectively causes ENTER...USING to use the input rules of the simple ENTER statement. The number of characters read and assigned to the string variable is determined by the length of the string or substring and the entered data. The following statements have the same result.

```
910   ENTER Source;Text$[1,4],Chars$
920   ENTER Source USING "K";Text$[1,4],Chars$
```

The `H` specifier is the same as `K` for string input. The `-K` and `-H` specifiers cause the DELIM character to be stored in string variables as a data character and does not terminate the string.

Binary Input Fields

The numeric field specifiers imply that input characters are converted to the numeric value they represent. If you are reading an INTEGER value from a GPIO interface you may not want this conversion to occur. Likewise if you are reading characters and want to store their character codes in numeric variables, it is inconvenient to enter the characters in a string and convert them to code form with NUM. This section discusses the `B`, `W` and `Y` specifiers which allow you to read byte or half-word numeric values without conversion to character.

Binary Byte Input (Specifier B)

This specifier is intended primarily for use with 8-bit interfaces or where the data already represents Model 520 character codes. The data is always returned as a numeric value in the range 0 through 255.

Here is the secondary addressing example from the "Ready for Input?" section of the "Introduction to Input" chapter. The example has been changed; note that it requires simpler statements by using the B specifier. It also avoids a potential "bug" in the original example; if either of the returned identity bytes is a decimal 10 (the LF DELIM character), the `ENTER 4;B1$[1,1],B2$[1,1]` of that example would store a blank in the associated string.

```
740   SEND 4;UNL;MLA;UNT;SEC 6     !Configure bus and send command
750   ENTER 4 USING "B,B";B1,B2    !Read 2 self-identify bytes
760   DISP IVAL$(B1),IVAL$(B2)     !Display bytes in binary
```

Binary INTEGER Input (Specifiers W and Y)

These specifiers are intended primarily for use with 16-bit interfaces such as the HP 27112A GPIO card, or other destinations such as BUFFERs and BDAT files, which can accept data in 16-bit INTEGER units. The returned numeric value is in the range −32 768 through 32 767.

The `Y` specifier reads either a single INTEGER or two bytes. The `W` specifier reads a single INTEGER quantity from the source. If the source's byte pointer is not aligned on an even byte boundary, ENTER...USING reads and discards one byte to align the source.

In the following example, the "?" character is skipped by this word alignment process.

```
930   String$="A?"&CHR$(0)&"A"&CHR$(0)&"A"
940   ENTER String$ USING "B,W,Y";Byte,Word,Bytes
950   PRINT Byte;Word;Bytes
```

This program prints the following result:

	6	5		6	5		6	5		eol

Input Termination

The ENTER...USING statement normally terminates, and your program resumes execution when the last ENTER variable has had a value assigned to it, and simultaneously, or subsequently, ENTER detects either the current DELIM character or an END condition. ENTER...USING can read up to 256 bytes beyond the last assigned data byte waiting for DELIM or END.

The default DELIM character and END conditions depend on the source. See the "I/O Resources" appendix of the BASIC Language Reference manual. For example, the default DELIM expected by ENTER KBD USING is the ASCII line-feed (LF, CHR$(10)) character. The keyboard sends a CR LF when you press (RETURN). The keyboard can also simulate an END condition when you press (CTRL) plus (E) This END must be enabled by a CONTROL statement.

During input the ENTER...USING statement is not always scanning for the DELIM character. In fixed-width fields such as SDCDD or AX5A the DELIM is ignored or treated as data. It is only during free-field (K, H) input or after the last variable is assigned data that ENTER...USING processes the DELIM as an item or statement termination condition.

If an END occurs in a free-field item, it terminates assignment to the associated variable. If an END occurs in a fixed-width field, an error occurs, unless a % specifier is in effect. You can use the - specifier to "turn off" END detection beyond the last data byte.

You can re-define the DELIM character for I/O path names. For all sources, there are four image specifiers that change the way ENTER...USING interprets DELIM characters and END conditions; they are: #, +, - and %. If you include more than one of these specifiers in a single image, each specifier processed establishes a new mode and overrides the earlier specifiers.

Input without Terminator (# Specifier)

The # specifier causes the ENTER statement to read no characters after satisfying the variable list. # is useful when reading beyond the last required byte would cause the loss of valuable data.

If the last specifier is a fixed-width field, ENTER performs a **counted** read, requesting only the necessary number of characters from the interface or device.

If the last specifier is a variable-width field, ENTER reads characters only until it detects a DELIM character or END condition. ENTER must read each character in a separate operation (**single character** read mode). This means that numeric #...K input can result in slow I/O data rates.

Early Statement Termination (Specifier %)

The ENTER...USING statement normally terminates after the last variable in the list has had a value assigned to it. Detection of the DELIM character or an END condition before the list is satisfied terminates items or is an error. You can use the % specifier to terminate ENTER...USING immediately on detection of an END condition. The remaining variables in the enter list retain the values they had prior to the ENTER statement.

Even if you specify `%`, an END condition occurring prior to the end of a fixed-width field causes an error. You can "trap" such errors with an ON ERROR statement.

```
480   Reading_1=Reading_2=0
490   ENTER Bus_device USING "%,3D";Reading_1,Reading_2
500   PRINT Reading_1;Reading_2
```

This example reads data from an HP-IB device. The device asserts EOI (an END condition) with the last character of each data item. For our example it sends this data stream:

data bytes	**1**	**2**	**3**	**7**	**8**	**9**
EOI line state	**0**	**0**	**1**	**0**	**0**	**1**

The example prints the following result:

	1	**2**	**3**		**0**	**eol**

Ignoring DELIM as Statement Terminator (Specifier +)

If you include a + specifier ENTER...USING reads and discards up to 256 bytes after satisfying the last variable, but it ignores the DELIM. Statement termination occurs only on an END condition or error. DELIM is still recognized as an item terminator.

A typical use for + is to force ENTER...USING to read and ignore any remaining data (including DELIMs) to the end of the current magentic tape record.

Ignoring END as Statement Terminator (Specifier -)

The - specifier is similar to the + specifier except that the roles of END and DELIM are reversed. Using -, ENTER...USING ignores END as a statement terminator, but not as an item terminator. After satisfying the last variable, ENTER...USING reads up to 256 characters waiting for the DELIM, if the last item was not terminated by a DELIM.

A typical use for - is in reading data from an HP-IB instrument. The instrument may return six readings and assert EOI (an END condition) with the last byte of each and send a CR LF after the last reading. If you are only interested in two of the readings you can use a statement similar to the following example. This ENTER accepts two values and ignores the remaining four, as well as ignoring their EOI's.

```
560   ENTER Bus USING "-,K,K";Reading_1,Reading_2
```

Skip to New Line (Specifier /)

The / specifier causes ENTER...USING to read and ignore all input characters until after the next line-feed character. Note that / does not skip to next DELIM (unless DELIM is LF).

The / specifier is useful when your ENTER...USING statement is reading only part of a line (record) and the line is over 256 bytes long. In this case an error might occur because ENTER...USING can only skip 256 bytes. The / forces ENTER...USING to skip the necessary number of bytes.

Chapter 18
Bit Manipulations

This chapter looks at how your computer's 32-bit word is used to represent the numeric and string data types, two's complement arithmetic, and finally bit masking and shifting. The "Variables" chapter discusses storage for the four numeric and string data types in greater detail.

Binary Storage of Data

The system's internal memory holds your program and the data your program uses. You can manipulate the individual bits of memory locations using statements in the BASIC Language System.

When the state of an individual bit is of interest, it is common to describe the contents of a memory location using a binary notation.

In order to convert the familiar decimal numbers you use to the binary representation of the number which the computer uses, take the following steps:

1. Change the number into base 2 or binary form:

   $$\text{Binary number} = \sum_{n=b}^{e} (0 \text{ or } 1) \times 2^n$$

   where b is the beginning power of 2
   e is the ending power of 2

   For example, the decimal number 5 can be represented in the binary number system as 101 –

   $1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$
   $(4 + 0 + 1)$

The following table illustrates some other decimal or base 10 numbers with their binary representations.

An 8-bit representation is used for brevity. The MSB, most significant bit, labeled bit 7, is saved for the sign. The remaining 7 bits, labeled from left to right (6-0), contain the binary digits representing the number. In the 32-bit word, the MSB, bit 31, contains the sign. The remaining 31 bits, labeled left to right (30-0), represent the number.

binary number								
2^7 MSB	**2^6 64**	**2^5 32**	**2^4 16**	**2^3 8**	**2^2 4**	**2^1 2**	**2^0 1**	**decimal equivalent**
0	0	0	0	0	0	0	1	1
0	0	0	0	0	1	0	1	5
0	0	0	0	0	1	1	0	6
0	0	0	0	1	0	1	1	11
0	0	1	0	0	0	0	0	32
0	1	1	0	1	0	0	1	105

2. If the number is positive, you are finished.
3. Negative numbers take the two's complement form for integers and signed magnitude and binary exponent form for reals.

   a. Two's complement form:

   Change the zeros to ones and the ones to zeros, and then add 1, for example:

binary number								**decimal number**
0	0	0	0	0	1	0	1	5
1	1	1	1	1	0	1	0	
							+1	
1	1	1	1	1	0	1	1	−5

   b. Signed magnitude form:

   Signed magnitude representation uses the MSB for the sign; 0 if plus and 1 if minus, followed by the positive binary representation of the number. There is no conversion necessary for negative numbers.

binary number								**decimal number**
1	0	0	0	0	1	0	1	−5

Notice that in the signed magnitude form two representations for zero are possible, a plus zero and a minus zero (the MSB equal to 0 or 1, followed by all zeros). Note also that you can represent as many positive numbers as negative numbers.

In the two's complement form only one representation for zero is possible: 0 in the MSB followed by all zeros. It is possible, therefore, to represent one more negative number than positive numbers.

There are also systems of numeric representation based on eight symbols (octal) and 16 symbols (hexadecimal). When 8 bits of memory called a byte, are sent to a printer of display, it may also be represented as a character. The Character Codes Table in the "Useful Tables" section of the BASIC Language Reference shows the character, binary, octal, decimal and hexadecimal equivalents for numbers in the range 0 through 255.

Binary Arithmetic

Your computer can do binary arithmetic. Integer arithmetic is performed using full word, 32-bit precision; real arithmetic uses either one or two words, depending on the operation. The chapter, "Numeric Computations", discusses arithmetic operations and precision.

Two's Complement Addition and Subtraction

Two's complement addition involves normal binary addition with the carry generated by the leftmost bits thrown away. For example:

(An eight-bit number is used again for brevity. The MSB is saved for the sign.)

```
   0 0 1 0 1 0 1 1 ( 43)         1 0 1 1 0 0 1 1 (-77)
 + 0 0 0 0 1 1 1 0 (+14)       + 0 1 1 0 0 0 0 1 (+97)
 -----------------------       -------------------------
   0 0 1 1 1 0 0 1 ( 57)       1 0 0 0 1 0 1 0 0 (+20)
                               ↑
                           discarded
```

If both numbers added are of opposite signs, overflow cannot occur. When both numbers are of the same sign, overflow occurs if carry into the MSB is different than the carry out of the MSB. For example:

```
   0 1 0 0 0 0 0 0  (+64)
 + 0 1 0 0 0 0 0 0  (+64)
 -------------------
   1 0 0 0 0 0 0 0  (-128) POSITIVE OVERFLOW
                           should be +128

   1 0 1 0 0 0 0 0  (-96)
 + 1 1 0 1 0 1 0 0  (-44)
 -------------------
 1 0 1 1 1 0 1 0 0  (116)  NEGATIVE OVERFLOW
 ↑                         should be -150
discarded
```

Since the eighth bit is the sign bit, only numbers in the range of −128 to +127 can be represented in this 8-bit example. In the addition labeled POSITIVE OVERFLOW above, the carry into the MSB is 1 and the carry out of the MSB is 0. In the NEGATIVE OVERFLOW example, the carry into the MSB is 0, while the carry out is 1.

Subtraction is performed by adding the negated augend.

Two's Complement Multiplication and Division

Two's complement multiplication is performed by multiplying one digit of the multiplier, necessarily either 0 or 1, times the multiplicand. The partial product obtained is shifted one place to the left for each successive multiplication; these partial products are then summed. An example follows.

$$\begin{array}{r} 1\,0\,0\,1 \\ \times\,0\,0\,1\,1 \\ \hline 1\,0\,0\,1 \\ +\,1\,0\,0\,1 \\ \hline 1\,1\,0\,1\,1 \end{array} \qquad \begin{array}{r} 9 \\ \times 3 \\ \hline 27 \end{array}$$

The division algorithm is analogous to the multiplication algorithm, except that it is accomplished by subtracting and shifting.

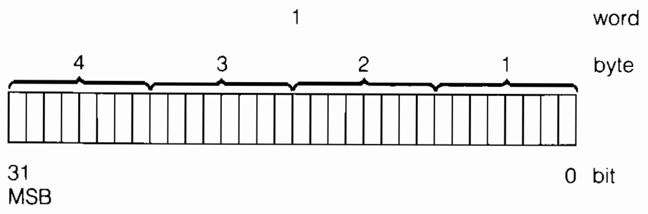
Bit Functions

When you want to manipulate the bits of the 32-bit word directly, use the functions discussed in this section. These functions return the value of a specified bit position, perform binary logical operations on their arguments or shift the binary number one bit to the right or to the left. To use these functions, the IO option of the BASIC Language System must be loaded first. Refer to the "Getting Acquainted with BASIC" chapter for the procedures necessary to load binary files. The chapters on advanced I/O use the bit function concepts.

BIT

BIT returns a 0 or a 1 representing either the value of the specified bit in the argument, or whether specified bits in the argument match those in a bit mask.

The argument and the bit number can both be numeric expressions and are both rounded to 32-bit integers. The bits in the binary representation (two's complement) of an argument are numbered left-to-right (31 through 0).



The bit mask is a string of binary digits, 0 to 32 characters long, composed of 0's, 1's and "don't care" characters. You can use any character other than a 0 or 1 as a "don't care" character. If the bit mask is a null string or entirely don't care characters, a 1 is returned. If you specify fewer than 32 characters, the bit mask is considered right-justified and the omitted bits are interpreted as don't care bits. The BIT function returns a 1 only when each bit in the argument matches the bit mask or when the corresponding bit mask is a don't care.

Example statements using BIT are:

```
 60 Value=BIT(Number,Bit_number)
190 IF BIT(Number,Bit_mask$)=1 THEN CALL Service
```

In line 60 when Number equals 00000000000000000000000000100000, Value is equal to 0 unless Bit_number is 6.

In line 190 with the same Number argument, Bit_mask must either be all don't cares or have the form "xxx1xxxxx", where x is either a 0 or a don't care, to have the BIT function return a 1.

Binary Logical Functions

Each of these four functions converts its arguments to 32-bit two's complement values. Each bit value in one argument is appropriately compared with the corresponding bit value in the other argument. They return the decimal equivalent of the 32-bit result in the range of DOUBLE.

BINAND returns the binary AND of two arguments.

BINEOR returns the binary exclusive OR of two arguments.

BINIOR returns the binary inclusive OR of two arguments.

BINCMP returns the binary complement of its argument.

Truth Table

bit A	bit B	A AND B	A EOR B	A IOR B	Complement of bit A
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	0	1	0

The following program illustrates the use of the logical binary functions.

```
 10  INPUT "WHAT IS THE FIRST VALUE?",Val_1
 20  INPUT "WHAT IS THE SECOND VALUE?",Val_2
 30  PRINT "THE BINARY VALUE OF ARGUMENT 1 IS: "&DVAL$(Val_1,2)
 40  PRINT
 50  PRINT "THE BINARY VALUE OF ARGUMENT 2 IS: "&DVAL$(Val_2,2)
 60  PRINT
 70  PRINT "THE BINAND OF THESE VALUES IS: "&DVAL$(BINAND(Val_1,Val_2),2)
 80  PRINT
 90  PRINT "THE BINEOR OF THESE VALUES IS: "&DVAL$(BINEOR(Val_1,Val_2),2)
100  PRINT
110  PRINT "THE BINIOR OF THESE VALUES IS: "&DVAL$(BINIOR(Val_1,Val_2),2)
120  END
```

If you try these logical functions, remember to interpret the result as the two's complement representation. For example:

```
90 Result=BINCMP(0)
```

Result is all ones, which is the two's complement representation of -1. The numbers returned by these functions are printed as the decimal equivalents of these two's complement representation.

Functions Which Shift or Rotate

These four functions displace their argument to the right or to the left. A positive bit displacement causes rotation toward the least significant bit (right), while a negative bit displacement causes rotation toward the most significant bit (left).

Functions Which Rotate

These functions produce a "wrap around" effect, which means that the bit shifted out at either end is wrapped around and placed into the other end.

ROTATE returns the INTEGER value of a rotated numeric argument. The argument can be a numeric expression. The argument is first rounded to an integer and then evaluated MODULO 65536. The bit displacement can also be a numeric expression; however, it is is evaluated MODULO 16 after rounding to an integer.

DROTATE returns the DOUBLE value of a rotated numeric argument. The argument must be in the range of DOUBLE and is first rounded to a DOUBLE. The bit position displacement can be a numeric expression; it is first rounded to an integer and then evaluated MODULO 32. The bit shifted out is again "wrapped around".

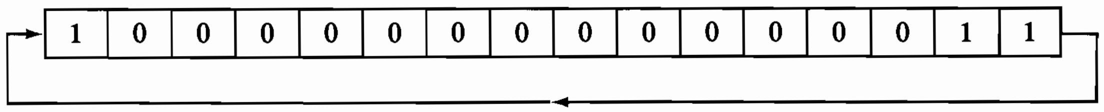
The following examples illustrate the ROTATE and DROTATE functions. Note the "wrap around" effect, especially where the value of the sign bit changes.

```
160 Result=ROTATE(7,+1)
170 Result=ROTATE(7,-1)
230 Result=DROTATE(7,+1)
240 Result=DROTATE(7,-1)
```

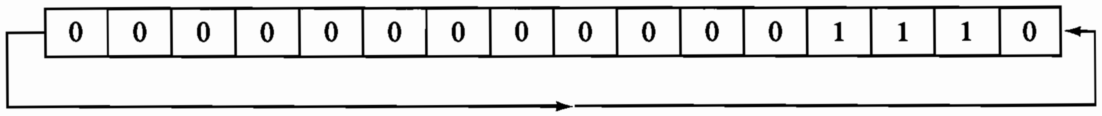
Initial binary equivalent of argument: **Value: 7**

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1

+1 bit ROTATE: $\longrightarrow$ **Result: −32765**

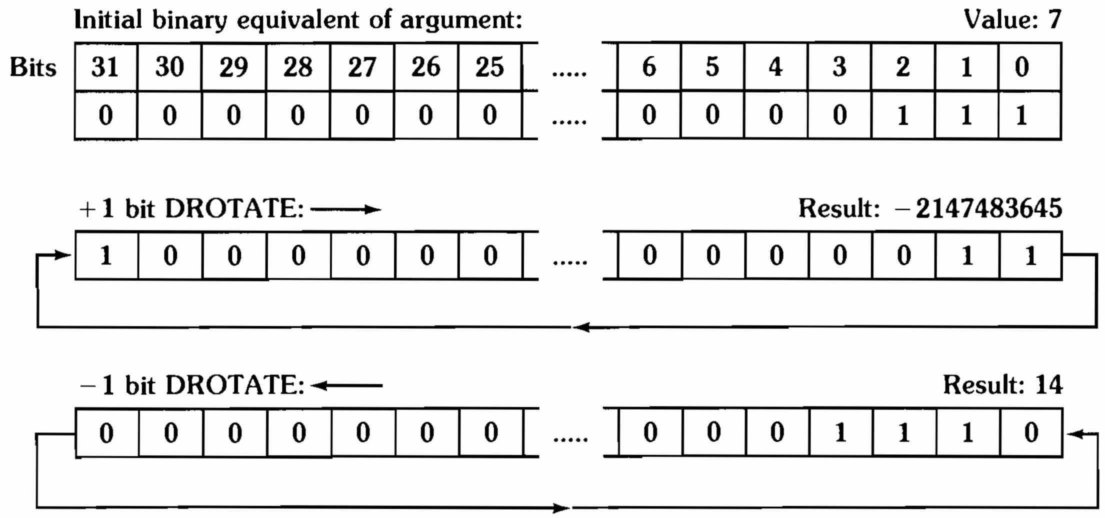


−1 bit ROTATE: $\longleftarrow$ **Result: 14**



The 16-bit value resulting from the rotation is treated as a two's complement number when it is returned. If the value of bit 15 changes, the sign of the result is not the same as the sign of the argument.

The following examples illustrate positive and negative 1-bit DROTATEs.



Shifting Functions

The bit shifted out of the word at either end is lost. A zero bit is shifted into the other end.

SHIFT returns the integer value of a shifted numeric argument. The argument and the bit position displacement can be numeric expressions and are first rounded to integers. The argument must be in the range of DOUBLE and is evaluated MODULO 65 536.

DSHIFT returns the integer value of a shifted numeric argument. Both the argument and bit position displacement must be in the range of DOUBLE, can be numeric expressions, and are rounded to integers.

The following examples illustrate the shifting functions.

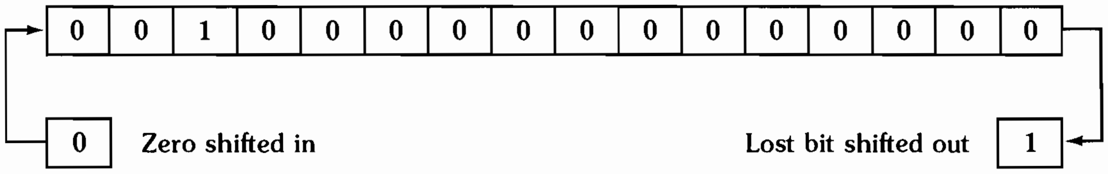
```
100 Result=SHIFT(16385,+1)
110 Result=SHIFT(16385,-1)
350 Result=DSHIFT(1073741825,+1)
360 Result=DSHIFT(1073741825,-1)
```

The following examples illustrate positive and negative 1-bit SHIFTs.

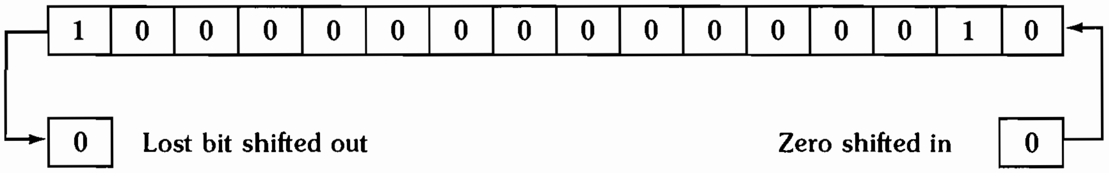
Initial binary equivalent of argument: **Value: 16385**

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1

+1 bit SHIFT: **Result: 8192**



−1 bit SHIFT: **Result: −32766**



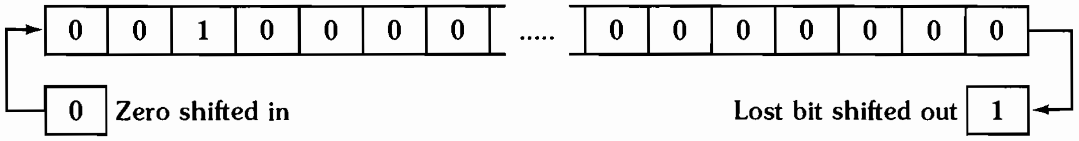
If the returned value of SHIFT is assigned to a variable of type DOUBLE, the value of the sing bit (15) is propagated through bits 16 thru 31 of the DOUBLE variable.

The following examples illustrate positive and negative 1-bit DSHIFTs.

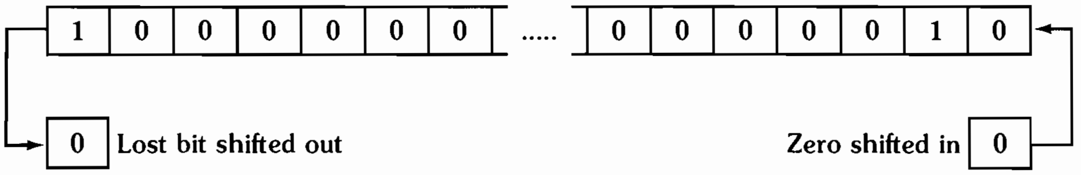
Initial binary equivalent of argument: **Value: 1073741825**

Bits	31	30	29	28	27	26	25	.....	6	5	4	3	2	1	0
	0	1	0	0	0	0	0	.....	0	0	0	0	0	0	1

+1 bit DSHIFT: **Result: −536870912**



−1 bit DSHIFT: **Result: −2147483646**



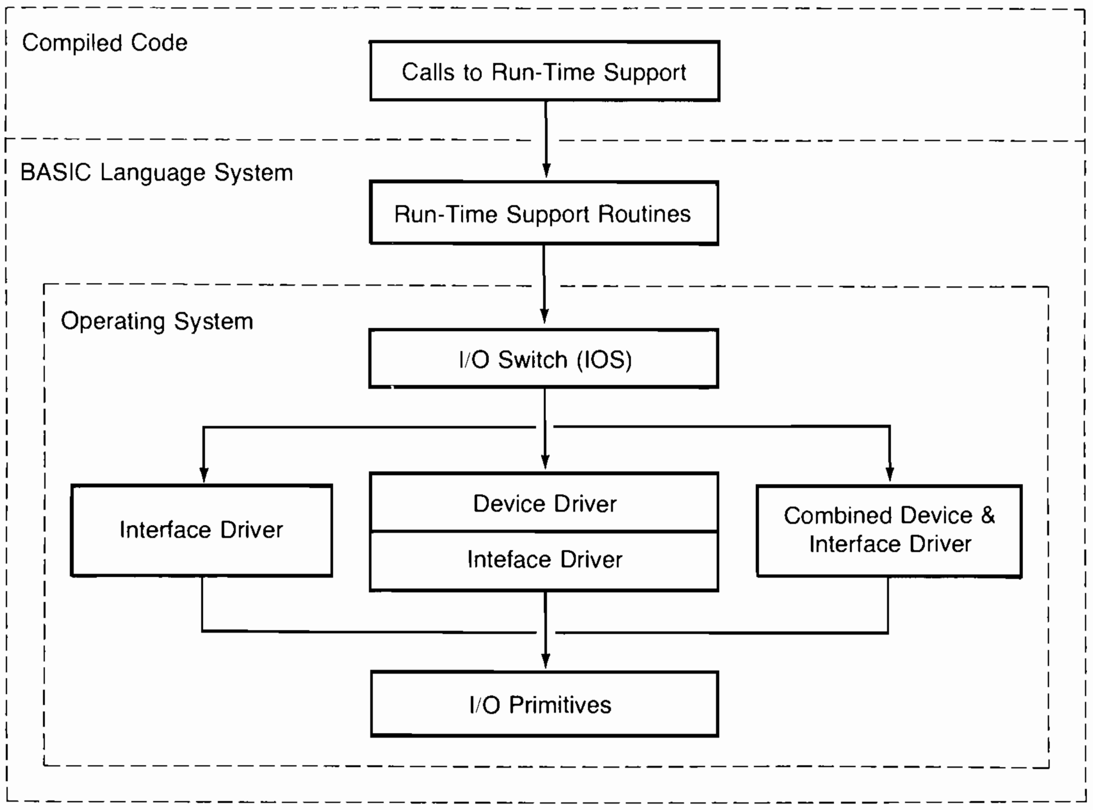
Chapter 19
Advanced I/O Operations

This chapter introduces BASIC I/O concepts and statements which provide you with much more control over I/O operations than those previously discussed. To take advantage of the flexibility and performance offered by these operations, you need a more detailed picture of how the BASIC Language System performs I/O. Where the "Introduction to I/O" chapter began by describing the **hardware** I/O architecture of the Model 520, this chapter begins by discussing the **software** I/O architecture.

Software I/O Architecture

The software of the BASIC Language System that performs I/O is in **layers** of modules. Some of these software modules are supplied as BIN option files. To use the features of a module, you must load it into the system. The first part of this chapter discusses the layers and how the modules relate to one another.

The following diagram summarizes the layer organization, or **architecture**, of the BASIC Language System. The terms and function of each layer are described in the following paragraphs.



BASIC Language System I/O Architecture

BASIC I/O Statements

When you enter a BASIC statement with GET or during EDIT, the system checks its syntax and converts it to a form known as intermediate code, or i-code. When you execute the statement or run its program, the system first compiles the i-code into machine language. I/O statements are compiled at run time or by the COMPILE statement into machine instructions which call BASIC run time support routines. At run time your program consists of the **compiled code** layer shown in the block diagram. When the instructions execute, they call the run time support code.

To enter and execute a BASIC statement, the modules which perform the syntaxing, compilation and run time support must be present. For some I/O statements, such as DISP, the support is always present. For many others, such as OUTPUT, the support is provided by the BIN option file. The BASIC Language Reference manual identifies any BIN option file(s) required in the Keyword Usage Table for each statement. These files must be loaded before you can execute the program.

In I/O operations, the **run time support routines** layer of the BASIC Language System performs most of the "work". The run time support layer is responsible for preparing your data, if required, and making one or more I/O requests of a standardized form to the next lower layer. For example, this is where binary to ASCII data conversion occurs and where image specifiers are acted upon. This layer, and all the layers below it, are not part of your program. They are part of the memory resident system and consume no memory in your program's partition. All programs in the computer share these layers.

Some BIN option files add only to the run time support layer, and do not add new statements to the language. These BIN option files only add capability to the statements. For example, file "LIF_DISC" adds support for the "Logical Interchange Format" mass storage directories, but adds no mass storage statements to the language. The BASIC Language Reference manual identifies any BIN option file required for a specific capability in the Semantics Section for each statement.

The I/O Switch

The next layer below the BASIC run time support code is the I/O Switch (IOS). To this point, the internal calls and requests performed on behalf of your BASIC I/O statement are interface and device **independent**; that is, nothing about the support software is unique to any specific interface or device. The IOS is responsible for "switching" the request to the appropriate interface and device **dependent** software (driver).

The IOS also ensures that no other I/O request accesses the I/O resource specified in your I/O statement until your statement completes.

I/O Drivers

Although the interface cards supported by your computer have the same electrical and mechanical protocol at the I/O backplane, they have very different capabilities. Additionally, the devices which you connect to any given interface also have different capabilities. So that you can use the same BASIC statement to access any interface, BASIC Language System has software modules which account for the different interface and device programming considerations. These modules are called **drivers**.

Some drivers are supplied as part of the minimum BASIC system; others are supplied in BIN option files. To access a given I/O resource, the driver or drivers that support the resource must be loaded in the system.

Each driver has a name, such as "KEYBOARD", "HP27110" or "HP7971". Some BIN option files, such as "HP9885" contain a single driver, in this case "HP9885". Other BIN option files contain drivers for several devices, such as "HP7970", which contains the driver for "HP7970" and "HP7971". The Software Manual Catalog contains a list of drives and the file in which they are located.

Levels of I/O Drivers

To avoid duplication of code, driver software is divided into modules in a manner similar to the hardware it supports. For the different interface cards, there are **interface** drivers. For the various devices which can be connected to interfaces, there are **device** drivers. There are a few drivers which combine the two functions and there are two special purpose drivers.

Interface Drivers

An **interface** driver accepts interface-independent requests from the IOS (or device driver) and converts them to the **interface** dependent form required by the interface card involved in the operation. An interface driver is programmed to transfer data, control and status information to and from the external world via an interface card. An interface driver is not programmed to provide for the requirements of any specific peripheral device connected to that interface.

The following are typical interface drivers.

Driver Name	BIN File Name	Interface Card Supported
HP27110	HP27110	HP 27110A HP-IB Interface
HP27112	HP27112	HP 27112A GPIO Interface
SERIAL	SERIAL	HP 27128A ASI Interface

You may be able to access a peripheral device with only an interface driver; however, your BASIC program must contain the programming for the device dependent features. For example, using only the HP27110 HP-IB interface driver, you can rewind the tape on an HP 7971A tape drive, but you cannot transfer data to or from the tape drive due to the complexity of the bus sequence and timing constraints. On the other hand, if you use the HP7971 device driver plus the HP27110 interface driver, you can simply use the OUTPUT statement.

Device Drivers

A **device** driver accepts device and interface independent requests from the IOS and converts them to the **device** dependent form required by the specified peripheral device. The device driver passes the requests on to the interface driver which supports the interface card to which the device is connected. The device driver is programmed to transfer data, control and status information to and from the peripheral device. A device driver is not programmed to provide for the requirements of the interface card; that is the job of the interface driver.

The following are typical device drivers.

river Name	BIN File Name	Peripheral Device(s) Supported
CS80	—	HP 7908/11/12/33 External Disc Drives
HP7970	HP7970	HP 7970E, 7971A Master Tape Drives
HP7971	HP7970	HP 7971A, 7970E Master Tape Drives
HP82901	HPIB_FLEX	HP 82901M/S Flexible Disc Drives
HP82902	HPIB_FLEX	HP 82902M/S Flexible Disc Drives
HP8290X	HPIB_FLEX	HP 82901M/S or HP 82902M/S Drives
HP9895	HPIB_FLEX	HP 9895A Flexible Disc Drive

A device driver is programmed to be independent of the interface used to communicate with the device(s) it supports. Consequently, a device driver is always used with an interface driver. For example, the device drivers listed in the previous table all require the HP-IB interface driver (HP27110).

Combined and Non-device Drivers

In cases where a peripheral can be used only with a specific interface, the functions of interface and device driver may be combined in a single module. There are also two special purpose drivers which do not support physical interface cards or devices.

The following are typical combined drivers.

Driver Name	BIN File Name	Device Supported
CS80	—	Internal mass storage
HP9885	HP9885	HP9885M/S Flexible Disc Drive
INTERNAL_CRT	—	Internal alpha CRT display
KEYBOARD	—	Internal keyboard
THERMAL_PRINTER	—	Internal thermal line printer

Driver "CS80" is a combined driver only for the internal disc and other mass storage devices. For external disc drives it is a device driver and requires the use of the HP-IB interface driver.

The following are the two special purpose non-device drivers.

Driver Name	BIN File Name	Capability Supported
MEMORY	MEMORY_VOL	Memory resident files
NULL	—	Simulated output

Driver "MEMORY" simulates a mass storage device driver. It provides the capability to initialize mass storage volumes in read write memory and store files in those volumes. Because no interface is involved, no interface driver is required.

Driver "NULL" provides the simulation of output without an actual transfer of data. When "NULL" receives an output request through the IOS, it immediately signals completion and the data is simply lost. When "NULL" receives an input request through the IOS, it immediately signals an END error and transfers no data. "NULL" does not support status or control requests.

I/O Primitives

None of the software modules discussed so far actually execute machine I/O instructions. The various modules perform computations, manipulate data and execute requests (calls) to the next lower layer. Access to the I/O Processors and I/O backplanes is reserved for the lowest layer, the I/O Primitives module.

Specifying I/O Drivers

This topic has two parts. The first part is: When is it necessary to specify a driver?

- You always must specify mass storage device drivers, whether combined or or not.
- You must specify noncombined device drivers in all cases.
- You must specify the "NULL" driver when you want to access it.

The second part of this topic is: How do you specify a driver? There are four methods each; of which is discussed separately.

Implicitly Specifying Interface Drivers

You do not need to specify interface drivers, but there are some considerations concerning them. When you power-up the computer, execute a LOAD BIN or SCRATCH A command, the reset process calls each interface driver module installed. Each driver is requested to scan all the installed interface cards and identify those which it supports. If no driver "claims" a given interface, the system alerts the IOS that no driver is present for the interface.

When you specify a device selector or interface select code in an I/O statement, the system uses the interface driver which "configured" itself to the interface. If you attempt to perform I/O with an interface for which no driver is loaded, the system returns BASIC ERROR 163, "Device or interface not present".

Specifying Mass Storage Drivers

Whenever you use a file specifier, you must identify the mass storage device driver which supports the medium involved. You specify the driver in the mass storage unit specifier (msus). You can either include the msus in the file specifier, or you can use the default msus which was specified in non-volatile memory (NVM) or in the most recent MASS STORAGE IS statement.

The syntax of the msus and media specifier are described in the "Mass Storage Organization" chapter. Note that the msus does not include a leading colon (":"). This is part of the media specifier syntax and not part of the driver name. Note also that the media specifier ":INTERNAL" does not name a driver. The msus INTERNAL is merely a pseudonym for CS80,7,1.

Specifying Graphics Drivers

When you initialize a graphics plotter with the PLOTTER IS statement or initialize a graphics input device with GRAPHICS INPUT IS statement, you must specify the device driver required. If the graphics plotter is an array, you do not specify a driver. You can specify **only** a graphics device driver, such as "GRAPHICS", "HPGL", "INTERNAL", "ARROW KEYS" or "LIGHT PEN". You cannot specify a nongraphics device driver, such as "HP7970". For more information see the BASIC Graphics Programming Techniques manual.

Specifying I/O Path Name Drivers

When you open an I/O path name with an ASSIGN @ statement, you can specify the non-graphics driver name with the DRIVER I/O attribute. The only circumstance in which you must specify the driver is if a **device** driver is required. For example, here is a typical assignment for an HP 7971A magnetic tape drive.

```
    20 ASSIGN @Tape TO 403;DRIVER "HP7971",BUFFER SIZE 32768
```

If you want to change the device driver used for an open I/O path name, you must first **close** the I/O path name. Reopen it with the new DRIVER attribute.

If you use the DRIVER attribute for any resource supported by an interface driver or combined driver, the DRIVER expression must match the default driver. If you use the DRIVER attribute when opening a file to an I/O path name, the DRIVER expression must match the driver name of the msus.

Status and Control Registers

When you need to know the state of an I/O resource or perform an I/O operation which does not involve the transfer of data, you need to access I/O registers. I/O registers are memory locations associated with an I/O resource. They are located in the memory of the computer, an I/O interface card or peripheral device.

Here is an example of I/O register access. This program line obtains the current column and row on the STD SCREEN.

```
    360   STATUS CRT,0;Column,Row    !Obtain col and row
```

You access I/O registers with the STATUS and CONTROL statements, and the IOSTAT function. Only device selectors (including interface select codes) and I/O path names have registers that you can access.

Specifying I/O Registers

I/O registers are identified by an integer number having a value ≥0. The register numbers that you can access depend on the type of I/O resource. The registers are documented for each resource in the "I/O Resources" section of the BASIC Language Reference manual.

I/O path name registers 0 through 99 are associated with the status of that path name only in the context which opened the path name. For example, file pointer information in one context does not affect, or reflect, I/O activity in another context (or program) accessing the same file.

Device selector, and interface select code registers and I/O path registers above 100 reflect the current state of the physical device. Although a register may correspond to a physical register within an interface or device, the number used in the BASIC Language System is not necessarily the identity of the register in the interface or device reference manual.

Each register contains either numeric or string data. The variables specified in a STATUS statement and the expressions specified in a CONTROL statement must be of the same type as the register. Although numeric variables may be any numeric type, STATUS variable types should be chosen with the full range of the register in mind to avoid overflow errors.

Register Access Restrictions
The STATUS, IOSTAT and CONTROL statements cannot, with a single addressing method, access **all** I/O registers which may be related to a given operation. The following table summarizes the access restrictions. An example follows the table. The "I/O Resources" section of the BASIC Language Reference manual documents only the registers you can access for any particular resource.

Register Range	Associated I/O Resource	To access these registers, the statement must specify:
0 thru 13	CRT, KBD, PRT	appropriate device selector; an I/O path name assigned to these device selectors cannot access these registers, only the I/O path name registers
0 thru 99	any I/O path name	any I/O path name
1xx	HP 7970E, HP 7971A mag tape drives	an I/O path name assigned with DRIVER attribute of "HP7970" or "HP7971"
2xx	HP 9885M/S flexible disc drive	an I/O path name assigned to a file with a media specifier of ":HP9885"
3xx	HP 27110A HP-IB interface	the bus interface select code, a bus device selector or an I/O path name assigned to the interface or a bus device with a DRIVER "HP27110" attribute or no DRIVER attribute
4xx	HP 27112A GPIO interface	the interface select code or an I/O path name assigned to the interface select code with a DRIVER attribute of "HP27112" or no DRIVER attribute
5xx	HP 27128A ASI interface	the interface select code or an I/O path name assigned to the interface select code with a DRIVER attribute of "SERIAL" or no DRIVER attribute

The following statement assigns an I/O path name to an HP 7971A tape drive.

```
40   ASSIGN @Mag_tape TO 403;DRIVER "HP7971",BUFFER SIZE 32768
```

The following STATUS statements access the I/O path name registers and the "HP7971" device registers.

```
400   STATUS @Mag_tape,1;Select_code   !Get 7971 interface sel. code
410   STATUS @Mag_tape,102;Re_tries    !Any re-tries on last write?
```

You cannot access the "HP27110" interface registers with I/O path name "@Mag_tape". To obtain interface status, you can use any of the following techniques.

```
420   STATUS Select_code,303;Card_status   !Use @name reg 1 data
430   STATUS SC(@Mag_tape),303;Card_status !Use SC function
440   ASSIGN @Bus TO 4                    !Assign @name to bus...
450   STATUS @Bus,303;Card_status          !and Use new @name
```

I/O Status

I/O status reporting has many uses. Some of the most common are:

- determining that the interface or device is ready or in the proper mode before beginning an I/O operation;
- determining that your I/O operation had the desired result in a case where BASIC does not report an ERROR;
- establishing the state to which your program must return an I/O resource on completion of a task.

The IOSTAT function reads a single numeric status register. The following example shows how a program can check to ensure that an internal printer is present before executing a PRINTER IS statement.

```
370   Status=IOSTAT(PRT,0)               !Read status register
380   Present=BIT(Status,15)             !Check "installed" bit
390   IF Present THEN PRINTER IS PRT     !Re-assign printer if OK
```

The STATUS statement can read one or more registers containing either numeric or string data. The opening example of this section demonstrated reading two registers (CRT column and row) with a single STATUS statement.

I/O Control

The CONTROL statement is the complement of the IOSTAT and STATUS statements. CONTROL **writes** to I/O registers. Writing to I/O registers not only changes their contents, it often causes some action. Here are some uses for CONTROL:

- You can change the "current position" of the CRT, file, BUFFER or device, such as rewinding a magnetic tape drive.
- You can change the state or mode of an interface or device, such as disabling CAPS LOCK on the keyboard.
- You can perform an I/O operation which is other than a data read or write, such as writing an IRG (record gap) or EOF (file mark) on a magnetic tape.

The following example configures the HP 27128A ASI card for communication with an RS-232C peripheral.

```
 40   ASSIGN @Asi TO 5;DRIVER "SERIAL"
 50   Handshake=IVAL("01100010",2)      !Echo on, host Enq/Ack
 60   Enable_shake=IVAL("10000000",2)   !Enable handshake mask
 70   Bits_per_char=3                   !8 bits per char
 80   Parity=0                          !No parity
 90   Baud_rate=IVAL("F",16)            !9600 baud
100   RESET SC(@Asi)                    !Restore card defaults
110   CONTROL @Asi,526;Handshake,Bits_per_char
120   CONTROL @Asi,520;Enable_shake
130   CONTROL @Asi,529;Parity
140   CONTROL @Asi,561;Baud_rate
```

Internal Buffering and Overlapped I/O

I/O statements transfer data between program expressions, program variables and a block of memory known as a buffer. I/O drivers transfer data between the buffer and the I/O resource. In most I/O operations, the buffer is an intermediate "holding area" for your data. This two-step process is necessary for several reasons; for example, because the form of program data is not the same as the form required by the device or file or because the I/O statement cannot prepare and pass the data as fast as the device can accept it.

You can have the system perform both of these steps or you can perform them yourself. When the system performs both steps, your data passes through buffers internal to the system. When you perform both steps, you declare a BUFFER and access it as an I/O resource.

When the system performs both steps, it is possible for the transfer between the buffer and the I/O resource to occur while your program performs unrelated operations. That is, I/O and program execution can **overlap**.

The BASIC Language System provides several statements and I/O attributes which you can use to control these features and improve the I/O performance of your programs.

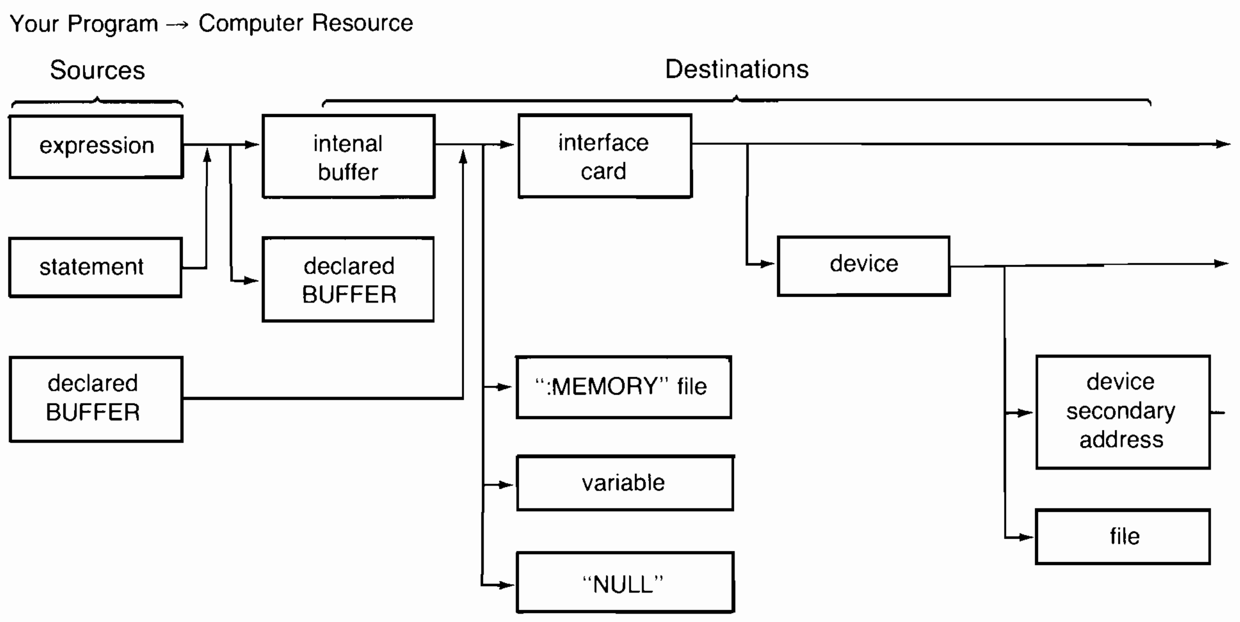
Internal System Buffers vs Declared BUFFERs

Internal buffers are used for all I/O operations except those involving I/O path names assigned to BUFFERs or devices. For an I/O path name you can dimension or allocate a BUFFER which is used as the I/O source or destination. Here are examples of internal system buffering, dimensioning a BUFFER and allocating a BUFFER.

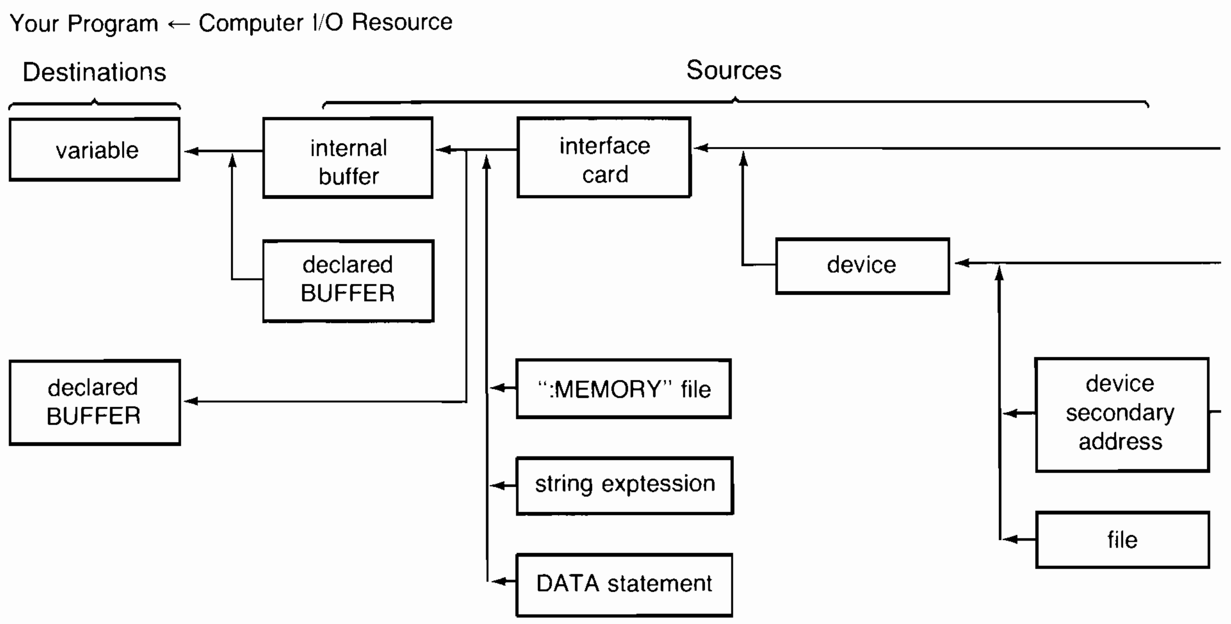
```
20   ASSIGN @Dest TO 406                  !System-buffered name.


10   DIM Printer$[512] BUFFER             !BUFFER variable
20   ASSIGN @Dest TO BUFFER Printer$      !User-buffered name.


20   ASSIGN @Dest TO BUFFER [512]         !Allocate a BUFFER.
```

There is a significant difference between internal system buffers and declared BUFFERs. If you elect to use an internal system buffer, your I/O data always passes **through** the buffer. The system performs both steps of the I/O operation. If you use a declared BUFFER, the BUFFER becomes the I/O resource. Your program must perform each step of the I/O operation in separate I/O statements.

The following diagrams are the data path diagrams from the "Introduction to Output" and "Introduction to Input" chapters. The diagrams are enhanced to show the difference between internal system buffering and the use of a declared BUFFER.



Output Data Paths



Input Data Paths

Changing Internal Buffering

The default internal buffering for interfaces and devices is two buffers each 256 bytes long. The default for files is a single buffer 512 bytes long. Before you change these defaults, you need to determine whether there is any benefit from doing so.

Increasing the Buffer Size - BUFFERSIZE

The size of an individual BUFFER variable is limited to 0.5 megabytes.

To determine the potential benefit of increasing the buffer size, you need to know the length of the data that is being transferred by the **driver**. For example, if you execute the following statement:

```
    630   PRINT PI,PI,PI,PI          !Print Pi four times.
```

the output line, or record, is 77 bytes. The record is not 32 bytes (four REAL values). The PRINT statement converts the REAL values of PI to characters (13 bytes each), prefixes them with a blank because they are positive (1 byte each), postfixes them with a blank because they are numbers (1 byte each), follows the first three with trailing blanks to the next default field (5 bytes each) and terminates the line with a CR-LF (2 bytes). The output record is constructed in the internal buffer by the PRINT statement. The 77 byte buffer is printed by the driver.

As long as the record is smaller than the buffer, the run time support code calls the driver (through the IOS) only once to transfer the data. If the record is larger than the buffer, the driver is called as many times as required to handle each fragment of the record. On a peripheral like a printer, you may not notice this extra overhead. If the standard printer is an HP 7970E or 7971A mag tape, your output record is written as several small records. This can severely slow down a data logging activity.

To avoid extra overhead, the buffer should always be at least as long as your logical records. For an I/O path name or the standard printer, you can change the size of the internal system buffer(s) with the BUFFERSIZE I/O attribute, for example:

```
    20   ASSIGN @Black_box TO 921;BUFFER SIZE 1024  !1K record size
```

Multiple Records in One Buffer - BUFFERED ON/OFF

If your records are smaller than the buffer size, it is possible for your BASIC I/O statements to write more than one record's worth of data into the buffer before passing the buffer to the IOS and drivers for output. This can improve I/O performance by further reducing the number of driver operations to transfer a given amount of data.

When buffering is off (BUFFERED OFF, the default), the contents of the buffer are written (posted) to the destination after each I/O statement. With BUFFERED OFF, no more than one statement's worth of I/O data is held in the buffer, regardless of the buffer size.

You enable the buffering of multiple records with the BUFFERED ON/OFF I/O attribute, for example:

```
    30   PRINTER IS 406;BUFFERED ON,BUFFER SIZE 4920!Page buffer.
```

When BUFFERED ON is in effect, the buffer is only posted when:

- the buffer is full;
- the output statement sends an EOL(DELAY or END);
- you explicitly or implicitly (SUBEXIT, STOP) close the I/O path name or change the PRINTER IS assignment;

Unless you output an amount of data which is an exact multiple of the buffer size, some data is always "left over" in the buffer. Unless you force posting of this remaining data by closing the I/O path name or altering the PRINTER IS assignment, this data may not be output until the program terminates.

If you press STOP to terminate a program which has unposted data in I/O buffers, the buffers are not posted[1]. The message "I/O ABORTED" appears in the MESSAGES SCREEN to alert you. The ABORTIO statement also deletes (flushes) unposted buffer data.

Concurrent I/O and Program Execution - OVERLAP

To take further advantage of the buffering features described in the previous sections, the I/O concurrency mode of your program (partition) should be OVERLAP rather than SERIAL. OVERLAP is both a BASIC statement and an I/O attribute. SERIAL is the default[2] after power-up, SCRATCH A and partition creation.

When OVERLAP is in effect, a system process called the "overlap process" supervises the passing of data in system buffers to the IOS and drivers. The overlap process is an "invisible" program which executes concurrently with your program, sharing CPU time.

When you perform overlapped output, your BASIC I/O statement begins filling the buffer(s). When the buffer is full (BUFFERED ON) or at statement completion (BUFFERED OFF), the overlap process initiates output of the buffer data to the destination. While this I/O is in progress, your program can add more data to a different buffer or execute code unrelated to the I/O operation.

For OVERLAP to be in effect, both the program's and the I/O path name's or standard printer's modes must be OVERLAP. You set the program's mode with the OVERLAP statement. You set the I/O path name or standard printer mode with the OVERLAP I/O attribute.

When SERIAL is in effect, your program suspends while the output is in progress. For a large buffer and a slow device, this can be a significant period of time. You can establish the SERIAL mode with either the SERIAL statement or the SERIAL I/O attribute. The SERIAL statement affects all output in your program. The SERIAL I/O attribute affects only the standard printer or specified I/O path name.

1 I/O involving I/O path names having the NONSTOP attribute continues after pressing STOP See the "Overlapped I/O and STOP" section of this chapter.

2 The I/O concurrency mode is always SERIAL for the PRINT ALL printer.

Overlapped I/O and STOP

If your program performs overlapped I/O, that I/O may still be in progress when your program reaches a STOP or END statement. The result depends on the type of STOP operation:

- A STOP or END statement executed in the MAIN program suspends the program until all overlapped I/O has completed. If any ON conditions occur as a result of the I/O, any enabled branches which handle those conditions occur.
- A STOP statement in a subprogram suspends the program until all overlapped I/O has completed. If the completion of any I/O operation results in ON conditions occuring, those ON conditions are ignored.
- The (STOP) key aborts all overlapped I/O in progress for I/O resources which have the STOP I/O attribute in effect. For I/O resources which have the NONSTOP I/O attribute in effect, the program is stopped, but the overlapped I/O continues. The RUN LIGHT remains because the overlap process is still active in the partition. If there is a problem with the I/O, such as printer off line and no timeout in effect, you may have to power down the computer to regain control.

Multiple I/O Buffers - MAXBUFFS

A disadvantage of a single large buffer is that once the buffer is full no output statement can add more data to it until the driver has finished transferring data from the buffer to the device. A way to avoid this is to establish more than one buffer. With multiple buffers, when one is full, the output statement switches to an empty one and continues.

You can change the number of buffers used for I/O path name and standard printer I/O with the MAXBUFFS I/O attribute, for example:

```
20   PRINTER IS 406;BUFFER SIZE 82,MAXBUFFS 60   !4920 bytes.
```

The use of multiple buffers requires that both the OVERLAP mode and the OVERLAP attribute be in effect.

Using Declared BUFFERs

Using a declared BUFFER provides some capabilites not available with internal system buffers. Here are two typical uses for declared BUFFERs:

- You can write to and read from an I/O path name assigned to memory (a BUFFER) as if it were a device.
- You can perform actual I/O with TRANSFER, which is the highest performance I/O statement.

This section discusses the use of BUFFERs as an I/O resource. The next section discusses the use of BUFFERs with the TRANSFER statement.

Creating a BUFFER I/O Resource

The first step in using a declared BUFFER is to create the BUFFER and assign it an I/O path name. If you choose to have the system allocate the BUFFER, you can create and assign the BUFFER in a single ASSIGN @ statement. This type of BUFFER is called an **unnamed** BUFFER. Having the system allocate the BUFFER has the advantage that you can release the memory when you no longer need the BUFFER. It has the disadvantage that you can access data in the BUFFER only with I/O statements. Here is an example of a system allocated BUFFER.

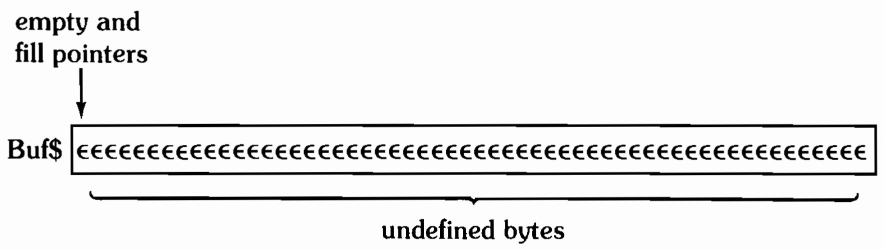
```
10   ASSIGN @Buf TO BUFFER [56]     !Allocate a 56 byte BUFFER
```

You can also create a BUFFER with the storage allocation statements DIM, INTEGER, DOUBLE, SHORT, REAL and COM (but not ALLOCATE). To make any simple string or numeric array a BUFFER, add the secondary keyword "BUFFER" to the dimensioning statement. You cannot use a string array as a BUFFER. It is suggested that you use a simple string, rather than a numeric array, as a BUFFER. A numeric array is useful as a BUFFER only if you must also access it as a numeric array.

If you use the BASIC name of a numeric array or string, the BUFFER is called a **named** BUFFER. Here is an example of a named string BUFFER.

```
10   DIM Buf$[56] BUFFER            !Dimension string as BUFFER
20   ASSIGN @Buf TO BUFFER Buf$     !Assign it a path name
```

After the BUFFER is created and assigned, it is **empty** for I/O purposes. Each BUFFER byte contains some value, of course, but this value is undefined. An undefined byte is represented by a lower case epsilon (ϵ) in the diagrams of this section. If the BUFFER is a string, its LEN (as a string) is set to the dimensioned string length.



A BUFFER has a pair of **pointers** which denote the beginning of valid data and the beginning of invalid data. The start of valid data is denoted by the **empty pointer**. I/O operations which read from (empty) the BUFFER start at this byte. The start of the empty portion of the BUFFER is denoted by the **fill pointer**. It points to the first byte after the last valid data byte. I/O operations which add data to (fill) the buffer start at this byte. The preceding diagram shows the pointers for "Buf$" after creation and assignment. "Buf$" is empty because the pointers denote the same byte.

BUFFER I/O Attributes

You use the ASSIGN @ statement to associate an I/O path name with the named or unnamed BUFFER. When you assign the I/O path name you can also specify I/O attributes. For example, the following assignment cancels the EOL. This allows you to output to the BUFFER without the need to suppress EOL. If you do not do this (or suppress EOL), I/O statements which perform automatic EOL write a CR-LF in the BUFFER following the data.

```
20  ASSIGN @Just_data TO BUFFER [80];EOL ""  !No eol this BUFFER
```

Unlike a device or file, some of the attributes do not apply to a BUFFER. Because the BUFFER **is** the I/O buffer for both input and output to the associated I/O path name, the BUFFERED, BUFFERSIZE and MAXBUFFS I/O attributes are ignored. Because no interface or device is involved in BUFFER I/O, the CHECKREAD, DRIVER, HOLDOFF, OVERLAP, SERIAL and XFERRATE attributes are ignored. EOL...DELAY and EOL...END are provided for devices and are ignored with BUFFERs.

All other I/O attributes are observed as if the BUFFER were a device. Support for the various attributes is summarized under the topic "BUFFER" in the "I/O Resources" section of the BASIC Language Reference manual.

"Empty" BUFFER Contents

It is possible to access the bytes in the "undefined" portion of the BUFFER. You can move the pointers with the CONTROL statement. If it is a named BUFFER, you can retrieve elements with variable assignment. If you do this, you can expect the following results.

- A named BUFFER is preset to binary zeros. Recall that dimensioning a string as a BUFFER causes its current length to be initially set to the dimensioned length of the string rather than zero. Binary zero is the ASCII NUL character.
- An unnamed BUFFER contains whatever was in the computer memory at that location. This is the repeating string "Nil " at power-up and SCRATCH A, but can be any "garbage" data after other operations.
- Reading (emptying) data from a BUFFER does not change the value of the affected bytes until those bytes are overwritten by subsequent writing (filling) of the BUFFER.

Adding Data to a BUFFER

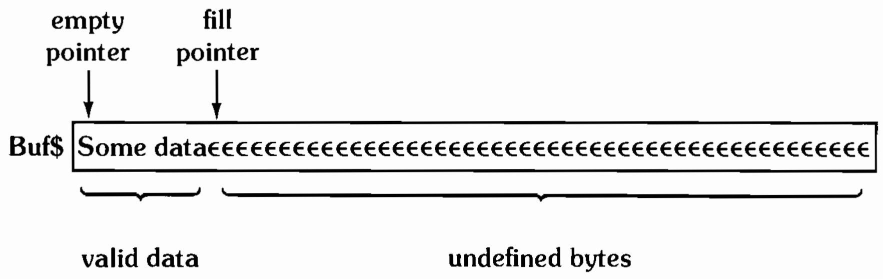
There are three ways that you can add data to a BUFFER:

- You can output program expressions to the BUFFER with the OUTPUT, OUTPUT USING or OUTPUTBIN statements specifying the I/O path name of the BUFFER.
- You can transfer data from another (non-BUFFER) I/O resource to the BUFFER with a TRANSFER statement specifying the I/O path name of the BUFFER as the destination.
- If it is a named BUFFER, you can assign program expressions to the BUFFER with variable or substring assignment. Note that this operation does **not** update the BUFFER pointers.

The following statement outputs nine characters to the BUFFER. To simplify the example, the OUTPUT statement has a trailing semicolon to suppress EOL. This and the other examples of this section use the 56 byte string BUFFER "Buf$".

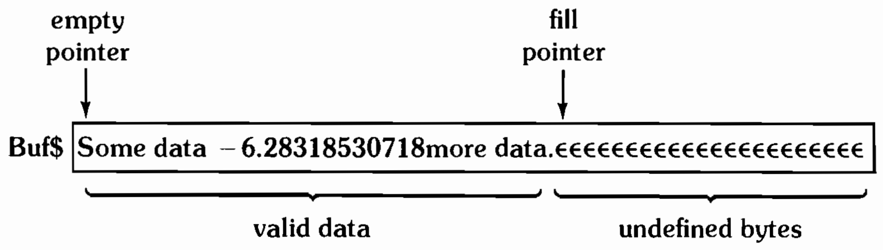
```
30   OUTPUT @Buf;"Some data";     !Write data to BUFFER.
```

This statement sends the string "Some data" to the BUFFER starting at byte 1, which was the original position of the fill pointer. After the output operation, the value of the fill pointer is advanced to byte 10. The result of the OUTPUT operation is summarized in the following diagram.



As you output more data to the BUFFER, you continue to advance the fill pointer.

```
40   OUTPUT @Buf;-2*PI;"more data."; !Output numeric & text data.
```



If the BUFFER is a string, output to the BUFFER (or any I/O operation involving the BUFFER) does not change the current length of the string. It remains set to the defined length of the string unless you change it with a string or substring assignment operation. Conversely, substring assignment to the string variable does not affect the BUFFER pointers.

Retrieving Data from a BUFFER

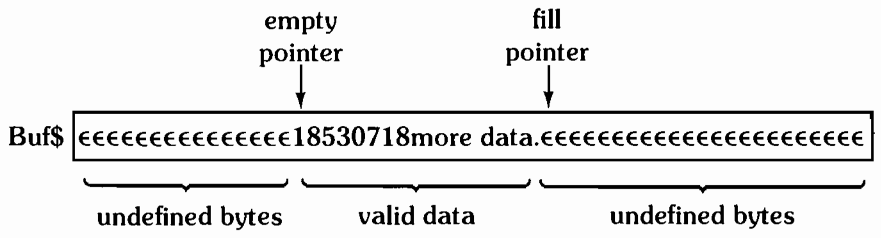
As with adding data to a BUFFER, there are three ways to retrieve data from a BUFFER:

- You can assign BUFFER data to program variables with an ENTER, ENTERBIN or ENTER USING statement specifying the I/O path name of the BUFFER. This does not affect the string's current length.
- You can transfer BUFFER data to another (non-BUFFER) I/O resource with a TRANSFER statement specifying the I/O path name of the BUFFER as the source.
- If it is a named BUFFER, you can transfer data from the BUFFER to program variables with simple assignment or the ENTER and ENTER USING statements. Note that this does not update the BUFFER pointers.

The following statement transfers the value of 15 bytes in the BUFFER to a sub-string of "String$".

```
50   ENTER @Buf USING "#,-K";String$[1,15]   !Read 15 bytes
```

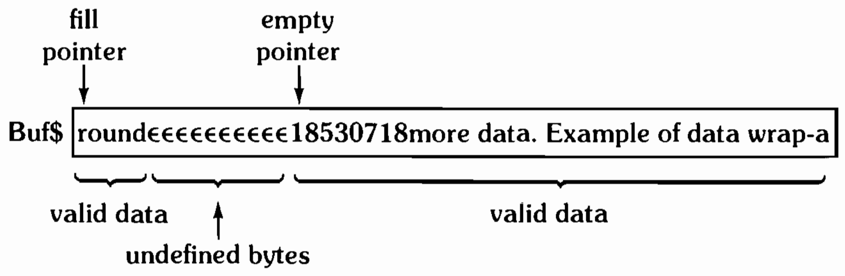
After the execution of the ENTER statement, the empty pointer is advanced to the next un-read data byte. The first 15 bytes of the BUFFER are now undefined for I/O purposes.



BUFFER Wrap-around

In I/O operations involving a BUFFER, the only bounds which are significant are the pointers. As long as there is a difference between the empty and fill pointers (or the BUFFER is empty), you can continue to output data to the BUFFER. When the output operation reaches the end of the BUFFER's memory block, it continues at the beginning of the memory block.

```
    60   OUTPUT @Buf;" Example of data wrap-around";
```



The converse is true of input from a BUFFER. If the ENTER or ENTER USING statement requires more bytes than remain to the end of the memory block, the statement continues reading at the beginning of the block.

If you access the BUFFER as a numeric array or string, wrap around does **not** occur. As a variable, a named BUFFER is treated no differently than any other variable.

BUFFER END

An attempt to write to a full BUFFER or read from an empty BUFFER with an I/O statement causes an END error. When a BUFFER is full or empty both of the pointers denote the same byte. Using the example BUFFER as it was left by the "wrap-around" example, the following statement causes an END error.

```
    70   OUTPUT @Buf;"longer than empty area";   !Causes END error.
```

Although an END error occurs, the data is still written to the BUFFER, up to the byte at which the error occurred. The END error does not occur if there is an TRANSFER active outbound from the BUFFER. See the "BUFFER and I/O Resource TRANSFER" section of this chapter.

Accessing BUFFER Pointers

A BUFFER I/O path name has 13 status and 5 control registers. Four of these are associated with pointer operations. Register 3 (r3) is the fill pointer. Register 5 (r5) is the empty pointer. Register 2 (r2) is the BUFFER size. Register 4 (r4) is the number of valid data bytes[1].

1 Register 4 is necessary because it is not possible to determine from r3 and r5 alone whether the BUFFER is full or empty when r3 = r5.

You can determine how many unused bytes remain in the BUFFER by subtracting r4 from r2, for example:

```
100   R2=IOSTAT(@Buf,2)                               !Get BUFFER size
110   R4=IOSTAT(@Buf,4)                               !Get contents
120   IF (R2-R4)>LEN(Data$) THEN OUTPUT @Buf;Data$;  !OK to add data?
```

Pointer Updating
If you use an I/O statement to access a BUFFER, the pointers are updated on statement completion. Since the OUTPUT, OUTPUT USING, OUTPUTBIN, ENTER, ENTER USING and ENTERBIN statements execute as a single operation, there is no opportunity to access the pointers while the statement is in progress. This is not true of the TRANSFER statement, which can be in progress while you are examining or changing the pointers. TRANSFER is discussed in the "BUFFER and I/O Resource" section of this chapter.

The TRANSFER statement updates the pointers in three circumstances:

- on completion of the TRANSFER statement;
- whenever a BUFFER "wrap around" occurs;
- whenever an end-of-record (EOR) occurs during the TRANSFER.

Changing the BUFFER Pointers
You can change the value of the fill and empty pointers and the valid data count. You cannot change them under all circumstances. If a TRANSFER I/O operation is in progress in the BUFFER, at least one of the pointers is "busy". You cannot change a busy pointer.

To change the value of one or more of these registers, use the CONTROL statement. The following example declares the BUFFER to be empty. (You can accomplish a similar result with the RESET statement.)

```
450  CONTROL @Buf,4;0     !Set valid data length to zero.
```

Because of the implicit mathematical relationship between r3, r4 and r5, the system imposes some rules on how CONTROL register values affect actual register values. There is a table under the "BUFFER" topic in the "I/O Resources" section of the BASIC Language Reference manual which summarizes the rules. For example, if the r3 pointer is busy in the previous example, the system cannot change it equal to r5; it sets r5 equal to r3 instead.

Additional BUFFER register techniques concerned with TRANSFER operations are discussed in the next section.

BUFFER and I/O Resource TRANSFER

The previous section, "Using Declared BUFFERs", discussed the transfer of data between your program and a BUFFER. This is only one half of an I/O operation involving an I/O resource. The other half is the transfer of data between the BUFFER and the I/O resource. The BASIC Language System provides a statement whose sole purpose is to move data between a BUFFER and another I/O resource: the TRANSFER statement.

The TRANSFER statement specifies two I/O resources; a **source** and a **destination**. TRANSFER also has several expressions, or attributes, which you can use to control various aspects of the transfer operation. Here is an example of a simple TRANSFER, using the final example BUFFER from the previous section.

```
 30  ASSIGN @Disp TO CRT      !Assign path name to CRT.
     .
     .
     .
560  TRANSFER @Buf TO @Disp   !Move BUFFER data to CRT.
```

This displays the following message on the CRT:

```
18530718more data. Example of data wrap-around
```

TRANSFER offers several capabilities not available with other I/O statements. Here are three:

- Each TRANSFER statement is a separate overlapped process which normally executes concurrently with your program. You can use TRANSFER to perform **overlapped input**, unlike all other input statements.
- You can TRANSFER data into and out of the same BUFFER concurrently. This provides the capability to perform simple device-to-file, file-to-device and device-to-device data transfers.
- TRANSFER provides higher data transfer rates than any other I/O statement.

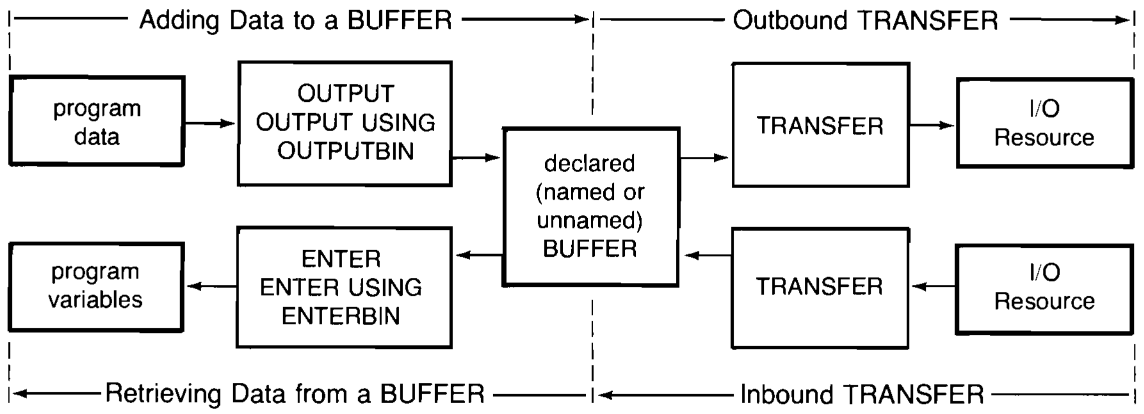
The TRANSFER statement is a simple "data mover". It performs no data conversions or reformatting. Data is transferred out of a BUFFER in the exact form that it appears in the buffer. If it is a named BUFFER, the BUFFER variable type is irrelevant. Data is transferred into a BUFFER in the form supplied by the interface, device or BDAT file. TRANSFER can detect DELIM characters, if specified, and transfers the character to the BUFFER. TRANSFER can detect END conditions, if specifed, but cannot transfer an END condition to the BUFFER because an END cannot be represented as data.

TRANSFER Data Paths

In a TRANSFER operation, both the source and destination must be specified by an I/O path name. One of the I/O path names must be assigned to a BUFFER. The other must not be assigned to a BUFFER; that is, it must be assigned to a device selector or file specifier. You cannot use a single TRANSFER statement to perform device to device, file to file, device to file or BUFFER to BUFFER data transfers[1].

1 You can perform these operations using two concurrent TRANSFERs specifying the same BUFFER but different non-BUFFER resources.

The following diagram shows the data flow in complete TRANSFER operations. The direction of data flow during TRANSFER is always described with respect to the BUFFER. Since it is possible to have concurrent TRANSFERs to and from the BUFFER, not involving program data or variables, the terms input and output are imprecise. TRANSFER operations are described in terms of whether data is **inbound** to the BUFFER or **outbound** from it.



TRANSFER Attributes

For the I/O path names involved, TRANSFER ignores all I/O attributes which specify data conversion or parsing. Outbound data must already be in the form required by the destination I/O resource. Inbound data is in the form supplied by the source I/O resource. If you need to parse or convert the BUFFER data, use the operations described in the "Using Declared BUFFERs" section.

TRANSFER ignores **all** I/O attributes of the BUFFER I/O path name. TRANSFER does observe those I/O attributes of the non-BUFFER I/O path name which are related to the activity of the driver(s). The following table summarizes TRANSFER's observance of I/O path name I/O attributes.

I/O Attribute	Interface or Device	File	Comments
BUFFERED	n.a.	ON	
BUFFERSIZE	n.a.	Observed	
BYTE/WORD	Observed	Observed	Affects BUFFER pointers
CHECKREAD	Always OFF	Always OFF	
DRIVER	Observed	Observed	
DELIM	n.a.	n.a.	Use TRANSFER's DELIM
EOL	Always OFF	Always OFF	
FORMAT	n.a.	n.a.	
HOLDOFF	Observed	n.a.	
MAXBUFFS	n.a.	Observed	
PARITY	Always OFF	Always OFF	
SERIAL/OVERLAP	n.a.	n.a.	
STOP/NONSTOP	Observed	Observed	Use Caution (see text)
WIDTH	Always OFF	Always OFF	
WRAP	Always OFF	Always OFF	
XFERRATE	Observed	n.a.	

To provide you with control over the length of the TRANSFER and provide indications of its progress, you can specify TRANSFER attributes in the TRANSFER statement. The following table summarizes their defaults.

TRANSFER Attribute	Outbound Default	Inbound Default	Comments on default state
CONT	Off	Off	TRANSFER stops on EOT condition
WAIT	Off	Off	TRANSFER is overlapped
EOR(DELIM)	n.a.	None	All characters are data
EOR(COUNT)	None	None	No byte counting performed
EOR(END)	None	None	END condition terminates TRANSFER
DELIM	n.a.	None	All characters are data
COUNT	None	None	No byte counting performed
END	None	END	END condition terminates TRANSFER
RECORDS	None	None	No record counting performed

TRANSFER Termination - EOT

If you do not specify any TRANSFER attributes, an outbound TRANSFER terminates when the BUFFER is empty; that is, when the empty pointer reaches the fill pointer. An inbound transfer terminates when the BUFFER if full or when a device dependent END condition is detected. A condition that terminates a TRANSFER is called an end-of-transfer or EOT condition.

If you want an EOT to occur on other conditions or without causing an error on input, you must specify a TRANSFER EOT attribute. If you define an EOT condition, your program can branch on the occurence of an EOT with ON EOT or wait for the end of a TRANSFER with WAIT FOR EOT or WAIT.

Outbound EOT

If you specify no TRANSFER EOT attributes, the TRANSFER terminates and signals EOT on BUFFER empty. It does not cause an error.

If you want the TRANSFER to terminate after transferring a specific number of data bytes, use the **COUNT** expression. The following example terminates and signals EOT after transferring "Max-bytes" bytes or on BUFFER empty, whichever occurs first.

```
890   TRANSFER @Buffer TO @Dest;COUNT Maxbytes !End on count
```

You can also define EOT to occur after a specified number of RECORDS have been transferred. To use the **RECORDS** attribute, you must define what a record is with the EOR attribute. Outbound, you can only use the **EOR(...COUNT...)** attribute. In the following example, a record is defined as 80 bytes and EOT occurs after 24 records have been transferred (or on BUFFER empty).

```
940   TRANSFER @From TO @To;EOR(COUNT 80),RECORDS 24 !EOT=1920 bytes
```

You can also specify that when EOT occurs (however defined) TRANSFER is to assert a file or device dependent END condition. You specify this with the **END** attribute. For TRANSFER to a BDAT file, END updates the file's EOF record and EOF byte pointers to denote the last byte transferred. For TRANSFER to HP-IB, END asserts the EOI bus line TRUE with the last transferred data byte.

Inbound EOT

If you specify no TRANSFER EOT attributes the TRANSFER terminates on a source END condition or destination (BUFFER) full. Either of these conditions also causes an END error (which you can trap with ON END or ON ERROR).

If you want the EOT to occur after transferring a specified number of bytes, use the **COUNT** attribute. The semantics of COUNT are the same for both inbound and outbound TRANSFER.

If you want the EOT to occur on detection of a specific character in the data, use the **DELIM** attribute. The DELIM expression specifies a string expression which evaluates to a single character. In the following example, EOT occurs on detection of a line feed character in the data.

```
    770  TRANSFER @Bus TO @Buf;DELIM CHR$(10)!EOT on ASCII LF
```

Note that since TRANSFER observes the BYTE/WORD I/O attribute of the non-BUFFER ("@Bus") I/O path name, TRANSFER DELIM is only valid if the I/O attribute is BYTE. TRANSFER DELIM causes an ERROR 606 (DELIM not supported) if the source is in **WORD** mode.

If you want the EOT to occur on detection of a source END condition, use the **END** attribute. Note that this differs from the semantics of END in the outbound case. Outbound, END is **asserted** at EOT. Inbound, END **causes** an EOT. Since an END condition on an inbound TRANSFER is an EOT condition by default, the END attribute merely serves to inhibit the END error that also occurs in the default case. If you specify END, an inbound END causes EOT, but no error and no ON END branch. The following example transfers data from a BDAT file until the end of file.

```
    170  TRANSFER @Bdat_file TO @Buf; END  !Transfer whole file.
```

This example assumes that the BUFFER is large enough to hold the file data. If this is not true, an END error still occurs on BUFFER full. To inhibit this error you must use the CONT attribute. CONT is discussed in the "Continous TRANSFERs" section of this chapter.

As with outbound TRANSFER, you can specify inbound EOT on a record count with the **RECORDS** attribute. As with outbound RECORDS, you must define what a record is with the **EOR** attribute. You can use an EOR(...COUNT...), EOR(...DELIM...), EOR(...END...) or a combination of EOR terms. The following example reads data from an HP-IB device and treats the comma character as an EOR. The example accepts ten readings from the device, each separated by a comma.

```
    340  TRANSFER @Dev TO @Buf;EOR(DELIM ","),COUNT (10) !Get 10 rdgs
```

You should not specify the same character for EOR(...DELIM...) and EOT DELIM. If you do, the first occurrence of the character causes **both** an EOR and an EOT, terminating the TRANSFER. The same is true for EOR(...END...), which defines EOR to occur on detection of an END condition. In both the DELIM and EOR(...DELIM...) cases, the DELIM character is passed to the BUFFER as data before the ON condition is signalled.

In a TRANSFER from an interface or device, specifying different EOR and EOT DELIM characters results in slow performance because the driver is requested to read only one character at a time.

Detecting EOR and EOT

It is very useful to know when an overlapped TRANSFER statement has terminated. A typical use of TRANSFER is to initiate an outbound data transfer and then perform some other tasks in your program. When you finish those tasks, you may want to check the TRANSFER to see if it has completed. There are several ways you can do this, two of the most appropriate are with the ON EOT and WAIT FOR EOT statements.

In the following example, when the program completes the "other tasks", it checks to see if the TRANSFER is still active (BUFFER register 7) and if it is, the program waits for the EOT.

```
330   TRANSFER @Buf TO @Dest    !EOT on BUFFER empty.
      .
      .  Perform other tasks
      .
810   IF NOT IOSTAT(@Buf,7) THEN WAIT FOR EOT @Dest
```

If your program has nothing else to do while waiting for an EOT, you can simply attempt to close the I/O path names involved in the TRANSFER. If the TRANSFER is still active, your program is suspended until EOT, at which time it closes the I/O path names.

If the TRANSFER is more important than the "other tasks" your program is performing, you may not want to wait for EOT. In the following example, the program is "interrupted" by the EOT so that it can process the data now available.

```
240   ON EOT @Dest,9 GOSUB Get_the_data  !Branch on data avail..
250   GOSUB Get_the_data                 !Start the 1st TRANSFER
      .
      . Perform other tasks
      .
910   Get_the_data:                       !Service routine
      .
      . Retrieve and process data from BUFFER
      .
970   TRANSFER @Dest TO @Buf; END        !EOT on source END
980   RETURN                             !As you were
```

The termination of a TRANSFER statement signals an EOT, whether or not you specified any EOT attributes. The termination can occur normally or as a result of an ABORTIO statement or a STOP statement. EOT is signalled in each case. In the STOP case, this means that the program might not stop until all ON EOT service routines complete.

ON EOT and WAIT FOR EOT statements must specify the I/O path name of the non-BUFFER end of the TRANSFER. EOT is not signalled for a BUFFER.

EOR is signalled only by the occurrence of a condition defined in an EOR(...) attribute of the TRANSFER statement. If an EOT occurs before the occurrence of the EOR condition(s), no EOR is signalled. However, to prevent your program from suspending indefinitely at a WAIT FOR EOR statement in this circumstance, an EOT releases a WAIT FOR EOR.

Simulations - CAUSE EOR/EOT

If you want to test your ON EOR/EOT and WAIT FOR EOR/EOT service routines, you can use the CAUSE EOR and CAUSE EOT statements. You can simulate an ON branch with a CAUSE statement or command. You can release a WAIT FOR statement only with a CAUSE statement executed from the keyboard.

Only one actual EOT can occur during a TRANSFER, regardless of whether the ON EOT branch is enabled. If you execute multiple CAUSE EOT statements while the ON EOT branch is disabled, you increment the I/O path name's WAIT FOR EOT semaphore, but only one ON EOT can occur.

Non-overlapped TRANSFERs - WAIT

When you want a TRANSFER statement to complete before you execute the next program statement, use the WAIT attribute, for example:

```
570  TRANSFER @Buf TO @File; END,WAIT !Final outbound
```

The WAIT attribute is necessary because the TRANSFER statement ignores both the SERIAL/OVERLAP statements and the SERIAL/OVERLAP I/O attributes of the I/O path names.

Continuous TRANSFERs - CONT

TRANSFER statement I/O is overlapped unless you specify the WAIT attribute. For each active TRANSFER statement, the BASIC Language System creates a system program (a process) which monitors the progress of that TRANSFER. A TRANSFER process is responsible for issuing the I/O requests through the IOS, detecting EOR/EOT conditions and updating BUFFER pointers.

A TRANSFER statement normally terminates and signals EOT when the BUFFER is empty (outbound) or full (inbound). If you specify the CONT attribute, this EOT does not occur. Instead, the TRANSFER process suspends. On outbound TRANSFERs, the process suspends until more data is available in the BUFFER. On inbound data TRANSFERs, the process suspends until some or all of the data is removed from the buffer. When your OUTPUT @Buf or ENTER @Buf statement (for example) has changed the BUFFER pointers, the TRANSFER process resumes. A continuous TRANSFER terminates only on an EOT condition other than BUFFER full or empty.

Here is an example of a continuous outbound TRANSFER. The TRANSFER is set up to output data to a printer whenever data is available. When no data is available, the TRANSFER simply waits. Each output to the BUFFER's I/O path name adds new data to the BUFFER.

```
10   DIM Spool$[8192] BUFFER          !Out-spool buffer.
20   ASSIGN @Spool TO BUFFER Spool$   !Open I/O path name.
30   ASSIGN @Printer TO 406           !Actual printer.
40   TRANSFER @Spool TO @Printer;CONT !Start TRANSFER
      .
      .
      .
910  OUTPUT @Spool;Text_line$          !Output text, EOL
```

There is an interaction between OUTPUT, OUTPUT USING, OUTPUTBIN and continuous TRANSFERs. In an earlier example, the program line tested the BUFFER to see if there was room for the OUTPUT statement's data. This is only necessary if no TRANSFER is actively transferring (or waiting to transfer) data from a BUFFER. If there is not enough BUFFER space and a TRANSFER is active, the output statement suspends until enough space becomes available. In the case of a continous TRANSFER, the TRANSFER is active until EOT.

There is a similar relationship between ENTER, ENTER USING, ENTERBIN and TRANSFER. If the BUFFER is empty, but a TRANSFER is active, the entry statement suspends until data is available.

A continuous TRANSFER terminates only on an EOT condition. The CONT attribute **cancels** buffer empty and buffer full as EOT conditions. Therefore, you should specify another EOT condition to ensure that the TRANSFER eventually terminates. If the previous example executes a STOP or END statement, the TRANSFER is still in progress, even though no data may be in the BUFFER. The RUN LIGHT is still , indicating that your program is running. To stop the TRANSFER process, you must press (STOP). This causes an implied ABORTIO.

There are ways that you can terminate the TRANSFER even though it has no EOT attributes. You can cancel the TRANSFER with an ABORTIO specifying the I/O path name of the BUFFER or you can cancel the "CONT" attribute with a CONTROL operation specifying BUFFER register 9, for example:

```
    990   CONTROL @Spool,9;0    !Re-activate EOT on BUFFER empty.
```

ABORTIO terminates a TRANSFER immediately. CONTROL...,9;0 simply cancels the CONT attribute. The TRANSFER terminates on the next BUFFER empty condition.

Note

If your program does not terminate continuous TRANSFERs before executing a STOP or END statement, there is a potential inconvenience. If the non-BUFFER I/O path name has the NONSTOP I/O attribute in effect, the (STOP) key does not cause an implied ABORTIO of the TRANSFER. Moreover, since your program is not running, but the partition is still in the run state, you can no longer execute any I/O statements which might terminate the TRANSFER. Unless you can make another partition the foreground partition with ATTACH statement or (NEXT PART), you must power-down the computer to regain control.

Simultaneous TRANSFERS

An earlier section pointed out that device to device, file to device and file to file TRANSFERs are not allowed in a single TRANSFER statement. However, two TRANSFER statements can be active in the same BUFFER as long as one is inbound and the other outbound. This means that these disallowed TRANSFERs can be accomplished with a pair of TRANSFER statements.

The following example copies data from a BDAT file to the internal printer with two continuous (CONT) transfers. The first TRANSFERs from the BUFFER to the printer and has no EOT conditions specified. The second TRANSFER fills the BUFFER from the file and signals EOT on end of file. The EOT causes a branch to the CONTROL statement which cancels the CONT mode of the first TRANSFER. When the last data is posted from the BUFFER to the printer, an EOT occurs at which time the I/O path names are closed.

```
 10  ASSIGN @File TO "BDAT_FILE:INTERNAL" !Source data file.
 20  ASSIGN @Buf TO BUFFER [80]           !Record size buffer.
 30  ASSIGN @List TO PRT                  !Destination printer.
 40  ON EOT @File,15 GOTO Eot             !Trap file EOT
 50  !
 60  TRANSFER @Buf TO @List;CONT          !Start outbound xfer
 70  TRANSFER @File TO @Buf;CONT, END     !Start inbound xfer
 80  !
 90  FOR Index=1 TO 2147483647            !Do other tasks
100    Nop=Nop
110  NEXT Index
120  WAIT FOR EOT @File
130  !
140 Eot:CONTROL @Buf,9;0                  !EOT on BUFFER empty
150  ASSIGN @File TO *                    !Close file now
160  ASSIGN @Buf TO *                     !Close BUFFER on EOT
170  ASSIGN @List TO *                    !Close List on EOT
180  END
```

I/O Timeouts - ON TIMEOUT

Should an I/O device fail or enter a non-responsive state, your program can suspend indefinitely when it performs I/O to or from the device. For example, if the HP 2631B printer runs out of paper, it ceases accepting data on the bus. If you are performing serial output to the printer, the output statement suspends. The state of suspension lasts until you load more paper in the printer and press its ON LINE key.

In some cases, you can perform a STATUS operation to determine that a device is ready. There is no way to determine if a 2631B is about to run out of paper. To avoid program suspension, you can specify that an error (168) is to occur if the interface or device fails to respond within a certain period of time. The occurrence of this error is called a **timeout**. You define it with the ON TIMEOUT statement.

In the following example, a timeout occurs if an I/O operation to the printer does not complete within three seconds. The timeout causes the subprogram "Reset_list" to be called.

```
 10   ASSIGN @List TO 406
 20   ON TIMEOUT @List,1 CALL Reset_list !Set up 3 second timeout
  .
  .
  .
390   OUTPUT @List;Text_data$          !Typical I/O
```

When the timeout occurs, the I/O operation that failed (the OUTPUT) is terminated. The following example "Reset_list" subprogram displays a message and pauses for the operator to place the printer back on-line.

```
1450  SUB Reset_list                              !Entered on timeout
1460    BEEP
1470    DISP "2631B Timeout, Correct problem and press CONT"
1480    PAUSE                                     !Wait for CONT
1490    SUBEXIT
1500  SUBEND
```

When an ON TIMEOUT GOSUB or ON TIMEOUT CALL service routine completes, the line to which program execution returns depends on how the error was detected. In serial I/O operations, such as ENTER, the timeout terminates the statement, and the timeout is detected at the SOLC of the next program line. The service routine returns to that next program line.

In overlapped I/O operations, the timeout is detected in the SOLC of the next statement which accesses the I/O resource specified in the ON TIMEOUT statement. The service routine returns to the line which is accessing the resource. Although an overlapped timeout might not be detected until you close an I/O path name, the ON TIMEOUT branch does return to the close operation. There is no automatic method for restarting the aborted I/O operation.

Resources and Timeouts

Timeouts apply only to interfaces and devices (except KBD). You cannot specify ON TIMEOUT for a file[1] or a BUFFER. The timeout timer is set to zero at the start of each I/O operation. Some I/O operations involve several IOS requests; for example, each EOR in a TRANSFER. The timer is reset to zero for each request.

1 Files have an internal timeout based on a worst case calculation for the operation involved. A file timeout causes a mass storage error and not an ERROR 168 (timeout).

You should select a timeout value which considers the worst case normal operation of the device, not just its nominal transfer speeds. For example, the HP 7971 tape drive requires less than a second to read or write the longest record without error. Retries due to marginal tape can increase this to several seconds, and a rewind can take five minutes.

You can establish a timeout for an interface by specifying in the ON TIMEOUT statement either an interface select code or an I/O path name assigned to the interface select code. If you specify an interface select code and not an I/O path name, the timeout applies to all devices connected to that interface. If you specify the I/O path name of an interface, the timeout applies only to operations which use the I/O path name.

You can specify a timeout for a primary device address (a device selector) only with an I/O path name assigned to the device. This timeout applies only to operations using that I/O path name. It does not apply to any other I/O path name assigned to the same interface nor to any other device(s) connected to the same interface.

If you have a timeout established for a device's interface select code but not for an I/O path name assigned to the interface or a device connected to the interface, operations involving the I/O path name can cause the interface ON TIMEOUT branch to occur.

The following table summarizes what happens when the timeout period elapses with respect to the various ON TIMEOUT statements which can specify the resources involved. The table uses the following example statements. The "@Dev" I/O path name can be assigned to "Isc" or a device connected to that interface.

```
110  ON TIMEOUT Isc,Time_i GOSUB Interface   !Interface select code
120  ON TIMEOUT @Dev,Time_d GOSUB Device     !I/O path name
```

ON TIMEOUT statements involved		**Branch taken on timeout**	
Status of `ON TIMEOUT Isc` **statement when timeout occurs**	**Status of** `ON TIMEOUT @Dev` **statement when timeout occurs**	**I/O statement specifies Isc; after "Time.." result is...**	**I/O statement specifies @Dev; after "Time.." result is**
inactive	inactive	no timeout	no timeout
inactive	disabled	no timeout	ERROR 168
inactive	enabled	no timeout	GOSUB Device
disabled	inactive or disabled	ERROR 168	ERROR 168
disabled	enabled	ERROR 168	GOSUB Device
enabled	inactive	GOSUB Interface	GOSUB Interface
enabled	disabled	GOSUB Interface	ERROR 168
enabled	enabled	GOSUB Interface	GOSUB Device

If you want to test your ON TIMEOUT service routine without waiting for an actual timeout, you can simulate the timeout with the CAUSE TIMEOUT statement.

Interface Interrupts

I/O interface cards have the capability to alert the computer to conditions not necessarily related to any I/O operation that might be in progress on the interface. The process of alerting the computer is called an **interrupt**, because it diverts the attention of the system and takes priority over many other system activities, such as executing the next statement in your program.

Your program can respond to interrupts with the ON INTR and WAIT FOR INTR statements. When used with the ENABLE INTR statement, these statements provide the branch-on-condition and suspend-until-condition capabilities common to other ON... and WAIT FOR statements.

Ordinary I/O requests like input and output are initiated by the computer. The unique property of an interrupt is that, once enabled the request is initiated by the interface card. An enabled interrupt can occur at any time. It can occur while the interface is idle, or it can occur during an input or output operation.

This section includes two examples of interrupt servicing.

Enabling Interrupts

Several steps are necessary before an interface can signal the computer that an interrupt has occurred and your program can service the interrupt. The following paragraphs list the steps in a suggested order of implementation.

1. Unless you can risk servicing a pre-existing interrupt condition on the interface, you should disable the interface (and device). The DISABLE INTR statement prevents an interface card from generating an interrupt until an ENABLE INTR is given.
2. If you want to have a program branch occur on interrupt, rather than wait for interrupt, you must define the branch with an ON INTR statement. This statement specifies the interface select code of the interface card. You cannot specify a device selector. See the HP-IB interrupt example for a discussion of how to service device interrupts. If you do not know the interface select code, you can obtain it from the I/O path name with the SC function.
3. You must enable the I/O processor (IOP) and interface card for interrupt with the ENABLE INTR statement. This statement specifies an "interrupt mask" which is written to the interface. This mask is an integer in the range of DOUBLE and is described for each interface in the "I/O Resources" section of the BASIC Language Reference Manual. The mask is written to one or more registers on the interface card.

   The interrupt mask is written to the interface only when both an ENABLE INTR and an ON INTR, or an ENABLE INTR and a WAIT FOR INTR, are executed.

   In general, the binary equivalent of the mask must have one or more bits set to "1". Each of several bits enables the interface to interrupt on a specific condition. Although you may be able to access this register with the CONTROL statement and enable the interface, only the ENABLE INTR statement enables the IOP to respond to the interrupt.
4. Enable the device, if any, for interrupt. Some devices require that you set a switch, others require that you send a command or escape sequence, before they can cause the electronic action that the interface recognizes as an interrupt.
5. If you have elected to wait for the interrupt, rather than branch on interrupt, execute a WAIT FOR INTR specifying the interface select code. Note that no keyboard operation except STOP is allowed during WAIT FOR INTR. No other ON... conditions are serviced because the program is suspended, so no SOLC can occur.

Servicing Interrupts

What happens when an interrupt is detected depends on what type of servicing (e.g. ON, WAIT FOR) you have elected to use. How much time elapses between the interrupt and execution of the first (or next) line of your service routine also depends on the type of servicing and the type of PARTITION in which your program is running.

ON INTR Servicing

When you use ON INTR, interrupts are detected only at a start of line check (SOLC). This means that response to an interrupt can be delayed by the amount of time required to execute the longest statement in your program. For example, no interrupts can be serviced for five seconds during this statement:

```
    470  WAIT 5                  !Wait five seconds
```

Like other asynchronous ON conditions, ON INTR can also be disabled by reason of context, priority or the DISABLE statement. While disabled, one interrupt per select code can be logged. When the ON INTR statement becomes enabled, the branch occurs. Note that there is a DISABLE INTR statement. This statement disables the IOP and interface but does not otherwise disable the ON INTR statement. You can still simulate the interrupt with a CAUSE INTR statement.

ON INTR does have the advantage that your program can perform useful tasks while waiting for an interrupt, and it can respond to one of several interrupts. If you need a rapid response, however, you should use WAIT FOR INTR.

WAIT FOR INTR Servicing

WAIT FOR INTR immediately suspends your program until an interrupt occurs on the specified interface. This eliminates the delay of waiting for the next SOLC, but your program can perform no other tasks until the interrupt occurs. Although this provides faster response than ON INTR, it is not the fastest. The shortest interrupt response times are obtained by running your program in an INTR PRIORITY partition. See the ''Partitions and Events'' chapter for information on how to create a partition.

Considerations After Interrupt

An interrupt causes an implied DISABLE INTR. After an interrupt occurs, the ENABLE INTR statement which enabled it is cancelled. If you want to service further interrupts, you must execute an ENABLE INTR statement at the conclusion of your service routine. You should not execute it at the beginning; your service routine might be interrupted by another interrupt.

When **re-enabling** an interrupt with ENABLE INTR, you do not need to respecify the mask unless you want to use a different mask. This is useful when the interrupt is serviced by a subprogram which does not have access to the original mask.

Because you can specify only an interface select code in INTR statements, your service routine must have additional code to identify the interrupting device if more than one device is connected to the interface. This applies particularly to HP-IB interrupts. It is usually not difficult in the case of HP-IB Service Request (SRQ) interrupts because most HP-IB devices confirm that they asserted SRQ when subsequently polled with PPOLL and/or SPOLL.

HP-IB Interrupt Example

Like many HP-IB peripherals and devices, the HP 2631B printer can assert the Service Request (SRQ) line of the bus. The 2631B's SRQ capability is enabled by a switch near the bus connector. If enabled, the 2631B asserts SRQ on several conditions, one of which is paper-out. You can use the 2631B's SRQ capability to service a paper-out condition when it occurs rather than when you detect a timeout.

The HP 27110A HP-IB interface can generate interrupts on a variety of bus conditions, one of which is detection of SRQ. To enable the interface to interrupt on SRQ, you must set a specific bit (12) in the card's interrupt mask register.

The following program lines enable the computer to respond to an HP-IB SRQ interrupt.

```
10   ASSIGN @Hp2631 TO 406;BUFFER SIZE 229,BUFFERED OFF,SERIAL
20   DISABLE INTR 4
30   Poll_mask=1
40   Paper_out_only=IVAL("00010000",2)
50   SEND 4;UNT;UNL;MTA;LISTEN 6;SEC Poll_mask;SEC Paper_out_only
60   ON INTR 4,15 CALL Paper_out             !Define branch
70   ENABLE INTR 4;IVAL("1000000000000",2)   !Set mask bit 12
```

When a paper-out condition subsequently occurs, the 2631B asserts SRQ. At the next SOLC, the branch "CALL Paper_out" occurs. The following is an example of a paper-out service routine.

```
150  SUB Paper_out  !Service 2631B paper-out SRQ <821205.1536>
160    IF BIT(PPOLL(4),7-6) THEN            !If 2631B's SRQ, then
170      IF BIT(ENTERBIN(40614),1) THEN     !If paper-out...
180        OUTPUT CRT;CHR$(7);"Paper out on 2631B."
190        DISP "Please correct and press CONT."
200        PAUSE                            !Wait for operator
210      END IF
220    END IF
230    ENABLE INTR 4                        !Re-enable interface
240    SUBEXIT
250  SUBEND
```

The service routine "Paper_out" does identifies the interrupting device by conducting an HP-IB parallel poll (PPOLL). The 2631B, like most HP-IB devices, responds to parallel poll by asserting the DIO (Data Input/Output) bus line corresponding to its bus address 7-(Address MOD 8), in this case DIO line 1.

Once "Paper_out" has established that the SRQ was indeed from the 2631B, it verifies that the SRQ was caused by a paper-out. It does this by sending the "Io_status" secondary address to the 2631B with the ENTERBIN statement and then testing the paper-out status bit. If it is a paper-out, the subprogram alerts the operator and waits (PAUSE) for correction. Regardless of what caused the 2631B to assert SRQ, "Paper_out" sends the "ON-LINE" escape sequence to the printer to restore it to operation.

Note that this example assumes that your program's printing is overlapped or consists of lines less than 229 characters long. This is the length of the internal buffer of the 2631B. If the output is serial, and the line being sent when paper-out occurs is longer than 229 bytes, the statement suspends when the printer stops "handshaking" data on the bus. Because your program is suspended, no SOLC occurs and the SRQ/interrupt is not detected.

ASI Interrupt Example

In this example there is a terminal near some test equipment which is connected to the computer via an HP 27128A Asynchronous Serial Interface (ASI) card. When batches of units are being tested under computer control, your program analyzes data and displays results on the terminal. At the end of the batch, the operator must signal the computer to generate a summary report and then enter information (on the terminal) about the next batch.

Since the computer does not know when a batch is ended, it does not know when to execute an ENTER statement to read the batch information from the terminal. If an ENTER or inbound TRANSFER from the terminal were simply left pending at all times, that would prevent output of result data to the terminal. The solution is to have the operator use the terminal to interrupt the computer.

Like the HP-IB card, the ASI card can interrupt on a variety of conditions. One condition which is generally used to interrupt computers in serial data communications is the BREAK sequence. A terminal generates a BREAK sequence when you press its **BREAK** key. If you are not familiar with BREAK, there is a summary at the end of this section.

To enable the ASI for BREAK-detect interrupt, you set bit 1 of its mask register to 1. The following lines show most of the MAIN program required for this application[1].

```
100  DISABLE INTR 3                  !Clear any pending interrupt
110  ON INTR 3,15 GOTO End_batch     !Operator pressed BREAK
120  !
130 Start_batch:                     !Start a new batch
140  CALL Enter_info                 !Get operator info
150  ENABLE INTR 3;2                 !Enable INTR on BREAK
160  !
170  LOOP                            !Until BREAK detected
180    CALL Test_one_unit            !ON..GOTO disabled
190  END LOOP                        !ON..GOTO re-enabled
200  !
210 End_batch:                       !BREAK detected
220  CALL Print_report               !Report on current batch
230  GOTO Start_batch                !Start another
```

Note that this example uses an ON INTR GOTO rather than an ON INTR RECOVER. This is to delay the servicing of the BREAK until the current (and final) unit test is completed and the subprogram `Test_one_unit` is exited.

1 This example assumes that a terminal, rather than an HP 2601A printer is connected to the ASI card in select code 3 of the example system of the "Introduction to I/O" chapter.

ASI BREAK Detect

The kind of serial communications used by the ASI card and most terminals uses one line to transmit data in each direction between terminal and computer. Character data is sent by alternating the state of the transmit line between electrical states known as MARK and SPACE.

A BREAK is defined as the assertion of the transmit line to the MARK state for some period of time. The ASI card detects BREAK if the transmit line (to the ASI) is held in the MARK state for the period of time normally required to send two characters. (You can change this time with a SERIAL CONTROL register.)

Simulating Interrupts - CAUSE INTR

If you want to test your ON INTR service routine, you can use the CAUSE INTR statement. You can only simulate an interrupt with CAUSE INTR for the program in the foreground partition. You cannot simulate an interrupt for a program suspended at a WAIT FOR INTR because the keyboard is ignored (except for STOP).

Multi-programming Considerations

The combination of a successful ENABLE INTR and an ON INTR statement obtains the ownership of the "interrupt resource" of the interface involved. You can execute these statements in either order. If another program (partition) already has the interrupt resource, your attempt to obtain it by executing the second of the two statements causes an ERROR 426. A successful WAIT FOR INTR statement prevents all other I/O access to the interface and does not need to secure the "interrupt resource".

You can prevent programs from destructively competing with one another for I/O resources by using an EVENT convention. The following program prevents other cooperating programs from "stealing" the interface in between lines 330 and 500 by using an EVENT semaphore.

```
300  Interface$=VAL$(Interface)         !Get select code as string
310  ON ERROR GOTO Wait                 !Ignore CREATE fail
320  CREATE EVENT Interface$,1          !Create for immediate release
330 Wait:WAIT FOR EVENT Interface$      !Wait until released
340  RESET Interface                    !Interface now owned
350  DISABLE INTR Interface             !Inhibit interrupts
360  ON INTR Interface CALL Service     !Set up branch
370  ENABLE INTR Interface;Mask         !Enable it
 .
 .
 .
500  CAUSE EVENT Interface$             !Release interface event
```

This example uses the convention of having an EVENT for each interface. The EVENT name is the string numeric equivalent of the interface select code. This technique works only for **cooperating** programs. Any program which does not WAIT FOR EVENT is not denied access to the interface. Each program should attempt to create the EVENT (as this example does) in case the EVENT does not yet exist.

You can also establish a similar convention using memory files. Each interface's file has as its name the string numeric interface select code. Cooperating programs request and release access with the LOCK and UNLOCK statements. This scheme requires more memory but has the advantage that you can list the file (interface) lock status with the CAT statement.

Chapter 20
Mass Storage Organization

Most computers do not have the capability to retain information in memory once their power is shut off. Since most people don't want to leave their machine on constantly, but still have information they would like to keep, a means of saving that information is needed. Mass storage can provide this.

This is useful for those who might be interrupted by another user of the machine. Rather than having to make a person wait until you are finished, and instead of having to reconstruct what you were doing when "bumped" by someone else, mass storage can be employed to save information and permit you to return later after they are done. Another possibility is saving programs or data to be used at a future time - perhaps frequently.

Another inherent problem with every computer is the fact that its memory is not infinite. It is quite possible to exhaust the entire memory of the machine with a program and still have a need for more. In such circumstances, mass storage devices offer the capability of storing large amounts of information in an easily accessible form. A program can then retrieve small amounts of data as required, instead of keeping it all in memory.

In addition, mass storage is a significantly less expensive way to store data than using your computer's memory.

These are the primary reasons for considering mass storage. If they are relevant to you, you have a mass storage application.

Mass storage needs a very structured organization to keep track of the data involved. To do this, a heirarchy of storage units are used. They are devices, volumes, directories, data and program files, defined records, blocks and physical records or sectors.

Understanding the way that the information is stored on media is critical for efficient use. Knowledge of a medium's structure can increase the amount of information that can be stored, increase the speed of data retrieval, and create a more organized file system.

Statement Summary

The following table summarizes the mass storage statements discussed in this chapter.

To perform this operation:	Use this statement:
Define the default mass storage device.	MASS STORAGE IS
Initialize a medium, give it a volume label and specify a directory format for the medium's volume.	INITIALIZE
Read a volume's label.	READ LABEL
Redefine a volume's label.	PRINT LABEL

Unified Mass Storage Concept

The Unified Mass Storage Concept is an approach which enables you, the programmer, to rely upon the device-independence of mass storage statements used in your programs. It is designed so that writing a record to a flexible disc, for example, is in all possible respects the same as writing a record to a hard disc. The concept should enable you to transport applications from one type of device to another, with a minimum of disruption to the program's logic.

Of course, there are still differences between devices which have an impact upon your programming. These are pointed out in the text. The differences between whole classes of devices, such as between flexible discs and hard discs, are also noted.

Particular operating, installation, and maintenance information for HP mass storage peripherals can be found in the operating or installation manual for that device.

Devices

The most obvious part of mass storage is the mass storage device; that is, the physical mechanism which accesses the data.

Selection of the type of mass storage device to use is important to the functioning of your programs. Some applications run better on some types of devices than they do on others. Hard discs are best when frequently accessing large amounts of data (>1Mb) such as a database. Flexible discs are good for small amounts of data which are occasionally accessed. They are also more useful for transporting data or where you need a personal medium. Tape drives work well for backing up the data on other mass storage devices or for transporting data to another location.

Your system has a number of devices available. Consult your HP Sales and Support Office for more information about which device or devices best suit your needs.

The msus

Whenever a statement works with a mass storage device, that statement must tell the system what kind of device it is addressing and where that device is found. This is done with a **mass storage unit specifier** also known as an **msus**. The msus is an item in the statement syntax which clarifies what device is to be operated upon.

The msus consists of a literal describing the device followed by its device selector. This can be followed by optional subunit and volume numbers. The following list shows the possible literals.

If your device is:	**Then type in:**
The system's internal RAM,	MEMORY
The system's internal flexible disc drive,	INTERNAL or CS80
HP 7908P/R, HP 7911P/R, HP 7912P/R, HP 7914P/R, HP 7933H, HP 7935H, HP 97093A,	CS80
HP 9885M/S,	HP9885
HP 9895A,	HP9895
HP 82901M/S,	HP82901 or HP8290X
HP 82902M/S,	HP82902 or HP8290X

The first literal, MEMORY, denotes that part of your computer's internal memory is set aside as a pseudo-mass storage device. Obviously, MEMORY does not have the typical attributes of mass storage, that is, non-volatile recording of data, large recording areas or portability. However, it does have a number of uses. For example, it is faster to use than any physical device. This can be important when you are debugging a program which accesses mass storage constantly.

MEMORY's device selector is 0.

INTERNAL refers to the flexible disc drive inside of your computer.

This msus is provided so that you can write programs which utilize the mass storage devices of other HP 9000 Model 520s or the HP 9000 Series 200 computers without conversions.

What is a Subunit?

Some mass storage devices have more than one **drive**. This is the part of the device that moves the recording medium. For example, the HP 82901M can hold two discs in two separate drives. The subunit number defines which drive on the device to use.

Additionally, the 82901M can control an 82901S or 82902S which has one or two more subunits.

MEMORY can have up to 32 different subunits numbered from 0 through 31. This gives you a way to create 32 separate "recording media" with which to work.

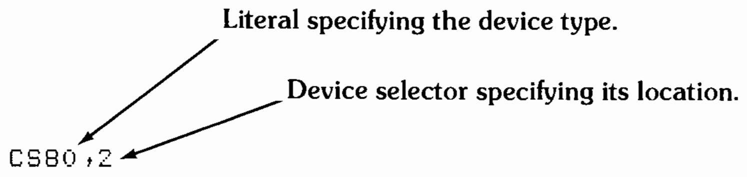
The internal disc drives use subunit numbers when referred to with the CS80 literal. The 97093A internal Winchester disc drive uses 0 and the flexible disc drive uses 1.

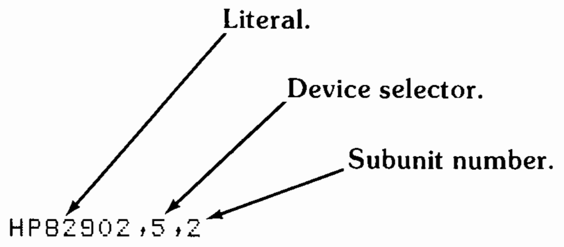
What is a Volume Number?

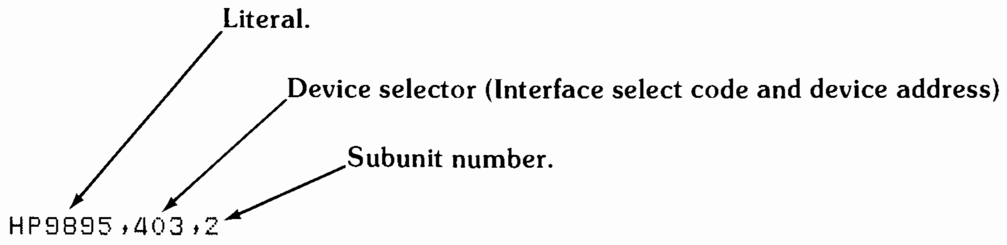
For some mass storage devices, the recording medium can have more than one volume or separate recording area. The volume number defines which volume to use on the medium. This concept becomes clearer as you read further.

Example msus'

To summarize the previous information, the following examples are presented.


Literal specifying the device type.
Device selector specifying its location.
CS80,2





The MEMORY msus' range from:

```
MEMORY,0,0
```

to:

```
MEMORY,0,31
```

Besides the devices listed in the previous table, CS80 can be used with the two internal disc drives. The internal Winchester disc drive uses:

```
CS80,7,0
```

or

```
CS80,7
```

(the 0 is assumed). The internal flexible disc drive uses:

```
CS80,7,1
```

As mentioned before,

```
    INTERNAL
```

also refers to the flexible disc drive.

Most of the examples in the following text use INTERNAL. If you cannot use the flexible disc drive, use MEMORY as your practice mass storage device. Be careful to check any medium that you use to be sure it does not contain valuable data.

Note

Before trying any examples, make sure there is no valuable data recorded on the mass storage media with which you are working.

Media Specifier

When referring to a device in a statement, you do not type in the msus as an unquoted literal. Instead, you use a string expression known as a media specifier. A media specifier must evaluate to the following form:

"**:msus**"

where **msus** is the msus determined above. The media specifier is the string which you actually use in a statement or command. Thus, given a command CAT, both of the following are correct versions of that command.

```
    CAT ":INTERNAL"
    CAT A$  (where A$ = ":INTERNAL")
```

The Default Device

It is inconvenient when working with a single mass storage device to repeat the msus with every statement. The MASS STORAGE IS statement sets a default mass storage device for the system. Thus, the msus can be left out when working with that device.

To do this, merely type in MASS STORAGE IS followed by the media specifier of the device which you want to use as the default device. Some examples of this statement are:

```
    MASS STORAGE IS ":CS80,7,1"
    MASS STORAGE IS ":INTERNAL"
    MASS STORAGE IS "HP9895,1200"
```

When the system is powered-up, the default mass storage device is the internal flexible disc drive. (unless the system's non-volatile memory is modified to specify another device - see the SET_NON_VOL_MEM utility in the Utilities Appendix of this manual). To avoid accidentally accessing another disc drive instead of the flexible disc drive, either always include the msus ':INTERNAL' or execute:

```
    MASS STORAGE IS ":INTERNAL"
```

at the beginning of each session with your system.

The term MASS STORAGE IS can also be typed in using the abbreviated form: MSI. The system takes this literal and interprets it as MASS STORAGE IS. It is quicker to use this statement if you want to switch default devices often. If you type in MSI on a program line, the system automatically converts it into the full form.

Certain statements still require that the media specifier be given. These are noted in the text as they are introduced.

Mass Storage Media

Mass storage media are the actual materials on which data is recorded. At this point it is useful to explain what kinds of media are available and generally how they work.

Your computer uses magnetic discs, tapes and RAM for mass storage. There are a wide variety of discs which are compatible with the system; however, there are two basic types: flexible discs and hard discs. These are used in two kinds of disc drives: fixed and removable.

A flexible disc is a flexible piece of mylar coated with a magnetic oxide material. It combines features of both phonograph records and recording tape. Like a record, it is circular, has tracks and rotates. Some flexible discs record data on one side only and some on both sides. Like a tape, data is read and written onto a disc by an electromagnetic head.

The disc drive mounted above the keyboard of your computer uses a 5-¼ inch diameter flexible disc. These discs consist of tracks or rings of recording area on both sides. You can't access all of it, however, since some is reserved for system use.

Hard discs look similar to phonograph records. They are flat, aluminum platters. These discs work on the same principal as the flexible discs, but they store much more information. You may have a hard disc known as a Winchester inside your computer.

Flexible discs are usually associated with removable-media disc drives. That is, the disc drives are designed so that media can be interchanged in the drive. Some hard discs are used in removable-media disc drives. Fixed-disc drives have one dedicated medium which cannot be replaced. The Winchester is a fixed-disc drive.

The hard discs may also have a backup tape which may be in the same physical cabinet as the hard disc or in a separate cabinet. The tape is used for storing all of the hard disc's information quickly on a removable medium. This tape can then be stored in some safe place so that if valuable data is lost on the disc, it can be retrieved from the tape.

Volumes

A **volume** is a complete and separate unit of mass storage which represents an independent file system. It is often a unique mountable medium such as a flexible disc. Some hard discs may have more than one volume. You can think of it as the largest organizational unit of mass storage on a medium.

Every disc that you use must have a volume (some may have more than one). To create this volume you must "format" or "initialize" the disc.

Initializing a Medium

Initialization is the process of preparing a mass-storage medium for data storage. This consists of the following steps.

1. Sector patterns (pre-amble, data area, post-amble) are written to every sector of the medium. Some media are factory-formatted and do not require this step.
2. A defined record number is written to each sector pre-amble. If the interleave factor is not 1, the defined number is not the same as the sector's physical location.
3. A test data pattern is written to the data area of each sector.
4. The test patterns are validated, and if a sector contains bad data, the sector is marked defective. Defective sectors are not used for data storage. A spare sector may be substituted, or the entire track may be marked defective and be replaced by a spare track or made inaccessible by re-numbering subsequent tracks.
5. The volume header is created and the specified or defaulted volume label, block size and disc format are written. A root directory of the specified or defaulted format and size is written. If necessary, a file attributes file is created.

With a flexible disc, if too many bad sectors or tracks are found, the initialize operation is terminated and ERROR 66 is reported.

A mass storage medium can be provided by a number of vendors. A list of approved media manufacturers is available through your HP Sales and Support Office. It is important to make sure that the discs you use are compatible with HP equipment. The recommended flexible disc to use for an internal flexible disc drive is HP Product Number 92190A (this is a package of 10).

CAUTION

DISC DRIVE PERFORMANCE AND RELIABILITY ARE DEPENDENT UPON THE TYPE OF MEDIA USED. DISC DRIVE SPECIFICATIONS CAN BE ASSURED ONLY WHEN USING HP MEDIA. THE USE OF IMPROPER MEDIA CAN RESULT IN PREMATURE DISC FAILURE OR DAMAGE TO THE DISC DRIVE.

ON SOME DISC PRODUCTS, HP MAY QUALIFY OTHER NON-HP MEDIA. WHEN TESTED, THIS MEDIA MET HP SPECIFICATIONS; HOWEVER, HP DOES NOT WARRANT OR SUPPORT THIS MEDIA AND CANNOT CONTROL CHANGES IN ITS SPECIFICATIONS OR QUALITY. THE SELECTION AND USE OF SUCH PRODUCTS IS THE CUSTOMER'S RESPONSIBILITY. HP RESERVES THE RIGHT TO EXCLUDE FROM THE WARRANTY AND MAINTENANCE AGREEMENT COVERAGE ANY REPAIRS WHICH HP REASONABLY DETERMINES OR BELIEVES WERE CAUSED BY THE USE OF MEDIA NOT PROVIDED BY HP. HP PROVIDES SUCH REPAIRS UPON REQUEST AND ON A TIME AND MATERIALS BASIS.

WARRANTY AND MAINTENANCE AGREEMENT COVERAGE OF REPAIRS NOT CAUSED BY THE USE OF NON-HP MEDIA IS UNAFFECTED.

Make sure that the disc does not contain any important data or programs. When a disc is initialized, all the data on it is lost.

Check that the disc is not "write protected". For the flexible disc used with the internal disc drive, the envelope has a small, square notch on one side. When this notch is open, the computer is allowed to write on the disc. If this notch is covered, data may be read from the disc, but recording is not allowed. (eight inch flexible disc drives use the opposite convention). Trying to initialize a "write-protected" disc results in ERROR 83.

Hard disc media may be write-protected with a switch on the cabinet. If you cannot write to a hard disc, refer to the disc drive manuals.

To initialize a medium, the disc must be in its drive and the drive must be connected to the computer. The details on how a device is connected, how media are inserted and other items are found in the operating manual for the device itself or in the "Getting Acquainted with BASIC" chapter of this manual.

Initialization of a hard disc can take a long time. In particular, the internal Winchester initializes in 20 to 40 minutes depending on the interleave factor chosen. Watch the run light in the lower-right corner of the internal CRT to determine when the INITIALIZE statement has completed. If the light is being displayed, the intialization is still continuing; if the corner is blank, the initialization is over.

Note

Do not turn off your machine while it is initializing a disc. This will corrupt that disc. If you turn off your machine while initializing the internal Winchester, it may be recovered using the WINCHESTER system integrity test explained in your computer's Installation and Test manual. If you corrupt a CS80 disc, contact your HP Sales and Support Office for help.

In this example, you initialize a flexible disc in the internal disc drive. Get a blank disc and place it in the internal disc drive.

Now execute the command:

```
    INITIALIZE ":INTERNAL"
```

Wait for the run light to go out, indicating that the initialization process is complete. The processing time required is determined by the fact that every physical record on every track on the medium (tracks only for hard discs) is accessed and checked.

Interleaving

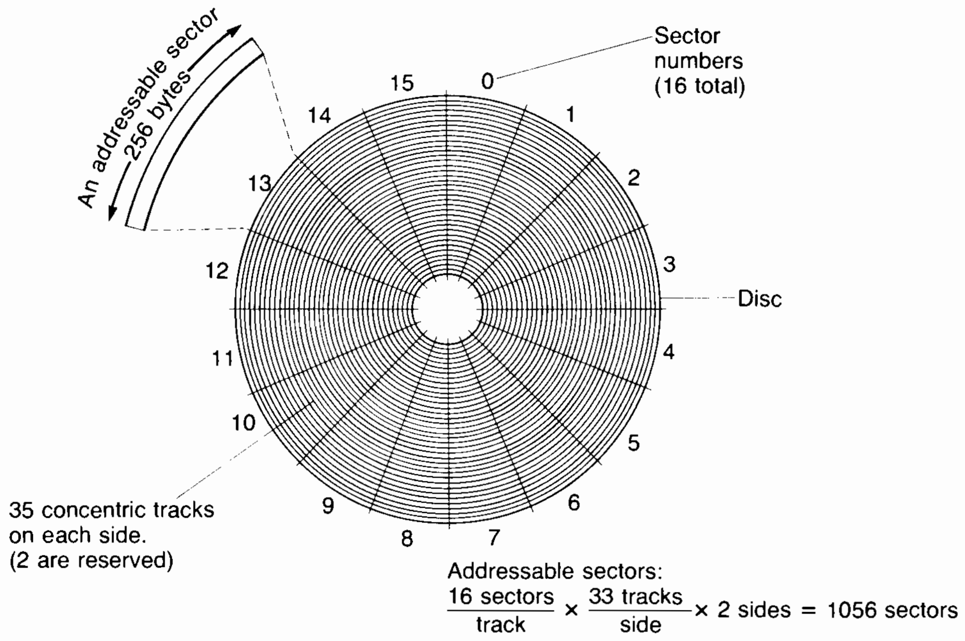
Interleaving is a way to increase the speed of reading and writing to a flexible disc. This is done with an optional parameter in the INITIALIZE statement.

The purpose of this factor is to allow you to control the I/O efficiency of your disc. The interleave factor is a numeric expression which is rounded to an integer. It must be in the range 1 thru 255. If you omit it when initializing a flexible disc, it defaults to 1.

Interleaving is a process whereby the system effectively renumbers sectors on a track so they are no longer consecutive. The sectors may remain consecutively numbered or numbered by skipping every other sector, or by skipping every third one, and so on. Because it takes a finite amount of time to read a sector, to transfer data to your computer, and to prepare for the next sector, and because the disc is spinning during that time, it is possible for the next sector to skim right past the read/write head. It then requires another full revolution of the disc to access two successive sectors on the same track. Interleaving works to decrease that effect.

To speed up this process and enable successive records to be read on the same revolution, the interleave factor causes the numbering of records to be altered so that there is physical separation between them, enabling a minimum number of revolutions to be sufficient to read all the records on a track. This can result in access speed improvements of up to a five-times increase.

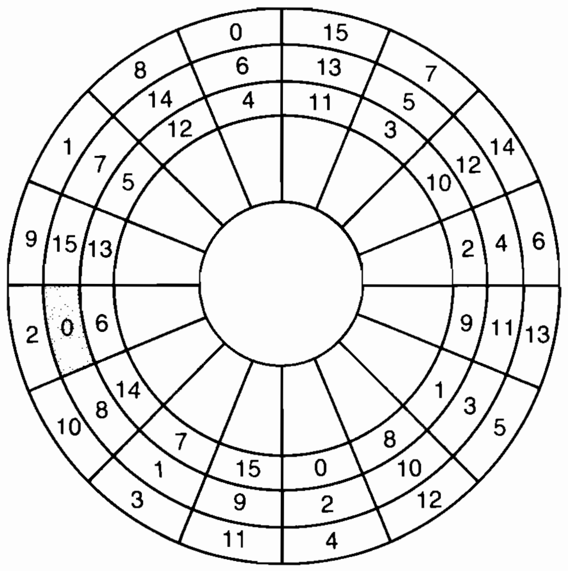
Each disc used in the internal flexible disc drive has 35 circular tracks on each side, numbered 0 thru 34. Each disc is also subdivided into 16 pie-shaped sectors. Each sector contains one physical record from each of the tracks.



An addressable sector
256 bytes
Sector numbers (16 total)
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
Disc
35 concentric tracks on each side. (2 are reserved)
Addressable sectors:

$$\frac{16\text{ sectors}}{\text{track}} \times \frac{33\text{ tracks}}{\text{side}} \times 2\text{ sides} = 1056\text{ sectors}$$

A diagram of disc tracks and records with an alternating numbering system caused by an interleaving of 2 is shown next. The shaded area shows the location of record 0, track 1, as an example.



Tracks of a disc with Interleave Factor of 2

In addition to an alternating numbering system, the location of the beginning sector (sector 0) of each track is skewed to avoid a revolution when the drive accesses a new track. For example, after sector 15, track 0 is accessed, then sector 0, track 1 is accessed without an extra revolution.

It is recommended that the internal Winchester disc drive be initialized with an interleave factor of four; the default interleave factor of one degrades performance.

Initializing MEMORY

Initializing the system's internal memory to act as a mass storage device is somewhat different than initializing physical mass storage media.

The system's internal memory does not have a physical device which must move mechanically to access data. Thus, the concept of an interleave factor is not applicable. However, you can specify the amount of memory to be included in the mass storage. Thus when you initialize MEMORY, the optional parameter that normally specifies the interleave factor is interpreted as the number of 256 byte records in the MEMORY subunit. For example,

```
INITIALIZE ":MEMORY,0,12",12
```

means that MEMORY subunit 12 is to be initialized and is to contain ten records of 256 bytes each.

Volume Labels

It is possible to give a volume a label when initializing a disc. There are three major reasons for doing this.

- It insures that the access fails if the wrong volume is mounted at the specified address.
- Using the volume label in a program instead of the device's msus makes the program more readable and easily updated.
- It can help you remember what kind of files are stored in that volume.

Volume labels consist of characters excluding the slash (/), less-than (<), colon (:) and semi-colon (;). A single period (.) or double period (..) cannot be used as a name. Leading and embedded blanks are deleted.

Try initializing the flexible disc again with this media specifier.

```
INITIALIZE ":INTERNAL;LABEL LEARNING_DISC"
```

Now the disc's volume has the label LEARNING_DISC. Note that this does not eliminate the name INTERNAL. INTERNAL refers to the drive while LEARNING_DISC refers to the volume. LEARNING_DISC will help you remember that this is the disc which you used to learn mass storage techniques.

To examine what the volume's label is, use the READ LABEL statement. Execute:

```
READ LABEL A$ FROM ":INTERNAL"
DISP A$
```

The string variable A$ now contains the volume's name.

Once a disc has been initialized, you may want to change the volume label. PRINT LABEL changes the label of a volume without affecting the volume's contents. Try executing the command:

```
    PRINT LABEL "NEW_DISC" TO ":INTERNAL;LABEL LEARNING_DISC"
```

Now do a READ LABEL and see what the volume's label is.

Including a device type (such as INTERNAL) as well as its label has the advantage of providing a check of the device to be operated on. If the device type and the label don't refer to the same medium, an error occurs. For example, if you have a pair of HP 9895s, you may want to include the volume label and device type whenever referring to them so that no accidents occur.

Files

Every volume consists of a set of files. These files are the next level of data organization on the medium. A file is created to hold information which you want the system to manipulate as a unit.

The first type of file discussed is a directory. A directory (or DIR file) contains all the bookkeeping information about a group of files: their names, types, lengths, starting locations, and other information. In order to access a file, you must also access its directory. A file created using the BASIC language can only exist in one directory at a time.

A directory is like a mailbox. You can open a mailbox and see what kinds of letters are inside. You can gain some information, such as who the letters are from or generally what they are about, but you can't read all the data in the letters unless you open the letters themselves. You can tell a lot about a file by examining the directory in which it's located, but you have to access the file itself to use its data.

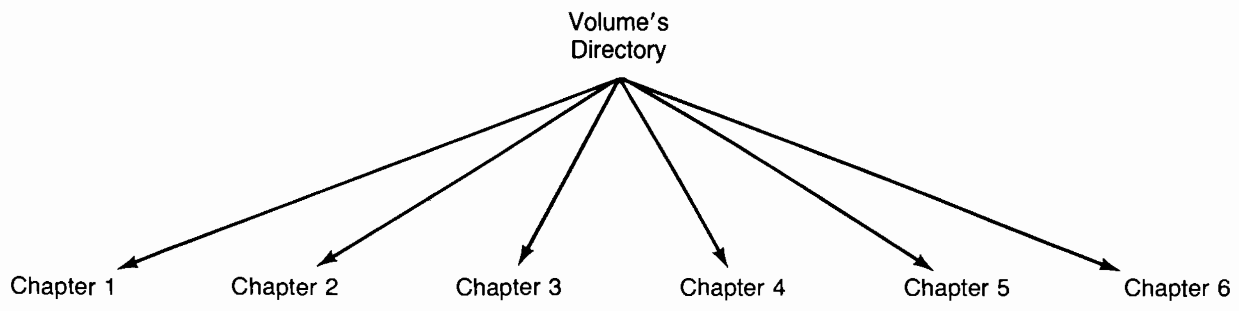
Directories and other types of files are explained more fully later in the text.

Directory Formats

The directory format is the way in which the files are organized in the volume. You have three directory formats from which to choose: the Structured Directory Format (SDF), the Logical Interchange Format (LIF), and the 9835/45 format (9845). Each one has a specific use.

Logical Interchange Format (LIF)

The Logical Interchange Format is somewhat different from SDF. There is only one directory on the medium and all data and program files on the medium are listed in that directory.



No two files on the medium can have the same name since it would be impossible to differentiate files. The directory itself does not need a name since it is the only one in the volume. It is simple to use and requires very little knowledge of the volume's organization to find a file. This format is useful when you have a limited number of very similar files. An example might be a volume composed entirely of test data; this case doesn't require multiple levels of directories and divided files. It might be simpler to have a single level of files with descrptive file names to distinguish the different versions or sets of data.

LIF Internal Structure
The Logical Interchange Format has three parts: the volume header, the directory and a data area for files.

The volume header consists of a single 256 byte block at the beginning of the volume (block zero). It contains information about the identity of the directory format, the volume's label, the starting address of the directory and the length of the directory.

The directory is a linear list of entries, one entry per file, each of which contains the following information: the file's name, its file type, its starting address, the length of the file and the time of creation.

The number of files that can be recorded in the directory is dependent upon the size of the volume. The number of entries possble is equal to one tenth of the number of physical records on the disc. If you know the size of your disc in physical records and the fact that each file entry is 32 bytes, you can determine a rough estimate of the size of the volume's directory.

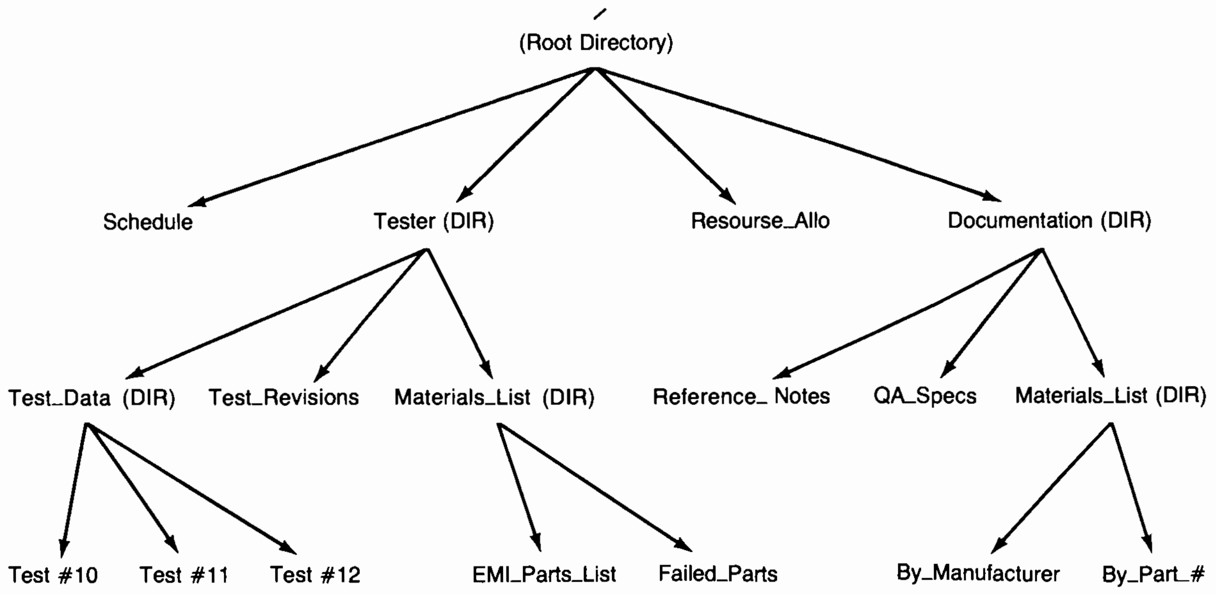
The first file starts at the beginning of the next block after the directory. The next file starts at the block following the end of the previous file and so on until all the files in the directory are recorded. If the volume is filled, no more directory entries are allowed and only as many files as can be listed in the directory are allowed.

Summary
The Logical Interchange Format is a standard format used by HP computers and should be used whenever you want to transport a particular medium to another type of HP computer. Note that only the ASCII file type is defined by all HP systems which support LIF.

9845 Directory Format
The 9845 Directory Format is very similar in structure to the LIF format; it is single directory containing all the files. The 9845 directory format is provided for program and data interchange with the HP 9835/45 and the HP 250.

Structured Directory Format (SDF)

The Structured Format is organized like a tree. There is a root directory from which a set of files can be accessed. It has no name or label. One or more additional directories can be in this set. They are known as subdirectories. Each one of these subdirectories, in turn, leads to another set of files which may include further subdirectories. The number of levels of directories possible with this format is limited only by the capacity of the mass storage medium.



The first level of the directory format always contains the root directory, created during INITIALIZE. The root directory points to the files `Schedule`, `Tester`, `Documentation`, and `Resource_Allo`. `Schedule` and `Resource_Allo` are files containing information which you need to retain. These files can only be accessed through the root directory. `Tester` and `Documentation` are directories within the root directory and are known as **subdirectories** of the root. The root directory is known as their **parent directory**. Each of these subdirectories points to other files at the next level. `Tester` points to `Test_data`, `Test_Revisions`, and `Materials_List`. `Documentation` points to `Reference_Notes`, `QA_Specs`, and `Materials_List`.

Note that this third level of files contains files with names which are exactly the same as other file names in the tree. This is permissable as long as the files with the same names are not in the same directory. Having a file called `Materials_List` in the `Tester` directory is acceptable because the other `Materials_List` is in `Documentation`.

Test_Revisions, Reference_NotesQA_Specs contain data or programs. Test_data and both Materials_Lists are directories pointing to the fourth level of the file tree. Note that you can have two or more subdirectories in a single directory.

The fourth level of the directory format contains various data and program files. You could, of course, continue to expand the file tree by creating a directory at this level. Again, you are only limited by the size of the medium.

In addition to expanding the file tree, you can expand individual files as necessary. Thus, a file may be created with an initial size X and later expanded, as more data is placed in it, to 2X or 3X. For all files except BDAT, the expansion is only limited by the size of the medium. For BDAT files, you specify the maximum size that the file can have.

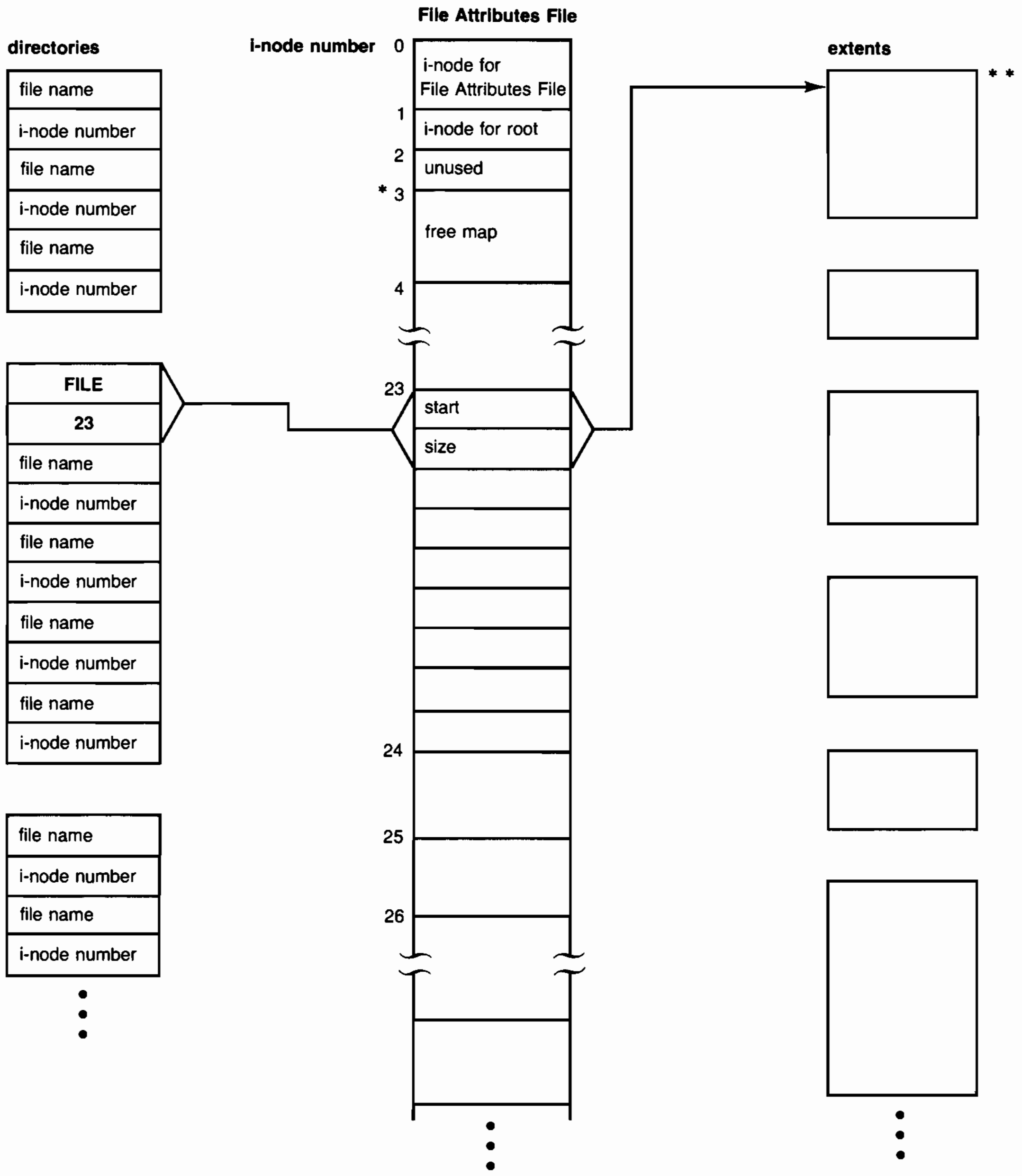
SDF Internal Structure

The Structured Directory Format has four parts: a volume header, an optional reserved boot area, a file attributes file and the data area for files.

The volume header is a single block of space located at the beginning of the volume. It contains information about the identity of the directory format, whether the volume is corrupted, the volume's name, the date and time that the volume was initialized, the block size, the start of the optional boot area, the boot area size, the start of the file attributes file, the largest block that can be addressed in the volume, the volume password and the last date that the volume was backed up.

The optional boot area is a part of the disc where you can place one or more copies of your operating system. The boot area is considered separate from the file system and is not shown in a CAT listing. For more information about the boot area, see the "System Configuration" appendix.

The SDF file system implementation is a concept best explained by example. Suppose that you have a file named FILE which contains a list of mailing addresses. When FILE is created the system places its name in a directory. It then associates a unique integer value with the file name. This integer value (called an i-node value or number) points to a structure called an **i-node**. In turn, the i-node contains a pointer to one or more groups of contiguous disc blocks, called **extents** in which FILE's data is actually stored. A pointer in the i-node consists of the starting address of the file's extent(s) and the size of the extent(s). This is shown in the illustration that follows. All i-nodes for a file system are kept in a file called the **File Attributes File**. Each disc has a single File Attributes File which describes the contents of the disc.



* The free map may actually occupy more than one i-node.

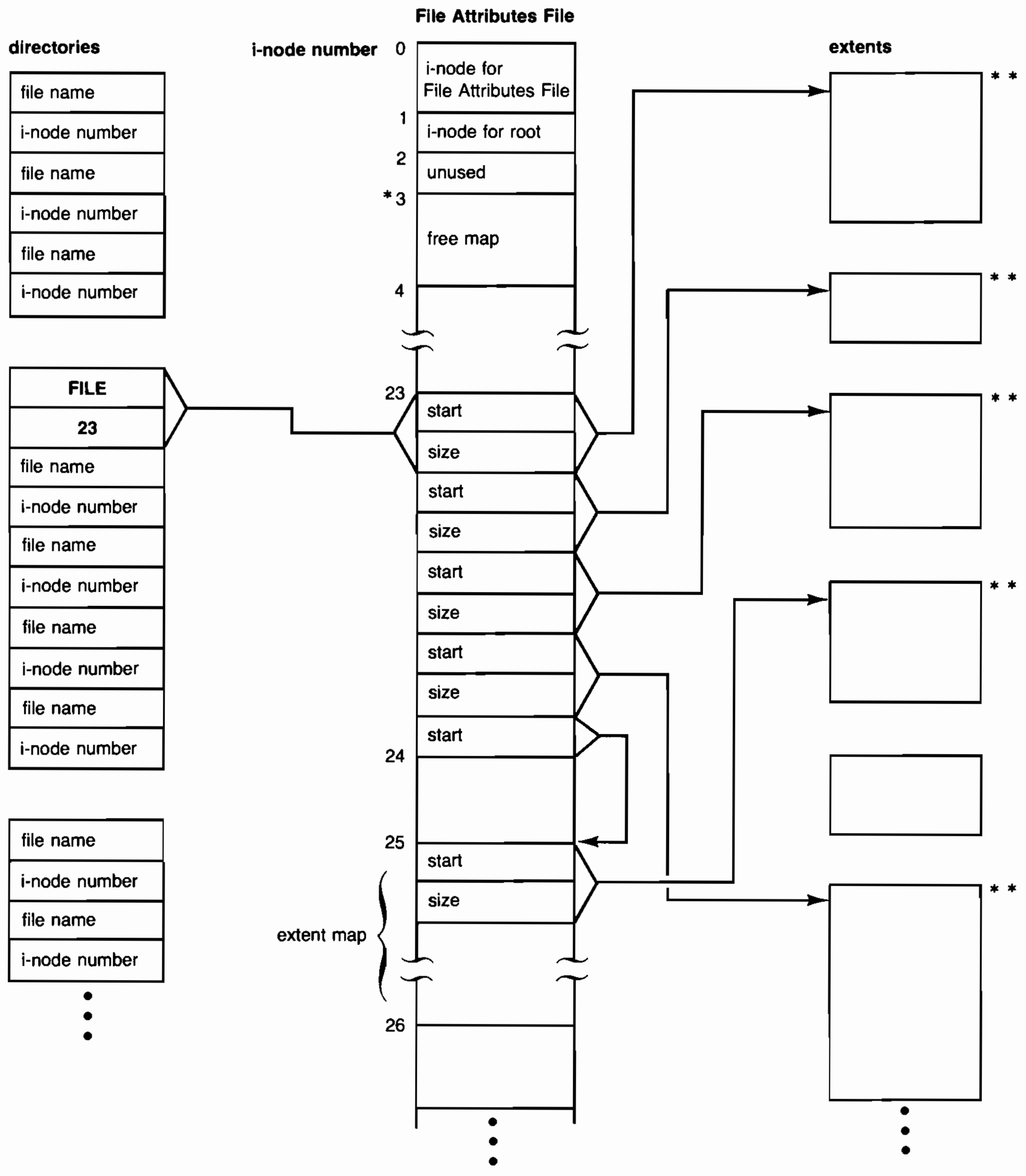
* * Actual contents of FILE.

File System Implementation
File with a Single Extent

From the diagram, you can see that besides an i-node for each file stored on the disc, the File Attributes File contains an area called the **free map**. The free map is an array of binary values, one for each block on the disc. If a bit value is 0, the corresponding disc block is currently being used for storage. If a bit value is 1, then the corresponding disc block is available for storage.

When a file is created, its contents are placed in one large extent, if possible. If the file is later modified and enlarged, the system attempts to place the added information in blocks that are contiguous with the existing extent. If there are not enough free blocks that are contiguous to the existing extent, the system places the added information in a single new extent (if possible); it then adds to the file's i-node, a pointer to the additional extent. The i-node is capable of holding four pointers to the extents which comprise the file. If the system finds it necessary to create more than four extents for a file, an extent map is automatically added (chained) to the file's i-node. An extent map is capable of holding an additional 13 pointers; there is no limit to the number of extent maps whicn may be added to an i-node. You should note, however, that as the number of extents claimed by a file increases, the amount of time required to access the information also increases.

The following diagram shows the structure of the file, FILE (from the previous example) after it has been modified several times. When FILE was modified, there were not enough free blocks contiguous with the existing extent in which to place the added information. Thus, additional extents were created to hold the new information. When the number of extents claimed by FILE surpassed four, the system added an extent map to FILE's i-node.



* The free map may actually occupy more than one i-node.

* * Actual contents of FILE.

File System Implementation
File with Multiple Events

Summary

The Structured Directory Format provides flexibility in organization and use of data. You can create distinct categories for various programs. You might have a separate directory and file set for your text editing tools. You might want a separate directory and file set for various test data. By the judicious organization and naming of files, you can make it very easy to keep track of all your programs and data. Also, by proper use of passwords (discussed later) you can set up a more usable environment for multiple users.

Specifying a Directory Format

The directory format is determined at the time of initialization. You may choose either SDF, LIF or 9845 using a certain phrase in your msus. Here are some typical initialization commands specifying the structure.

```
INITIALIZE ":INTERNAL; DISC FORMAT SDF"
INITIALIZE ":82901, 10; LABEL TEST, DISC FORMAT LIF"
INITIALIZE ":CS80, 12; DISC FORMAT 9845, LABEL EXP12"
```

If you don't specify a directory format when initializing a medium, the default format is SDF.

File Specifiers

A media specifier tells the system which device or volume the operation affects. The file specifier tells the system which file the command affects. The general form of a file specifier - an expanded form of the media specifier - is:

"**file name : msus**"

The colon and msus are optional. If they are not included, the filename refers to a file on the current MASS STORAGE IS device.

With the LIF or 9845 directory formats, file name is merely the name of the file which you use. For example, each of the following are valid file specifiers.

```
"Testdata:INTERNAL"
"CHECK:LABEL REPORT"
"Notes_2:CS80,11"
```

With the SDF directory format there is a wider range of possibilities. This format is structured so that you can specify a particular file in another part of the tree regardless of whether you are above it or below it. You may specify a file by an absolute file specifier or by a relative file specifier. The type of file specifier you use depends upon your location and the location of the particular file you want to use.

Absolute File Specifier

An absolute file specifier always begins with the root directory and follows the branches from file to file until it reaches the desired file. For example, in the file tree shown earlier, to find the file `Test#12` in the directory `Test_data`, you start your search in the root directory, proceed to `Tester`, then to `Test_data`, and finally arrive at `Test#12`. Putting this all together, the absolute file specifier looks like this:

```
"/Tester/Test_data/Test#12"
```

Note that the beginning slash always represents the root directory. All other slashes separate one file name from the next. You can verify for yourself that

```
"/Documentation/Materials_List/By_Part_#"
```

refers to the file `By_Part_#` in the directory `Materials_List`.

```
"/Tester/Materials_List/EMI_Parts_List"
```

refers to the file `EMI_Parts_List` in the directory `Materials_List`, which in turn is in the directory `Tester`.

Relative File Specifier
A relative file specifier never begins with a slash. This is because the system begins the search for the specified file in the directory in which you are currently working, known as your **working directory**. Assume that your working directory is `Tester`. To specify the file `Materials_List`, you could use an absolute file specifier `"/Tester/Materials_List"`, but all you really need to say is:

```
"Materials_List"
```

Since the file name does not begin with a slash, the system assumes it is a relative file name and begins the search in your working directory. To specify the file `EMI_Parts_List` the correct file specifier is:

```
"Materials_List/EMI_Parts_List"
```

The system finds no initial slash and begins the search in your working directory (once again, the slash in the file specifier serves only to separate files). It locates the directory `Materials_List` and then finds the file `EMI_Parts_List`.

You might think that you cannot use a relative file name if the desired file lies outside of your working directory. The SDF format provides a shorthand notation that enables you to do this. The system maintains two symbols, dot (.) and double-dot (..), which always refer to your working directory and its parent directory, respectively. Assume once again that your working directory is `Test_data`. If you want to specify the file `EMI_Parts_List` in the directory `Materials_List`, the correct relative file specifier would look like this:

```
"../Materials_List/EMI_Parts_List"
```

The file specifier contains no initial slash, so the system begins its search in your working directory. The double-dot refers to `Test_data`'s parent directory, `Tester`. The system finds `Tester`, looks for `Materials_List` in `Tester`, and finds `EMI_Parts_List` in `Materials_List`.

A relative file specifier is called "relative" because the search is carried out relative to your working directory. Note that a correct relative file specifier in one working directory would not necessarily be correct in another.

Specifying the Working Directory

To change your working directory, use the MASS STORAGE IS statement. All you do is specify a particular directory in the file specifier of the statement. For instance,

```
MASS STORAGE IS "/Tester/Materials_List:INTERNAL"
```

sets the working directory to Materials_List.

If you try to specify a file containing a program or data as the working directory, an error occurs.

Records, Blocks and Sectors

Every file consists of smaller units of information known as records, blocks and sectors. These units greatly affect how efficiently you can store and used data on the medium.

Logical Records

Logical records are a structured data concept and have no direct implementation in either the hardware or software of your computer. These are records in which you conceive the data to be organized. In short, it is a collection of individual data items which are conceptually grouped. For example, if you had a personnel application, you might, in the course of programming it, plan a logical record for each employee which contains the employee's number, name, address and department. You might want to deal with this record (like reading it from a data base) as a single record, at least logically. To assist in this application, you might want to make your defined records (explained next) large enough to hold all the data in a single logical record. Of course, you could assign the individual components of the logical record to various defined records and keep track of where things are stored.

Defined Records

Defined records are the smallest units of storage which you can randomly access individually. You can set the actual length of this record, in bytes, when you create a data file. Defined records are always an even number of bytes in size. If you select an odd number for the record size, it rounds up to the next even number. All defined records in a file are the same size. The defined-record size defaults to a physical record.

Sometimes you need to know if a certain size of defined record can hold all your data. In calculating the length of a part of a record with combinations of numeric variables and strings, it is helpful to know how much space each type of item takes up. The "File Access and Data Transfer" chapter explains how to calculate the record length needed for your data.

Block Sizes

When your system deals with mass storage devices, it works with **blocks**. When a medium is intialized, the system divides it into blocks of space. The actual size of the blocks is dependent upon the directory format chosen (SDF, LIF or 9845) and the size of the medium. If the directory format is LIF or 9845, the block size is 256 bytes. If the directory format is SDF and the medium has less than one megabyte of space, the block size is 512 bytes. If the directory format is SDF and the medium has more than one megabyte of space, the block size is 1024 bytes.

These block sizes are chosen for performance reasons. It is important to be aware of them because they affect how much information can be stored on a single medium. Every file is allotted a minimum of one block when it is created although this may not appear in any listing of files. When files are extended, they grow by the block. By carefully watching how much of each block you are using, you can use your medium most efficiently.

Physical Records

A physical record is the unit of storage dealt with by the mass storage devices themselves. You do not address physical records as such; the bookkeeping and manipulations involving these storage units are handled by the hardware and the operating system.

You should, whenever possible, to set the defined-record size to a multiple or divisor of the physical record length in order to achieve maximum efficiency. If there's no specific reason for establishing a different length, selecting the physical record length over another can result in I/O performance improvements of as much as 2:1 or 3:1.

Sectors

A sector is a specific type of physical record used when dealing with discs. Every track on a disc is divided into sectors. On the 5-¼ inch flexible discs, each track is divided into 16 sectors of 256 bytes. On other discs the number and size of sectors may be different.

All the information recorded onto a disc is organized by sector. Data items are placed in a sector until nothing more fits and then the next sector is used. Not all items are the same "size" so some data fills the sector more completely than other data. It is important to be aware of the sector size and the size of data you want to place in that sector. By proper management, you can save yourself a significant amount of mass storage space.

Chapter 21
Working with Files

Once a medium has been initialized and the file structure has been selected, you are ready to begin manipulating data. The discussion is divided into the following topics:

- file types;
- creating files;
- listing directories;
- naming and renaming files;
- saving and storing files;
- getting and loading files;
- re-saving and re-storing files;
- moving files;
- copying files;
- purging files;
- protecting files.

Statement Summary

The following table summarizes the mass storage statements discussed in this chapter.

To perform this operation:	Use this statement:
Create an ASCII file	CREATE ASCII
Create a BCD file	CREATE BCD
Create a BDAT file	CREATE BDAT
Create a DATA file	CREATE DATA
Create a DIR file	CREATE DIR
Create a PROG file	CREATE PROG
List the contents of a directory	CAT
Rename a file or move it in a volume	RENAME
Store a program in a DATA file	SAVE
Store a program in an ASCII file	SAVE ASCII
Store a program in a PROG file in intermediate form	STORE
Store a program in a PROG file in compiled form	COMPILE
Store all loaded BIN options	STORE BIN
Store special function key definitions	STORE KEY
Store a subprogram in a PROG file	STORESUB
Load a program from a data file	GET
Load a program or subprogram from a PROG file	LOAD
Load a BASIC Language option	LOAD BIN
Load special function key definitions	LOAD KEY
Load a subprogram	LOADSUB
Re-store a program in a data file	RE-SAVE
Re-store a program in a PROG file	RE-STORE
Re-store special function key definitions	RE-STORE KEY
Copy a file	COPY
Purge a file	PURGE
Protect a file	PROTECT

File Types

A file can be any one of seven types: DIR, BIN, PROG, ASCII, BCD, BDAT and DATA files. Each has its particular traits and is designed for a unique purpose.

DIR Files

A DIR file is a subdirectory in a Structured Directory formatted volume. It is used to organize other files on a mass storage medium. Each directory contains entries for its own unique files. The directory's information includes each file's name, type, size, location and protection. All DIR files draw from the same pool of available sectors when you create a file on a volume in the directory.

BIN Files

BIN files are files containing run-only programs written in the machine's internal code which add enhancements to the existing operating system. BASIC Language System options are contained in a series of BIN files which you bring into your system's memory to add language enhancements. For example, the Graphics option "GRAPHICS" and the graphics CRT driver "HP98770" are both BIN files.

PROG Files

A PROG file stores a program and its subprograms in one of two forms: tokenized or compiled form. The tokenized form is an intermediate code between the way you typed in the program and the program's compiled form. PROG files are used when you want to load and run a program quickly and efficiently. Programs can be stored as data in data files but they are recorded in their "source" form (the way you typed it in). This means it takes slightly longer to load and run the program because it must be compiled first. PROG files are also a more compact way to store programs.

ASCII Files

ASCII files are one of the four types of data files supported by Series 500 BASIC. They are generally used to store text or to transfer data to other Hewlett-Packard products.

BCD Files

BCD files are another of the four data file types available with Series 500 BASIC. It is provided primarily for data interchange with HP 9835/45 Desktop Computers. Its format is exactly the same as an HP 9835/45 DATA file.

BDAT Files

BDAT files are another type of data file supported by Series 500 BASIC. BDAT files can contain data in either ASCII format (FORMAT ON) or the machine's internal format (FORMAT OFF). BDAT files use less space to store the same amount of data as ASCII, BCD or DATA files. Since the data can be in the same format as the machine's internal data storage, data transfer between a BDAT file and the machine is more rapid than with an ASCII, BCD or DATA file.

BDAT files store much more data on a disc because there is less storage overhead (numeric data items do not have a header that describes the type and length of the item). You should use this file type whenever you need fast data access. Because numeric items do not have a header, extra care must be taken when reading from or writing to BDAT files.

BDAT files are compatible with HP 9826/36 BDAT files. HP-UX files are also identified as BDAT files when used with the BASIC system. That is, if you create files with an HP-UX system, they appear as BDAT files in BASIC.

DATA Files

DATA files are the last of the four data files available with Series 500 BASIC. DATA files are good for general data storage. Although a little bulkier and slower than BDAT files, they do not require any particular care during the reading or writing of information.

Summary

BIN files provide the code for the operating system and language (that is, gets the system running), DIR files organize the volume, PROG files contain programs, subprograms and functions that you wish to run and ASCII, BCD, BDAT and DATA files store data to be manipulated.

Creating Files

You must create a file before it can contain any data. There are a number of statements which explicitly create each type of file. They are:

CREATE ASCII
CREATE BCD
CREATE BDAT
CREATE DATA
CREATE DIR
CREATE PROG
DBCREATE (discussed in the IMAGE/Data Base Programming Techniques manual)

Creating a file causes the following. First, the file is entered in the directory. All associated information – name, defined-record size, number of records, physical-track, and physical-record location of the first record, file type, and protect code – are all stored in the directory. Every file starts at the beginning of a physical record, so there may be some wasted space between files.

For ASCII and BDAT files, the EOF mark is placed at the beginning of the file. For ASCII files this means the entire file's data is unusable.

Second for BCD and DATA files, the system writes an End Of File mark (EOF) at the beginning of every defined record in the file. An EOF flags where the end of the file is so that the system won't read meaningless data beyond the end of the the last record. Later on, when you are writing data to the file, you write over some of these marks (if not all). By having initially written an EOF mark to every record, you are guaranteed of having an end-of-file mark somewhere in the file, regardless of whether you write to it in serial or random fashion. In addition, this procedure effectively clears each defined record so that no old data remains from any previous use of the record.

EOF marks are irrelevant in BIN, DIR and PROG files because you don't read or write data to them like you do the data files.

As noted before, the system creates a directory in the volume at the time of initialization known as the root directory. To place an ASCII file in the root directory, type in the command:

```
    CREATE ASCII "TEST1:INTERNAL",4
```

The system creates an ASCII file called `TEST1` in the root directory. The number 4 specifies the initial length of the file in physical records or sectors.

You can also incorporate a variable which returns the number of any error that occurs during the execution of this statement. An example is:

```
    CREATE ASCII "TEST1:INTERNAL",4;RETURN Error
```

Normal error processing is suspended and the variable Error is given the value of the first error the system finds during the execution of the statement. During a program, this helps you to recover from an error and take an alternate action to correct the situation.

Creating other types of files is very similar. The BASIC Language Reference shows the syntax of each CREATE statement.

Listing Directories

You can find out what files are in a directory by using the CAT command. Check to make sure that `TEST1` was created. Type in:

```
    CAT ":INTERNAL"
```

The following table should appear on the internal CRT.

```
:INTERNAL
LABEL:
FORMAT: SDF
AVAILABLE SPACE: 1021
                                    SYS  FILE   NUMBER    RECORD     MODIFIED      PUB OPEN
FILE NAME                       LEV TYPE  TYPE   RECORDS   LENGTH DATE           TIME ACC STAT
=============================== === ==== ===== ======== ======== =============== === ====
TEST1                             1      ASCII        4      256 11-Nov-82  17:13 MRW
```

The first seven lines of the table are known as the "header".

The mass storage device is listed in the upper-left corner. In this case, it's the internal disc drive. LABEL shows the volume's name. FORMAT indicates the file structure. AVAILABLE SPACE shows how much empty space in sectors is left on the medium.

FILE NAME shows the names of the individual files. LEV shows the subdirectory level at which the file resides relative to the directory which was specified. In this case, `FILE1` is found within the root directory so its level is 1.

SYS TYPE is the type of system with which the file is compatible. It is blank in this example because ASCII files are compatible with many Hewlett-Packard computers. FILE TYPE is the file's type. Notice that `TEST1` is an ASCII file as specified. NUMBER RECORDS is the number of defined records in the file. `TEST1` should have 4 records. RECORD LENGTH is the number of bytes per defined record. For an ASCII file this is always 256. MODIFIED DATE TIME is the last date and time the file was modified. This indicates the last time the data was changed in the file.

Other Output Devices

The default output device for a CAT is the current PRINTER IS device. You may also specify another device by supplying a device selector in the statement. For example,

```
CAT TO 406
```

lists the contents of the current directory on the device at the device selector "406". This might be an HP 2631B printer so that you can get a permanent copy.

Formatting Options

A number of options are available with the CAT statement to tailor the convenient listing format to your needs. For example,

```
CAT ":INTERNAL";NO HEADER
```

eliminates the table heading from the listing.

You may also specify a variable which returns the number of lines in the listing including the header.

```
CAT ":INTERNAL";COUNT Lines
```

If you don't want the header included in the count or listed, you can execute:

```
CAT ":INTERNAL";NO HEADER,COUNT Lines
```

You can skip a specified number of files before starting the listing.

```
CAT ":INTERNAL";SKIP Total_lines
```

Notice that SKIP doesn't affect the header. To skip both the header and some files you have to execute:

```
CAT ":INTERNAL";SKIP 4,NO HEADER
```

You can also list all the files in a directory which begin with a certain set of letters or you can list all the files in all the subdirectories of the directory being listed.

```
CAT "/Directory";SELECT "TEST"
CAT "/Examples/Chapter_2";ALL
```

If SKIP and SELECT are both specified, the files are SELECTed before they are SKIPped. Thus, if you execute

```
CAT "/Test_Data";SKIP 4,SELECT "NEW_DAT"
```

and there is a total of seven files in Test_Data of which 5 start with the phrase 'NEW_DAT', the two files that don't start with 'NEW_DAT' are ignored and the first four 'NEW_DAT' files are skipped. Only one file is listed.

Similarly, if ALL and SKIP are both specified, the ALL occurs before the SKIP.

The important thing to remember about these options is that they may appear in any order in the CAT statement. The options do not affect how files are listed.

Output to a String Array

Instead of printing out the file listing, you may place it in a string array. This is useful for storing the file status for later use. You can have a program check to see if a file has been created or if a password has been placed on the file. The following is an example program using a string array listing.

Note that the date substring in the CAT listing is not in a form usable by the DATE function. Convert the string to the proper form before using it.

```
 90  Latest_date=0
100  Latest_time=0
110  X=0
120  MASS STORAGE IS "/"
130  CAT Media_specifier$ TO Cat_listing$(*);SELECT Select_string$,NO HEADER
140  ON ERROR GOTO 300
150  LOOP
160    X=X+1
170    Date_check$=Cat_listing$(X,1)[56,57]&" "&Cat_listing$(X,1)[59,61]&" "&
Cat_listing$(X,1)[63,64]
180    Time_check$=Cat_listing$[67,71]
190    Date=DATE(Date_check$)
200    Time=TIME(Time_check$)
210    IF Date=Latest_date THEN
220      IF Time>Latest_time THEN Latest_time=Time
230    ELSE
240      IF Date>Latest_date THEN
250        Latest_date=Date
260        Latest_time=Time
270      END IF
280    END IF
290  END LOOP
300  OFF ERROR
310  Time$=TIME$(Latest_time)
320  Date$=DATE$(Lates_date)
330  RESET SCREENS
340  PRINT "Data was last recorded";" ";Time$;Date$;"."
```

Naming and Re-Naming Files

When you create a file on a medium you must name it so that you can refer to it when necessary.

A file name should be descriptive of the contents of that file. For instance, name a file containing addresses 'ADDRESSES'. A file containing data from an experiment might be called `TESTDAT`. If you ran the experiment a number of times, you'd probably want to know which version is which. In this case you might want to name your files, `TESTDAT1`, `TESTDAT2`, and `TESTDAT3`.

A file name consists of HP 9000 characters excluding the slash (/), less-than (<) and colon (:). A single period (.) or double period (..) cannot be used as a file name but may be used within a name. Leading and embedded blanks are deleted. Names with a length less than the maximum for the directory format are considered to have trailing blanks.

In a Structured Directory Format (SDF) file names are 1 to 16 characters long. For compatability with the HP-UX file system, limit the length of SDF file names to 14 characters.

In a Logical Interchange Format (LIF) directory, file names are 1 to 10 characters long. LIF is supported by many HP computers. If strict LIF compatability is desired, you should also restrict LIF file names to the upper-case ASCII letters (A through Z), the ASCII numerals (0 through 9), the underscore (_) and always begin the file name with a letter.

In an HP 9835/45 directory, 9845, file names are 1 to 6 characters long. If strict compatability with the HP 9835/45 is desired, you should also restrict file names to exclude the ASCII NUL (CHR$(0)), quote (") and the HP 9000 smudge (CHR$(255)) characters.

Renaming Files

If you want to change the name of a file use the RENAME statement. For example,

```
    RENAME "File_1" TO "Next_file"
```

changes the file's name from `File_1` to `Next_file`.

The only rule for RENAME is that the new file name cannot be a duplicate of any other file name in the same directory and it cannot be an illegal file name.

Saving and Storing Files

There are six ways to create a specific type of file and store a program into that file.

The first way to store a program is to save it in a DATA file. When a program is saved in a DATA file, it is saved as a series of strings, with one string per program line. This is not the fastest method of storing and retrieving programs and it's not the most compact way to store a program, but it has a significant advantage. A program saved in a DATA file can be accessed as string data by other programs. For example, this gives you a way to alter a program using another interactive program.

You use the SAVE statement to record a program in this form. Execution of the SAVE statement creates a DATA file by "listing" the program and saving the list on the medium.

Create the following program.

```
10  DISP "This program was saved in ""Test_file1""."
20  Nil=Empty_set
30  Void=Nuthin
40  IF Nil=Void THEN
50    Infinite_loop$="YES"
60    GOTO 40
70  END IF
80 Line_label:     !
90  END
```

Now save it with the following command:

```
SAVE "Test_file1:INTERNAL"
```

Since `Test_file1` does not already exist on the medium, the file is created and the program is saved in it; this is one of the statements that implicitly creates a file. If the file already exists, an error occurs.

The SAVE statement can also specify the first line of the program that you want to save and the last line you want to save. For example,

```
SAVE "Test_file1:INTERNAL",10,Line_label
```

saves every line from 10 through Line_label.

Note that when a program is saved in a file on a medium it is not erased from the system memory. It is in effect copied, not moved, to the medium. Thus, you don't have to re-load the program to use it after saving it.

The second way to save a program is to save it in an ASCII file. This is similar to saving it in a DATA file in that the program is saved as a series of strings, one string per program line. This is done with the SAVE ASCII statement.

Try the following. Alter line 10 of the preceding program to read:

```
10  DISP "This program was saved in ""Test_file2""."
```

Now execute:

```
SAVE ASCII "Test_file2:INTERNAL"
```

ASCII files can be used on a number of other HP computers; this file type is best for transporting programs from one machine to another. Note that the ASCII file type does not guarantee that the program syntaxes properly on another computer, only that the file's data is readable by another machine.

Like SAVE, SAVE ASCII can specify which fraction of the program to save using starting and ending line identifiers.

If a program contains any compiled contexts, neither SAVE or SAVE ASCII copies them to the file.

The third way to store a program is to store it in a PROG file. When an uncompiled program is stored into a PROG file, it is stored in an intermediate internal code. This is a faster method for storing and retrieving programs than DATA or ASCII and it is a more compact format to use. Its main drawback is that the data in the file cannot be accessed by a defined record as with DATA and ASCII files.

To store a program, use the STORE statement. When this is executed, a PROG file is created on the specified mass storage medium and the main program and all the subprograms in a partition's memory are stored in that file.

Storing a program in a PROG file is different than storing it in an ASCII or DATA file in that the ASCII and DATA files just store lines as strings without concern for the structure of the program. Each line is stored as a separate entity. The PROG file stores the program by context. The main program is one unit and each subprogram is another unit. You can only work with PROG files in terms of each context, not as individual program lines.

Try the following. Alter line 10 of the program to read:

```
    10  DISP "This program was saved in ""Test_file2""."
```

Now execute:

```
    STORE "Test_file2:INTERNAL"
```

CAT the volume to see the files you have recorded.

If you get an error while trying to store a large program, scratch your variables with SCRATCH V to free up some memory, then store the program.

The last way to store a program is with the COMPILE statement. This statement compiles a portion of the program in memory and then stores that object code in a PROG file. A typical example of this statement is:

```
    COMPILE MAIN TO "First_prog:CS80,7,1"
```

This example statement compiles the main context and stores it in the file `First_prog`. If `First_prog` already exists, the main context in that file is replaced with the new version. If no file exists, the file is created and the object code is stored.

COMPILEd code runs faster when first loaded than a STOREd PROG file because it is already in the machine's internal format.

You can specify either the main context as previously shown or a subprogram, function or all of the contexts. Here are some examples of the COMPILE statement.

```
COMPILE Prog1 TO "Sublib:INTERNAL"
COMPILE ALL TO "Exprog"
COMPILE FNFun1 TO "Functions"
```

Storing Non-program Files

In addition to saving programs in DATA, ASCII, or PROG files, you can store BIN files, special function keys and subprograms.

To store a BIN file you need to use the STORE BIN statement. STORE BIN creates a BIN file on the specified medium and stores all the options currently in memory into that file. This is often useful when you want to create a custom system configuration that can be loaded with one file. For example, if you wanted to have the IO option loaded automatically when the minimum BASIC system is brought up, you could create a BIN file called AUTOBN which contains the IO option. Just load the minimum system and the IO option and then execute:

```
STORE BIN  "AUTOBN"
```

The "System Configuration" appendix of this manual provides further information on configuring your software if you wish to use this feature.

To store special function keys, use the STORE KEY statement. This statement creates a DATA file and stores the definitions of all the defined keys in it.

Key files consist of an INTEGER (specifying the key number) followed by a string for the user-defined typing aid. This is followed by the INTEGER value of the next defined key and its associated string and so on. Here is an example of the STORE KEY statement.

```
STORE KEY "Definition_16:INTERNAL"
```

If you have a program stored on a mass storage medium which makes use of certain special function key definitions, you can gain better control over your user's operating environment by storing the special function key definitions (not to be confused with the ON KEY declarations which you may have in your program) and retrieving them from the program.

These files are also useful when you want to create interactive work environments. For example, you can set up files defining keys for special editing functions you wish to perform, commands you use frequently or media specifiers you use. Just place the files on the medium and load them when needed.

If you want to store specific contexts (not the entire program), use the STORESUB statement. STORESUB stores contexts in a specified PROG file. This statement is somewhat different than the other ways to store or save in that if the file already exists, the subprograms are stored there. There must be room in the file in order to store the new context. A PROG file created by STORE does not have extra room for more contexts; it only has enough space allotted to hold the data stored initially. If you want to STORESUB a context into a PROG file, the file has to have been created with CREATE PROG. CREATE PROG has a parameter with which you can specify how many contexts the file can hold. For example:

```
CREATE PROG "Experiment",100,20
```

The first value "100" specifies the length of the file in 256 byte records. The second value specifies that the file can have up to 20 contexts stored in that file. The default is 25.

The main context, each SUB subprogram and each DEF FN function is considered a separate context.

If the file does not exist when a STORESUB is executed, a PROG file is created with that file specifier. A typical example would be:

```
STORESUB Data_recorded TO "Tests/Status:CS80,7,0"
```

In this case only enough space is allotted to fit the specified context or contexts in the file.

You may specify that all the subprograms in memory are to be stored with the ALL option. For instance,

```
STORESUB ALL TO "Tests/Status:CS80,7,0"
```

Or you can just store the main context with

```
STORESUB MAIN TO "Tests/Status:CS80,7,0"
```

Getting and Loading Files

Once a file has been successfully stored on a disc, it can be recalled with the GET, LOAD, LOAD BIN, LOAD KEY, and LOADSUB statements.

If a program has been SAVEd, it is brought back into memory with the GET statement. The GET statement can be used on any of the data files: ASCII, BCD, BDAT or DATA.

Execution of the GET statement causes your system to read the specified data file and expect to find a succession of strings that are valid program lines. As the program is retrieved, each line is read in and its syntax checked to make sure it is a valid statement.

Execute these commands:

```
SCRATCH A
GET "Test_file1:INTERNAL"
```

Now see if Test_file1 is in memory.

You can alter a portion of a program residing in memory by executing a GET and specifying an appending line. The following is a typical example.

If this program is in memory: and this is saved on a medium:

```
 10  DISP "Original Program"        10  Var=10
20  !                              20  FOR I=1 TO 10
30  Var=0                          30     Var=Var-1
40  FOR I=1 TO 10                  40  NEXT I
50     Var=Var+1                   50  END
60  NEXT I
70  END
```

and you execute this statement:

```
GET "Test_file1",30
```

you have the following program in memory:

```
10  DISP "Original Program"
20  !
30  Var=10
40  FOR I=1 TO 10
50     Var=Var-1
60  NEXT I
70  END
```

GET leaves any lines less than 30 alone. All the lines from 30 and up are deleted and the new program's lines are incremented to fit onto the remaining portion of the old program.

Any line number may be used for the append location. If the line does not already exist, the new program segment is appended from the next largest line on. If the line number is greater than the current program, the new segment is appended to the end of the program.

You can also specify a line at which you want to start executing once the program is loaded. For example,

```
GET "Test_file1",30,20
```

appends the new program at line 30 and starts execution of the program from line 20. You should be very careful about where you re-start a program. For instance, don't start execution in the middle of a FOR...NEXT loop.

You can use this when your program is much too large to fit in memory at one time. By having a master program called an "executive" or "root" which loads program segments as they are needed, you can cut down on the amount of memory used.

In summary, when GET is used to load a program, the program currently in memory is affected in the following way.

- All the values for variables which are not in COM are lost. If the COM area of the new program does not match the existing COM area, the values in the old COM area are lost.
- If a starting line number is supplied, the current program is deleted from that line number to the end and the new lines are appended to the program. However, no compiled contexts or BIN options are deleted. (Use DELSUB.)
- If a starting line number is not supplied, the new program is loaded and the current program and any compiled contexts are deleted. BIN options are not deleted.

If a program has been STOREd or COMPILEd, it is brought back into memory with the LOAD statement. The LOAD statement can only be used on a PROG file. Execution of the LOAD statement erases any programs and data in memory. However, any data stored in common is preserved if the loaded program has a matching COM area.

Execute these commands:

```
SCRATCH A
LOAD "Test_file3:INTERNAL"
```

Now check to see if Test_file3 is in memory.

You cannot append programs with LOAD, but you can specify the line at which to start execution after the program has been loaded. For example,

```
LOAD "Example_prog",46
```

loads the program Example_prog and starts execution at line 46.

To load BIN files use the LOAD BIN command. For instance:

```
LOAD BIN "AUTO_SETUP"
```

LOAD BIN causes the equivalent of a SCRATCH A to occur. Thus any important programs or data must be stored away before executing it. It is usually wise to load all the binaries you think you might use during a session before beginning the session to avoid having to load the binary later.

LOAD BIN can only be executed from the keyboard so you don't have to worry about a program causing a SCRATCH A.

To load special function keys use the LOAD KEY command. LOAD KEY causes a SCRATCH KEY and then the definitions of the keys in the file are loaded. Any keys not defined in the file are left unchanged.

This operation can be done from the keyboard, but is most usefully employed in a program. For example, if the program segment

```
  .
  .
  .
46  LOAD KEY "Select"    !Load Keys with numbers 1 thru n
56  INPUT "Press...",Selection
66  ON Selection GOSUB ...
  .
  .
  .
```

is executed, and the file Select has key definitions in it which appropriately set the value for Selection, then the appropriate subroutine is chosen for execution.

LOADSUB loads a subprogram from a specified PROG file. You can specify that the system should load one specific subprogram or all the subprograms in the file. Here are example LOADSUB statements.

```
LOADSUB Subprog FROM "/Xfer1"
23  LOADSUB ALL FROM "Standards/HP1293a"
```

LOADSUB does not have any effect on the main program in memory or on any of the variables.

This is also a useful way to break up a program which is very large. By placing each segment of a program in its own subprogram, you can load only those portions of code which are immediately necessary for your task.

Re-saving and Re-storing

A program stored in a data file can be loaded into memory and edited. It can then be re-saved into the same file using the RE-SAVE statement. This saves space when you don't really need a copy of the previous version of a file.

The RE-SAVE statement can also be used to force the saving of a program into the data file of your choice. The only data file types you can SAVE a program in are DATA and ASCII. By creating a BCD or BDAT file of the proper length on the medium and then using RE-SAVE statement to store the program in that file, you can have your program in any data file.

If the file specified in the RE-SAVE statement does not already exist, a DATA file with that file specifier is created.

As an example of this statement, try creating a BDAT file and then re-saving a small program into that file using these commands.

```
CREATE BDAT "Test_file4:INTERNAL",5
RE-SAVE "Test_file4:INTERNAL"
```

Now GET the file TEST to confirm that the file contains the program.

Like SAVE and SAVE ASCII, RE-SAVE can store a portion of a program by specifying starting and ending lines.

```
RE-SAVE "Test_file:INTERNAL",Report_line,89
```

It is important to note that the old file is not erased until the new file has been successfully recorded. Therefore, there must be room for both files on the medium.

As with SAVE and SAVE ASCII, RE-SAVE doesn't save compiled contexts.

A program stored in a PROG file can also be loaded into memory, edited, and then re-stored into the same file using the RE-STORE statement (not to be confused with the RESTORE statement). Again, this saves space when you don't really need a copy of the previous version of a file.

Try loading Test_file2 into memory, editing a few lines and then re-storing it. Then load it again to check whether the edited file has been recorded.

If you give a file specifier that doesn't exist, the RE-STORE statement creates a new PROG file at the location given and stores the program there.

Execute the following statement:

```
RE-STORE "Test_file2:INTERNAL"
```

Now CAT the file tree to see the results.

Note that as with RE-SAVE, there must be room for both the old and new files on the medium for the RE-STORE to occur.

You may also re-store BIN files and special function keys with the RE-STORE BIN and RE-STORE KEY statements respectively. The RE-STORE BIN statement stores all the binary programs currently in memory into the specified BIN file. If the file does not exist, a BIN file is created with that name. You can only RE-STORE BIN to a BIN file. That is, you can't "force" another file type. RE-STORE BIN is useful when you want to change the contents of a BIN file you have already created.

The RE-STORE KEY statement is similar. It stores all the special function key definitions in the specified file. If the file does not exist, a DATA file is created with that file specifier. This is used mainly to change the contents of a file containing key definitions or to force the file type of the key definitions to something other than BDAT.

As with RE-SAVE and RE-STORE, you must have room on the medium for both the old and new files.

Moving Files

You can also move a file within the file tree of an SDF volume. This is done with the RENAME statement. By giving a different file specifier for the new file name, you can not only change the name of the file, but the location of its directory entry. The contents of the file are not moved on the medium, only the file's directory entry. In addition, files cannot be moved across volumes. For instance, execute the following:

```
RENAME "Dir1/Dir2/Test_file2:INTERNAL" TO "Dir1/Dir3/Test:INTERNAL"
```

Now CAT the file tree and see where Test is located.

Copying Files

Copying files is often done when you want to "backup" important data, when you want to give someone else a copy or when you want to edit the file, but keep an original for other use.

This can be done with the COPY statement. Try the following:

```
    COPY "Dir1/Dir3/Test:CS80,7,1" TO "..:INTERNAL"
```

Check to see if Test now exists in both places.

COPY is also useful for moving files across volumes. RENAME does not work across a volume, but COPY does. You can then go back and eliminate the original file from the original volume.

Purging Files

You may find that your volume has become cluttered with old files. You can delete these files with the PURGE statement. It is important to remember that a purged file cannot be recovered. Be sure that you want to delete it before executing the PURGE command.

Note also that the remaining files are not automatically packed together. Thus, any space used by the purged file remains at the same location on the disc. This space is re-used whenever a file can fit into that spot on the disc.

Thus, you may create a file and then execute a CAT and look for the file at the end of the listing. It may not be there. It may be somewhere in the middle of the list.

Protecting Files

You can protect files from accidental destruction or use by others by using a series of passwords. This is useful if you want to make a single mass storage medium available for use by a large number of people.

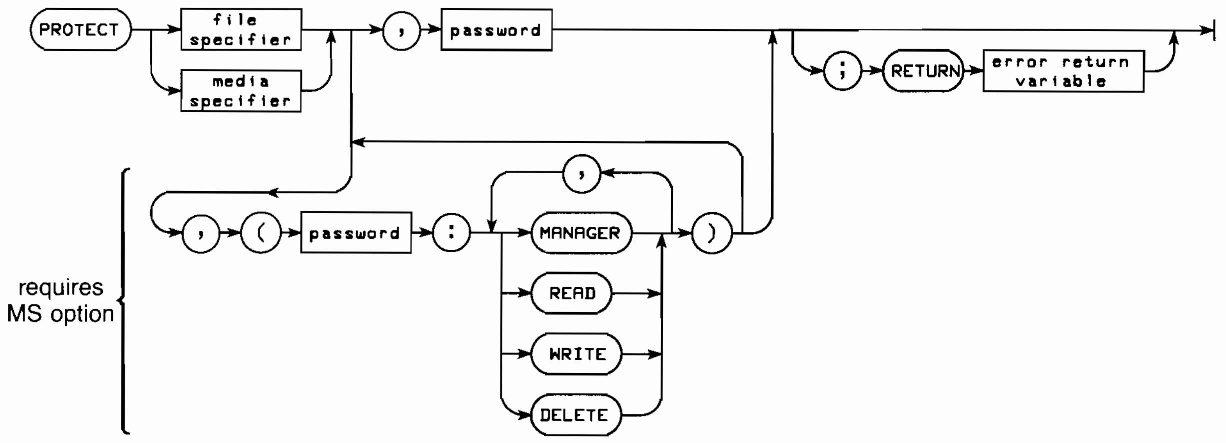
A password guards against accidental changes to an individual file, a particular branch of a file tree or an entire volume. A good system of passwords can create a more secure environment in the volume.

With LIF and 9845 formatted media, you can place a single password on the file which protects the ability to write to the file or to manage the file (purge the file or change the password).

With an SDF medium, you may place multiple passwords on any of three capabilities associated with a file. You may protect: the ability to read a file, the ability to write to a file, and the ability to manage the file (includes reading, writing, purging and placing passwords on a file).

The PROTECT Statement

There are a variety of ways to place passwords on files. The most common way is using the PROTECT statement. Here is the syntax diagram for that statement.



As you can see, there are two basic ways to specify a password; either a general password on the file or a password for specific capabilities. Here is an example of a simple PROTECT statement.

```
PROTECT "File1:INTERNAL","SP"
```

In the preceding example, the file, `File1`, was protected with the password SP. Since the statement doesn't specify a particular capability to protect, such as reading or writing, the password protects the MANAGER and WRITE capabilities. This is true regardless of what disc format is used. Notice that the ability to read the file is not protected.

The Structured Directory Format is the only one which can protect individual capabilities on a file. Here is an example of protecting a specific capability.

```
PROTECT "File1:INTERNAL",("SP":READ)
```

If this form of PROTECT is used with LIF or 9845 formatted media, the capability is ignored and the password SP still protects the ability to write to the file, purge the file or change the password on the file.

For Structured Directory Format, the password SP protects the ability to read the file `File1`.

A CAT listing shows which capabilities of a file are protected. Or to be more specific, it lists which capabilities are not protected. Here is a typical CAT listing.

```
:INTERNAL
LABEL:  TEST
FORMAT: SDF
AVAILABLE SPACE: 488
                             SYS  FILE    NUMBER    RECORD      MODIFIED       PUB OPEN
FILE NAME               LEV TYPE  TYPE   RECORDS    LENGTH DATE           TIME ACC STAT
==================== === ==== ===== ======== ======== =============== === ====
GAMES                     1        DIR        10        24 21-Jul-82  08:43  R
EXAMPLES                  1        DIR        21        24 24-Sep-82  15:16 MRW
DEMOS                     1        DIR        10        24 21-Aug-82  16:49
TOOLS                     1        DIR        32        24 28-Aug-82  08:01 MRW
SYMBOLS+FONTS             1        DIR        21        24 26-Aug-82  08:47  RW
```

Notice the next to the last column (labeled PUB ACC). This column contains three letters, M, R and W which stand for MANAGER, READ and WRITE. If a letter is listed in a file's row, that capability is available for public access. If the spot where that letter is expected is blank, that capability is protected for that file.

The exact password string cannot be recovered if a password is forgotten. It may be possible to delete it or create another password protecting the same capability, but it is valuable to record your passwords somewhere for reference.

Note

If you forget a password, you can use the FIND_PASSWORD utility to recover it. See the "Utilities" appendix of this manual for more information.

Passwords

Passwords may be two characters long with a LIF medium, six for a 9845 medium and sixteen characters with an SDF medium. The only character not allowed is ">". If more letters are specified in the PROTECT statement than the disc format can use, the password is truncated at the maximum allowable. Thus for a LIF disc, SECRET and SECURE are equivalent passwords. Uppercase letters are interpreted as different from lower case letters. Thus, Password is different from PASSWORD.

The following examples show multiple passwords being specified for individual capabilities.

```
    PROTECT "Dir1/Test:INTERNAL",("Secret":MANAGER),("#(":READ)
    PROTECT "Dir1/Dir2:INTERNAL",("4312":WRITE),("2123":READ)
```

Secret protects the manager capability of the file `Test` and #(protects the read capability. 4312 protects the write capability to Dir2 and 2123 protects the read capability.

Note that if `Test` is on an SDF disc, the various passwords protect the associated capabilities. However, if Test is on a LIF or 9845 disc, an error results since only one password is allowed on a 9845 or LIF file.

Once these passwords have been placed on the capabilities of a file, they must be included in its file specifier whenever you need that capability. For instance, if you wanted to change the passwords on `Test`, you would need to give the password Secret in the file specifier used in the command. Example:

```
PROTECT "Dir1/Test<Secret>:INTERNAL",("#New":READ)
```

If you want to read `Test`, you must include the password #(or the password #New. For example,

```
LOAD "Dir1/Test<#(>:INTERNAL"
LOAD "Dir1/Test<#New>:INTERNAL"
```

Passwords do not affect the capabilities on files further down the file tree. For example, the password 4312 protects the ability to place new file entries in directory `Dir2`. It does not affect the ability to write to any files in `Dir2`. The password 2123 affects the ability to purge `Dir2` and, therefore, all the files in `Dir2`, but it does not permit you to purge individual files in `Dir2`.

For an SDF medium, the same password can be used for more than one capability. For example,

```
PROTECT "Dir1/Test:INTERNAL",("Test":READ,WRITE)
```

protects the ability to read from or write to `Test` with the password Test.

You may also have the same capability protected by more than one password.

To delete the password on a file in a LIF or 9845 volume, execute the following:

```
PROTECT "FILE<Password>",""
```

where FILE is the file from which you want to strip the protection. Password is the password on the file. The pair of quotes with nothing in between places a null password on the file.

If the previous statement is executed with an SDF medium, the password is not eliminated, only redefined to protect nothing. Thus, if it has previously protected the write capability of a file, you can write to the file without the password. However, space is still taken up by the password on the medium.

DELETE can also be used to eliminate passwords on any medium. For example,

```
PROTECT "TEST<Password>:INTERNAL",("Password":DELETE)
```

deletes the password Password from the file.

The PROTECT statement may also be used to redefine the capability set of a password. The previous capabilities protected by the password are deleted and the newly specified set of capabilities are protected.

PROTECT can also place passwords on whole SDF volumes. This password gives you all capabilities for all files in that volume. to protect a volume , use the LABEL phrase of the media specifier. For example,

```
    PROTECT ":LABEL Volume","Password"
```

protects Volume with the password Password.

Any file, regardless of its protections, can then be accessed by including an msus with the LABEL phrase and the volume password. For instance,

```
    PURGE "FILE:LABEL Volume<Password>"
```

This password also protects the volume from accidental initialization because it must be included in the media specifier of the INITIALIZE statement.

Note that protecting the volume is not the same as protecting the root directory. The volume protection is for all files in the volume; the root protection is of a specific attribute of the root directory.

You can only place one password on the volume. LIF and 9845 volumes can only have one password per file. SDF volumes can have as many passwords as fit on the media.

Other statements may be used to place passwords on files. The various CREATE statements can place a password on the file to be created. Here is a typical creation of a file with a password.

```
    CREATE ASCII "File1<Password>",5
```

Upon creation of the file File1, the password Password protects the manager and write capabilities on the file. With all the CREATE statements, the password is assumed to protect the manager and write capabilities.

You can also place a password on a file with the SAVE, SAVE ASCII, STORE, STORE BIN and STORE KEY statements. These statements are similar to the CREATE statements. Just include a password when the statement is executed and it protects the manager and write capabilities.

The RE-SAVE, RE-STORE, RE-STORE BIN and RE-STORE KEY statements can also place a password on a file if that file has not already been created. The format is the same as used for the statements mentioned above.

HP-UX vs. BASIC

When an HP-UX file is accessed from BASIC, the file's HP-UX mode corresponds to the file protection of a BASIC file. The HP-UX file's read and write permissions for the class of users "other system users" (not the file's owner nor the file's group) are interpreted as the read and write protections of a BASIC file. The file's execute permission (as specified by its mode in HP-UX) is ignored when the file is accessed from BASIC.

The read, write and execute permissions of an HP-UX directory affect the manner in which the directory (and the files it contains) can be accessed from BASIC. The interpretations of the different protections for directories is shown in the following table.

HP-UX Protection	BASIC Protection
r - (read permission) allows the user to list the contents of the directory.	READ - allows the user to list the contents of the directory with the BASIC CAT statement or read a file.
x - (execute permission) allows the user to search the directory. This permission must be set inorder to access a file below the directory in the file system hierarchy.	No equivalent BASIC protection.
w - (write permission) allows the user to add and remove files from the directory.	WRITE - allows the user to create or delete entries from the directory or write to a file.
No equivalent HP-UX protection.	MANAGER - allows the user to add or delete passwords and to purge a file.

For an HP-UX file to be accessed from BASIC, all directores in the file's path must have both the read and execute permissions enabled. When an HP-UX file or directory is examined with the BASIC CAT statement, it is not possible to determine whether or not its HP-UX execute permission is enabled.

When creating files for use by both the BASIC Language System and HP-UX, assign no file protection to the file or to the directories containing the file. This ensures that both systems can access the file.

Chapter 22
File Access and Data Transfer

At several points in the discussion in the last chapter, "writing" and "reading" of individual data items is mentioned. Some files, PROG, BIN and DIR files, cannot be "read" or "written to" in this sense because they have no separate data items. These files are accessed as a whole. The other types of files, ASCII, BDAT, BCD and DATA, can be accessed as a whole with statements such as RE-STORE or GET, but they can also have individual pieces of data "read" or "written". This is useful in a number of ways.

- Instead of using up memory for data which is only occasionally needed, you can place the data in a file and bring it into memory as necessary.
- You can save data for later use.
- You can have more than one program access the same data.

Here are the steps necessary to write data to and to read data from a file.

1. An I/O path between memory and the file is opened. This is done with ASSIGN @... or ASSIGN #.
2. An accessing mode is chosen, either serial or random.
3. The data is sent from memory to the file or from the file to memory using OUTPUT, OUTPUT USING, OUTPUTBIN, ENTER, ENTER USING, ENTERBIN, PRINT # or READ #.
4. The I/O path is closed with ASSIGN @... or ASSIGN #.

In addition to these steps, you may also define a file as the current standard printer, buffer the input and output, preview the data to be read, cause program branching after data is read and lock files for input and output.

Statement Summary

The following table summarizes the mass storage statements discussed in this chapter.

To perform this operation:	Use this statement:
Open or close a file I/O path	ASSIGN @... or ASSIGN #
Alter the current output form in a BDAT file	ASSIGN @...;FORMAT ON or ASSIGN @...;FORMAT OFF
Output data	OUTPUT PRINT # or TRANSFER
Output data using an image specifier to a BDAT file	OUTPUT...USING
Output 8 bit or 16 bit binary values to a BDAT file	OUTPUTBIN
Assign a BDAT file as the current standard printer	PRINTER IS
Input data	ENTER READ # or TRANSFER
Input data using an image specifier from a BDAT file	ENTER...USING
Input 8 bit or 16 bit binary values from a BDAT file	ENTERBIN or ENTER...USING with "W" or "Y"
Verify output	ASSIGN @...;CHECKREAD ON/OFF or CHECKREAD
Preview a data item	TYP
Branch a program on an EOR or EOF error	ON END or ON EOR/ON EOT
Lock access to a file by other partitions	LOCK
Unlock access to a file by other partitions	UNLOCK
Return a file's LOCK/UNLOCK status	IOLOCK
Return the values of a data file's I/O path name status registers	STATUS
Use a data file I/O path name status register value in a function	IOSTAT
Alter a data file I/O path name status register	CONTROL

Opening a File I/O Path

Before individual data items can be placed into or copied from an already created file, you must "open" the file. That is, you must create an I/O path between memory and the file. This is done with ASSIGN @... or ASSIGN #.

ASSIGN @...

The first way to open an I/O path is with the ASSIGN @... statement. Here are some examples using ASSIGN @... with file specifiers.

```
ASSIGN @Data_file TO "Test_data:INTERNAL"
ASSIGN @Materials TO "Materials_list"
ASSIGN @Io_path TO "Texts/Glossary/A_E:CS80,7"
```

In each case the specified I/O path name, @Data_file, @Materials or @Io_path, is assigned to an I/O path from memory to the specified file. These are "open" I/O paths.

An I/O path name may be used any number of times in a program and may be re-assigned to another file at any time by simply executing another ASSIGN @... For example, in this program:

```
   .
   .
   .
 112  ASSIGN @File_1 TO "Keep1"
   .
   .
   .
1000  ASSIGN @File_1 TO "Keep2"
   .
   .
   .
1212  ASSIGN @File_1 TO "Keep3"
   .
   .
   .
```

@File_1 is first assigned to Keep1, then to Keep2, and finally to Keep3. Each re-assignment cancels out the one before it. This is done by opening the new I/O path then closing the old one. If after line 1000 the program loops back to 112, @File_1 is re-assigned to Keep1 again.

More than one I/O path name may be assigned to a single file. References to the separate I/O path names are the same as references to different files. This has a number of uses which are explained further on in this chapter.

ASSIGN

The other way to open an I/O path is to assign a "file number" to it. There are ten file numbers available for use, 1 thru 10. To assign a file number to a file, you must use the ASSIGN # statement. For example,

```
ASSIGN #1 TO "Population_data"
```

assigns file number #1 to the file "Population_data". Note that unlike the I/O path name the file number can be a variable. Thus,

```
ASSIGN #File_number TO "Population_data"
```

is a valid statement.

As with an I/O path name, a file number may be used any number of times in a program and may be re-assigned to another file at any time by simply executing another ASSIGN #.

As with ASSIGN @..., more than one file number may be assigned to a single file. References to the separate file numbers are the same as references to different files.

I/O Paths in Contexts

I/O path names and file numbers can be passed into subprograms and functions or declared in COM; however, any I/O path names or file numbers assigned in a subprogram are deallocated when the subprogram is left. For example:

```
  .
  .
  .
 115  ASSIGN @Data TO "Test_data:INTERNAL"
 125  CALL Sort(@Data)
  .
  .
  .
1145  SUB Sort(@Data)
1150  ASSIGN @Data TO "Text_file:INTERNAL"
1155  ASSIGN @Comments TO "Information:INTERNAL"
  .
  .
  .
```

When the SUB subprogram is entered the I/O path to "Test_data:INTERNAL" is closed because the I/O path name, @Data, is re-assigned to "Text_file:INTERNAL". Upon return to the calling context, the I/O path to "Information:INTERNAL" is closed, but the I/O path to "Text_file:INTERNAL" is still open.

In general, if a file is opened in a subprogram, it is closed automatically upon leaving that subprogram. However, if the file number or I/O path name is in COM or passed-by-reference to the subprogram which opens the file, that file stays open in the context which passed the file number or I/O path name. It is then closed upon leaving that context.

For more information, see the "Subprograms and Functions" chapter of this manual.

Using ASSIGN @... Vs. ASSIGN

Using ASSIGN @... is the recommended method of opening an I/O path. It has more options for transferring data. It also creates more readable code than ASSIGN #.

ASSIGN # is offered for compatibility with the HP 9835/45. It is also used in applications where a numeric expression determines the I/O path. If you are trying to conserve memory, ASSIGN # may be more useful because ASSIGN @... requires the IO option at a cost of more than 120 000 bytes.

Access Methods

There are two methods for sending data to or receiving data from a data file: via serial access or random access. **Serial access** is the input of data from or output of data to a file sequentially. **Random access** is the input from or output to a specific defined record within a file.

Serial access reads or writes data in order from the beginning of the file to the end of the file. Random access starts at the beginning of a defined record which you specify. Reading and writing can take place at any defined record in the file.

The main advantages of serial access are that it creates compact data storage for output operations and is a quick way to work with large quantities of similar data.

The advantage of random access is that access to specific small portions of data is quicker than with serial access.

The disadvantage of random access is the potential for large amounts of wasted space. The defined-record size determines how much space is needed to store each data item and the proper choice of defined-record size can yield a fairly compact storage scheme. An improper choice can be wasteful.

You can mix access modes as necessary with the same I/O path if that file supports both modes.

File Structure

This section describes in detail how the various data files store information.

DATA Files

A DATA file is a series of defined records which contain two types of data: numeric data and strings. INTEGERs and DOUBLEs are represented in two's complement form and SHORTs and REALs are represented in IEEE floating point. Here is how they are coded:

Data Type:	Output:
INTEGER	1 byte header and a 2 byte number
DOUBLE	1 byte header and a 4 byte number
SHORT	1 byte header and a 4 byte number
REAL	1 byte header and an 8 byte number

Each string starts with a five byte header (one byte to identify it as a string and a four byte DOUBLE specifying the length of the string) and has one byte per character, four bytes every time a defined record is crossed (specifying the string length remaining).

An advantage of DATA files is that the system can identify the data using the headers and automatically convert it to the proper type for the variable it is assigned to in a read operation. If the type can't be converted, such as when the value is too big for the type, an error is returned. This makes it much easier for you to use data. You don't have to worry about consistency in your data types as much. For example, if you had a file full of INTEGERs and you input them into variables of the type DOUBLE, the system reads the headers of the INTEGERs, recognizes them as such and places the value in the two low order bytes of the DOUBLE variable. The two high order bytes are set to zero.

You can read and write to any defined record in a DATA file.

BCD Files

The BCD file structure is similar to that of a DATA file. It is a series of defined records which contain numeric data in binary coded decimal and strings stored in form similar to that used in a DATA file. Unlike DATA files, BCD files do not support the DOUBLE data type. DOUBLEs are stored as INTEGERs; if the DOUBLE value is not in the range of INTEGER (−32 768 thru 32 767), an error occurs. This is how the other numeric data types are coded.

Data Type:	Output
INTEGER	2 byte header and a 2 byte number
SHORT	a 4 byte number
REAL	an 8 byte number

The SHORT and REAL data types do not need a header to identify their type because their numeric format uniquely identifies them as SHORTs or REALs.

Strings are stored with a four byte header (two bytes to identify it as a string and two bytes to specify its length). There are an additional four bytes every time the string crosses a defined record. If a string has an odd number of bytes, an extra byte is added to make it even. This extra byte is not included in the length of the string.

As with DATA files, you can read and write to any defined record in the file.

ASCII Files

An ASCII file consists of a series of strings which run sequentially from the beginning of the file to the end. Each string starts with a two byte header which specifies the string length in bytes. The data type of this header is INTEGER; thus, it can have a value from zero thru 32 767. This header is followed by the sequence of characters in the string. Each character in the string is specified by a one byte (eight bit) code. Since the character is represented by eight bits, it can be any of the HP 9000 characters (CHR$(0) through CHR$(255)).

Since the header specifies the string length, up to 32 767 bytes (or characters) can be included in the string. Strings of an odd length are stored with an extra trailing byte so that there is an even number of bytes stored. This extra byte is not included in the string length.

All numeric data is stored in STANDARD string numeric form. You can read numeric data back as string data or numeric data. If the data is brought back into memory as numeric data, a number builder is used to construct numeric values. The "Introduction to Input" chapter of this manual provides more information.

The only separation between two strings is the header of the second string. Thus, the system examines the header of the first string, reads in exactly the number of bytes specified in the header and then interprets the next two bytes as the header for the following string. This structure forces the data in an ASCII file to always be read and written serially.

The number and length of strings is up to you. You can have a file consisting of one long string or a string for each character. One consideration is the overhead caused by a two byte header before each string and the rounding up of odd strings. For example, if you have a string for each character, half of the file is taken up by headers specifying the length of the strings. One quarter is taken up by blank bytes to round up the string to two characters. Only one quarter is data.

When a program is stored using SAVE ASCII, each line of the program, including line numbers, is stored as one string. The End-Of-Line sequence (Carriage Return - Line Feed) is not stored with the line.

LIF ASCII files from other HP systems such as the HP 2642A may have Carriage Return - Line Feed End-Of-Line sequences in the string.

BDAT Files

BDAT files can have one of two structures: one with FORMAT ON and the other with FORMAT OFF. FORMAT is an I/O attribute which is specified with the ASSIGN @... statement. Here are two examples:

```
ASSIGN @Path_1;FORMAT ON
ASSIGN @Hp_9895;FORMAT OFF
```

FORMAT ON specifies that the data is output as one or more character strings with no header, that is, the format output by CAT, DBSTATUS, LIST and PARTITION STATUS. FORMAT OFF specifies that the data is output in the internal format of the system.

BDAT files are the only ones that support FORMAT ON; all others use FORMAT OFF.

With FORMAT ON, the data is ouput in the following form. All numeric data is evaluated and converted to ASCII characters and output in the STANDARD format. Strings are also output as characters. There are no headers in this data; it is simple string data.

With FORMAT OFF, the data is output in the following manner.

Data Type:	**Output:**
INTEGER	2 Bytes (No Header)
DOUBLE	4 Bytes (No Header)
SHORT	4 Bytes (No Header)
REAL	8 Bytes (No Header)

The INTEGER and DOUBLE data types are output in two's complement form and the SHORT and REAL data types are output in IEEE floating point.

Strings are output in a form similar to that used with ASCII and BCD files. There is a four byte header, one byte per character, and the string is always rounded up to an even number of bytes. The four byte header is a DOUBLE value specifying the length of the string.

You can define the record size in a BDAT file to be one byte, thus giving you a byte addressable file.

The important thing to remember about BDAT files is that numeric values do not have a header before the data. This makes the file much more compact (A BDAT file containing just INTEGERs is only half as long as a BCD file containing the same data.), but it also requires more care when reading and writing the data. You must read data back into the same type as you wrote it out.

BDAT files are very useful when you have a program debugged and running smoothly. They are not as good when you are having problems with your code. Generally, you should edit and debug code using a DATA file and then switch over to BDAT when everything is working.

Summary

The following table is a summary of the various file structures.

Feature	File Type				
	ASCII	DATA	BCD	BDAT (Format) (ON)	BDAT (Format) (OFF)
Defined records	No	Yes	Yes	Yes	Yes
Serial Access	Yes	Yes	Yes	Yes	Yes
Random Access	No	Yes	Yes	Yes	Yes
INTEGER type field	No	1 byte	2 bytes	No	No
INTEGER form	STANDARD[1]	16 bit 2c	16 bit 2c	Note 2	16 bit 2c
DOUBLE type field	No	1 byte	Forced to INTEGER	No	No
DOUBLE form	STANDARD[1]	32 bit 2c	16 bit 2c	Note 2	32 bit 2c
SHORT type field	No	1 byte	No, part of exponent	No	No
SHORT form	STANDARD[1]	32 bit fp	32 bit bcd	Note 2	32 bit fp
REAL type field	No	1 byte	No, part of exponent	No	No
REAL form	STANDARD[1]	64 bit fp	64 bit bcd	Note 2	64 bit fp
String type field	No	1 byte	2 bytes	No	No
String length header	2 bytes	4 bytes	2 bytes	No	4 bytes
EOR form[3]	No	1 byte	2 bytes	No	No
EOF form[3]	2 bytes	1 byte	2 bytes	Pointers[4]	Pointers[4]

Legend: 2c, two's complement binary
fp, floating point binary
bcd, binary coded decimal

1 Numerics are output as character string numerics in STANDARD form.

2 Numerics are output as character string numerics in the prevailing FIXED, FLOAT, STANDARD or ...USING... form.

3 EOR/EOF marks written only if data stops before end of record or file.

4 File pointers (all file types) accessible only via I/O path name.

File Pointers

To perform serial or random access, the system uses file pointers. A **file pointer** is one or more locations in memory which point to the next byte to be accessed. When an I/O path is first assigned to a file, the file pointers are created and their positions are initialized to the first byte in the file. As bytes are read or written, the file pointers are incremented to point to the next byte.

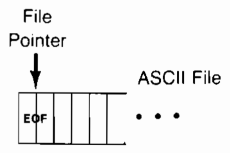
There are two type of conditions in the file which affect this behaviour: End-Of-Records (EORs) and End-Of-Files (EOFs). An EOR signals an end of a defined record in the file and an EOF signals an end of the file. As is explained further on, Some files only use EOFs and some use both EORs and EOFs. Some files can have more than one EOR and EOF. Some files can only have one EOF.

When a serial read operation is performed, the file pointer increments to each succeeding byte or data item value in a defined record until it finds an EOR. This causes the file pointer to jump to the beginning of the next defined record, reading all of its data and so on. When the file pointer finds an EOF, it stops and reports an End-Of-Files error. You cannot read beyond an EOF in serial mode.

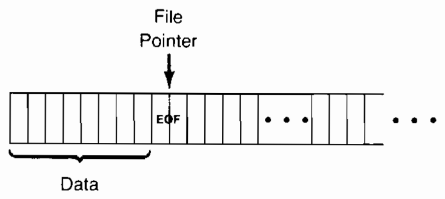
When a random access read operation is performed, the file pointer points at the first byte in the defined record specified. The pointer increments to each succeeding byte in the defined record until it finds an EOR or EOF and then stops. You cannot read past an EOR during a random access read.

ASCII Files

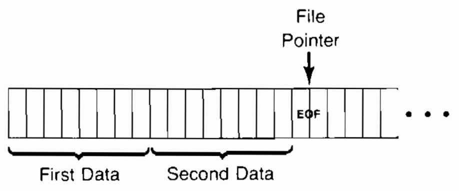
For an ASCII file, an EOF is written at the very beginning of the file when it is first created. The EOF is a two byte data item.



As data is written to the file, the EOF is overwritten but a new one is written at the end of the data item. Thus, the file pointer is still pointing at the EOF.



If another data item is written to the file, the EOF is again overwritten and a new one is placed at the end of that data item. Again, the file pointer points to the EOF.



If a read operation is then executed, the system immediately returns an End-Of-File error because the file pointer is pointing at the EOF. The only way to read the data in the ASCII file is to reset the file pointer back to the beginning of the file. This can be done in a number of ways.

- If you use an I/O path name, you can reset the file pointer using the RESET statement.

  ```
  RESET @Io_path
  ```

  where @Io_path is the I/O path name of the file.

  This statement sets the file pointer to the first byte in the file.

- You can assign a new I/O path name or file number; these always start with the file pointer pointing to the first byte in the file.
- You can also move the file pointer of an existing I/O path by executing a random read or write to the first record while actually reading or writing nothing. This is an exception; normally ASCII files cannot be accessed randomly. Although the following statements are explained more carefully later in this chapter, here is how you can reset the file pointer:

  ```
  ENTER @Io_path,1
  ENTER USING @Io_path,1
  ENTERBIN @Io_path,1
  READ #1,1
  ```

  No EORs are available with ASCII files. This is why ASCII files can only be accessed serially. There is no way for the system to find any defined records except the first one.

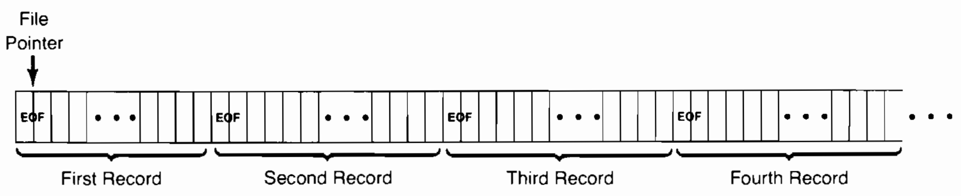
  Once you have read the data you need, you can set the file pointer to the end of the file by continuing to read until the EOF is reached.

BCD and DATA Files

For BCD and DATA files, EOFs are placed at the beginning of each defined record when the file is first created. The EOFs and EORs are two byte data items in a BCD file and one byte data items in a DATA file.

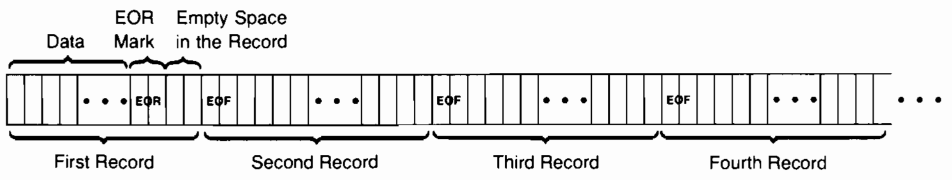
As you output to the file, the EOFs are overwritten. In addition, after each output operation, an EOR is written unless the item completely fills the defined record. In this way, the system ensures that you read back only current data.

For example, if you create a DATA file and try to read from it, there is an EOF in the first byte so no data is returned.



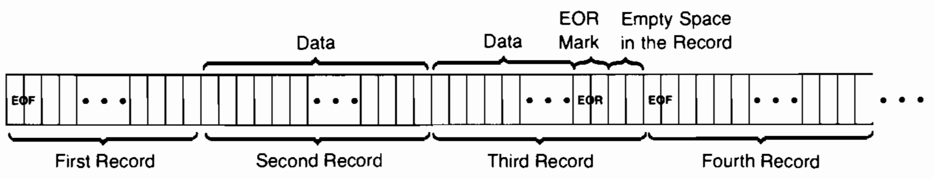
This is what you would expect since you haven't written any data to the file.

Now if you write some data to the first defined record, the EOF is overwritten and an EOR is output in the byte following the data.



If you read the first defined record, you get the data that you just wrote, then the system encounters the EOR in the next byte and jumps to the next defined record. The first thing it finds is an EOF in the first byte of that record so it stops.

Now imagine that instead of outputting information to the first defined record, you output data to the second defined record.



If you try to read the second defined record, you get that data back, and then the operation stops. If you try to read the first defined record, the system immediately finds an EOF and doesn't input any data.

BDAT Files

BDAT files are different from the other data files in that EORs and EOFs are not written in the file. All the information about the end of the files is computed by the system rather than discovered by a file pointer. Although random access is possible, no EORs are used with a BDAT file. Thus, if a record is filled with data and you write a new item at the beginning of that record, a read operation not only returns the new data item but interprets the following bytes as valid data.

Writing Data

You can write data to a file using either access method, serial or random.

Via I/O Path Names

There are three statements which can output using an I/O path name.

- The OUTPUT statement outputs data to ASCII, BCD, BDAT and DATA files.
- The OUTPUT...USING statement outputs data to BDAT files using an image specifier.
- The OUTPUTBIN statement outputs eight and sixteen bit binary values to BDAT files.

Serial OUTPUT

OUTPUT is the most commonly used of these three statements. Here are some example serial OUTPUT statements.

```
OUTPUT @Bdat_file;Character$(3,2)[6;32]
OUTPUT @Io_path;Data1;Data2,Data$
```

The syntax of a serial OUTPUT is simple. A semicolon separates the I/O path name and the data items to be sent. The data items are separated by semicolons or commas. If FORMAT is ON, comma delimiters output a comma after a numeric item and a carriage-return line-feed after a string item. Semicolon delimiters output nothing after an item. If FORMAT is OFF, the delimiters send nothing.

OUTPUT with FORMAT ON only works with BDAT files. OUTPUT with FORMAT OFF works with all data files.

Insert an initialized disc into the INTERNAL flexible disc drive and execute the following program.

```
10   MASS STORAGE IS ":INTERNAL"
20   CREATE DATA "Test_file",1   ! Allocate 1 defined record
30                               ! of 256 bytes.
40   ASSIGN @Practice TO "Test_file"
50   ALLOCATE INTEGER Numbers(1,10)
60   READ Numbers(*)
70   DATA 1,2,3,4,5,6,7,8,9,0
80   OUTPUT @Practice;Numbers(*)
90   END
```

The ASSIGN @... statement creates the I/O path. An INTEGER array is then dimensioned and some numbers are read into it. Finally, an OUTPUT statement writes the numbers to the file. Recall from the last chapter that an INTEGER requires four bytes in an DATA file. Thus, of the 256 bytes originally allocated for "Test_file", 40 bytes have been used starting with the first byte in the file. The 41st byte of the file is an EOF.

Random OUTPUT

OUTPUT can write to one record at a time by specifying the record which is to receive the data. Here is the syntax:

```
OUTPUT @Io_path,Record_number;Data1,Data2,Data3
```

Notice that this form of the statement is quite similar to the serial form. The only difference is the addition of the record number before the semicolon. In this case the data in the data list is output to the file specified by the I/O path name and to the particular record specified by the record number.

FORMAT ON only works with BDAT files, but FORMAT OFF is usable with all data files.

If you attempt to write more data to a record than the record holds, the system returns an error. The data already written to the file remains intact.

Omitting the data list causes an EOR to be written at the current position of the file pointer. This means any data in the remainder of the defined record is inaccessible.

OUTPUT...USING
OUTPUT...USING is the second way to output data. This statement outputs data to a file using a format specified by you. OUTPUT...USING can only be used with BDAT files because it is executed with FORMAT ON which is a form only BDAT files can record.

The format is a series of characters without any header. All numerics are converted to an image specified string form before being output. The form of the characters is defined by an image specifier which are discussed in the "Image Specifier" chapter.

OUTPUT...USING can be executed both in serial and in random accessing mode.

This statement is typically used for text applications where compact storage is a major requirement.

OUTPUTBIN
OUTPUTBIN is the third way to output data to a file. OUTPUTBIN sends eight bit or sixteen bit binary values to a BDAT file. OUTPUTBIN is always executed with FORMAT ON so it can only be used with BDAT files.

User-Defined EOF
Upon the creation of a BCD or DATA file, every defined record has an EOF placed into it at the beginning of the record. These EOFs are written over as you output to the defined records. Thus when you are using a file for the first time, after you finish your serial output statements, there is always at least one EOF following your data to indicate the file is complete, until you actually fill the entire file with data.

However, if you are re-using a file (one that has been previously written to in either a serial or random fashion), where you finish outputting your data , there may be some "old" data remaining and no EOF. This could cause difficulties with future uses of the file - trying to determine where the new data stops and the old data begins.

To overcome this difficulty, use the END attribute. By placing the literal END after the data list in an OUTPUT or OUTPUT USING statement, a single EOF is written to the file at the end of the last defined record used. This writes over the EOR which is written at the time of the file's creation. You can use this EOF to detect the end of your data.

Here is an example of this attribute.

```
    OUTPUT @Data_file;Data_array$(*),END
```

END updates EOF pointers in a BDAT file and writes an EOF in an ASCII file. It also flushes the file's buffer.

Via File Numbers

You can also output data using the ASSIGN # and PRINT # statements.

Serial PRINT

The syntax of the serial PRINT # statement is similar to that of OUTPUT.

```
PRINT #2;Var1,Var2,Var3$,Var4(*),Var5$(*)
PRINT #File_number;Variable$
```

A semicolon separates the file number and the data list to be output. The data items are separated by commas or semicolons. PRINT # can be used with ASCII, BCD, BDAT and DATA files because the FORMAT is always OFF for PRINT #.

The following is an example of serially printing data to the file `Testdat`.

```
10  MASS STORAGE IS ":INTERNAL"
20  CREATE DATA "Testdat:INTERNAL",1
30  ASSIGN #1 TO "Testdat"
40  PRINT #1;"First",24,2*A6,New$(*)
50  END
```

PRINT # has the same syntax as the HP 9835/45 PRINT # statement.

Random PRINT

You can also use PRINT # with random access mode. The syntax of the statement is similar to that used for OUTPUT in random access mode.

```
PRINT #File_number,Record;Data_list
```

Notice that this form of the statement is quite similar to the serial form. The only difference is the addition of the record number before the semicolon. In this case, the data in the data list is output to the file specified by the file number and to the particular record specified by the record number.

Again, PRINT # is always executed with FORMAT OFF.

If you attempt to write more data to a record than the record holds, the system returns an error. The data already written to the file remains intact.

Omitting the data list causes an EOR to be written at the beginning of the record. This means any data in the remainder of the defined record is inaccessible.

User-Defined EOF

You can also use the END attribute with the PRINT # statement. Here is an example:

```
PRINT #2,4;12.34,A$,Integer1,Integer2, END
```

Note that you cannot output with a file number and using an image specifier as you can with ASSIGN @... and OUTPUT USING. You also cannot output binary values as with ASSIGN @... and OUTPUTBIN.

For more information about output, see the "Introduction to Output" chapter.

PRINTER IS

BDAT files can also be used as the standard printer as assigned by PRINTER IS. This is a way to output CAT, DBSTATUS, LIST, PARTITION STATUS and XREF listings directly to a mass storage medium. In addition, PRINT and PRINT USING output is sent to the file.

The form of the PRINTER IS statement is:

```
PRINTER IS File_specifier$
```

Only BDAT files can be used because CAT, DBSTATUS, LIST and PARTITION STATUS all output with FORMAT ON.

This type of output can be very useful when you need a permanent record of the system's status.

For more information about PRINTER IS, see the "Introduction to Output" chapter.

Reading Data

As with writing records, there are two methods for reading data from a data file: serially and randomly.

Via I/O Path Names

For I/O paths associated with I/O path names, read data with the ENTER, ENTER USING or ENTERBIN statements.

Serial ENTER

ENTER is the most comonly used of these three statements. Here are some examples of serial ENTER statements.

```
ENTER @Bdat_file;Character$,Integer1,Integer2
ENTER @Io_path;Data1,Data2,Data3
```

The syntax of a serial ENTER is similar to that of a serial OUTPUT. A semicolon separates the I/O path name and the variables into which data items are read. Variables can be separated by commas or semicolons (there is no difference).

ENTER can be executed both with FORMAT ON and FORMAT OFF.

For ASCII, BCD and DATA files, the variable list in the ENTER statement must have numeric or string variables matching the data in the file. That is, you cannot read a number into a string variable or a string into a numeric variable. In addition, string variables must be large enough to hold string data items from the file and numeric variables must be able to hold a value at least as large as the data items in the file. For example, a string variable for twenty characters cannot have a thirty character string read into it. An INTEGER variable cannot contain a numeric item greater than 32 767.

The conversion from a REAL or SHORT to an INTEGER or DOUBLE is simple rounding.

For BDAT files, data is read back as bytes into a variable. Thus, eight INTEGERs (sixteen bytes) can be read back as two REALs (sixteen bytes). This is why care must be taken when reading from BDAT files; it can lead to incorrect values being placed in variables or an ERROR 29.

Random ENTER

ENTER can also be executed with random access. Here is the syntax:

```
    ENTER @Io_path,Record_number;Variable1,Variable2,Variable3$
```

Notice that this form of the statement is quite similar to the serial form. The only difference is the addition of the record number before the semicolon. Data is input to the specified variables from the beginning of the record.

If you attempt to read more data than the record holds, the system returns an error.

ENTER can be executed with FORMAT ON or FORMAT OFF.

Omitting the variable list causes the file pointer to point to the beginning of the specified record but no data is read.

ENTER USING

ENTER USING is the second way to input data. This statement inputs data from a file using a form you specify. ENTER USING can only be used with BDAT files because it is executed with FORMAT ON. The format of the characters is defined by an image specifier, which is discussed in the "Image Specifier" chapter.

ENTER USING can be executed both in serial and in random accessing mode.

ENTERBIN

ENTERBIN is the third way in input data from a file. ENTERBIN retrieves eight bit or sixteen bit binary values from a BDAT file. ENTERBIN is only used with BDAT files because this statement uses FORMAT ON.

Via File Numbers

For I/O paths associated with file numbers, read data with the READ # statement.

Serial READ #

The syntax of the serial READ # statement is similar to that of ENTER.

```
    READ #1;Var1,Var2,Var3$,Var4(*),Var5$(*)
```

A semicolon separates the file number and the variable list. The variables are separated by commas or semicolons (they have no effect). READ # can be used with ASCII, BCD, BDAT and DATA files because the FORMAT is always OFF for READ #.

READ # has the same syntax as the HP 9835/45 READ # statement.

Random READ

You can also use READ # with random access mode. The syntax of the statement is similar to that used for ENTER in random access mode.

```
READ #1,10;X,Y$
```

Note that you cannot input with a file number while using an image specifier as you can with ENTER USING. You also cannot input binary values as with ENTERBIN.

For more information regarding input, see the "Introduction to Input" chapter.

Closing a File I/O Path

Once reading or writing of data has been completed, the file I/O path should be closed. This is done with the ASSIGN @... or ASSIGN # statement.

To close an I/O path associated with an I/O path name, use this form of ASSIGN @...:

```
ASSIGN @Io_path TO *
```

The asterisk signifies that the I/O path "Io_path" is closed.

To close an I/O path associated with a file number, use this form of ASSIGN #:

```
ASSIGN #1 TO *
```

I/O paths should be closed after use to recover memory and to create a more complete program.

Media Errors

In many applications it is critical that the data written to a mass storage device be as accurate as possible. CHECKREAD is a way to significantly increase the reliability of data written to mass storage.

There are two forms of CHECKREAD: the CHECKREAD I/O attribute used with I/O path names and the CHECKREAD statement used with file numbers.

When using CHECKREAD for a particular file, the system checks every item it writes to that file. It does this by immediately following each output operation to the file with a read in the same area of the medium which it has just written (called a "read-afer-write" or "verification" operation). The results of this read are compared with the contents of memory for the data written. If they are identical, the output operation is considered successful. If they are not identical (implying there has been some sort of failure, either on the write or the read-after-write operation), it tries writing and verifying a number of times before sending an error message indicating that a problem exists. The "I/O Resources" section of the BASIC Language Reference lists the number of times each mass storage device write/verifies the data before sending an error message.

The error generated by a CHECKREAD (ERROR 89) causes a program to stop unless trapped by an ON ERROR statement. The ERRL, ERRN, and ERRM$ functions also report this error. More information on these functions and on the ON ERROR statement can be found in the "ON... Conditions" chapter of this manual.

If a CHECKREAD uncovers a bad sector or track on a disc, it can be spared by re-initializing the disc. Simply back up the data on the disc and execute the INITIALIZE statement (discussed in the "Mass Storage Organization" chapter). The system checks the tracks and sectors and spares any which are bad.

CHECKREADing slows output operations significantly. For flexible discs there must be a full revolution of the disc after each operation in order to read in contrast to writing many records per revolution. The resulting decrease in access speeds can be as much as 5-to-1.

In addition to a decrease in operating speed, there can be an increase in wear of the medium itself with flexible discs, in those spots where the heads come into actual contact with the medium.

Note

With flexible discs, because of the combined problems of decreased speeds of operation and increased wear on the media, it is recommended that CHECKREAD be used only when data correctness is a major concern.

The CHECKREAD Attribute

The CHECKREAD attribute of the ASSIGN @... statement institutes output verification on that I/O path.

```
    ASSIGN @Io_path;CHECKREAD ON
```

enables the CHECKREAD for the I/O path Io_path and

```
    ASSIGN @Io_path;CHECKREAD OFF
```

disables the CHECKREAD.

This attribute can be specified at the same time as FORMAT. Thus, this is a perfectly acceptable syntax:

```
    ASSIGN @Io_path;CHECKREAD ON,FORMAT ON
```

The CHECKREAD Statement

The CHECKREAD statement is used in conjunction with file number I/O paths. Here are some examples of enabling CHECKREAD:

```
  .
  .
  .
 80  CHECKREAD ON #1
 90  CHECKREAD ON #2
100  CHECKREAD ON #3
  .
  .
  .
500  FOR I=1 TO 5
510  File$="Data"&VAL$(I)
520  CREATE DATA File$, 100, 80
530  ASSIGN File$ TO #I
540  CHECKREAD ON #I
  .
  .
  .
```

If no file number is included, for example,

```
CHECKREAD ON
```

CHECKREAD is in effect for all operations to every mass storage device.

An enabled CHECKREAD for any file number can be turned off by closing the I/O path or by using the CHECKREAD OFF statement for that file number. Here is an example:

```
CHECKREAD OFF #1
```

Again, if the file number is omitted then executing the statement has the effect of turning off CHECKREAD for all files. If a CHECKREAD ON is executed, the files cannot be turned off individually, but must be turned off with CHECKREAD OFF.

In addition to the verification of data written with a PRINT # statement, by omitting a file number from the CHECKREAD statement, all file operations are verified. After executing a CHECKREAD, statement, all file operations are verified. After executing a CHECKREAD, any SAVE, RE-SAVE, STORE, RE-STORE, STORE BIN, or STORE ALL statement are verified. Directory updates (caused by creating, purging, copying, protecting, renaming etc.) are also checked when the CHECKREAD is in effect. Executing a CHECKREAD OFF statement cancels this.

CHECKREAD also has the effect of causing an immediate-write for a file on a device, flushing the device buffer. For further details on buffers, see the "Advanced I/O Operations" chapter of this manual.

Upon power-on and SCRATCH A, all CHECKREADs are turned off.

Previewing a Data Item

One requirement for any read is that the data types of stored data and the variables into which they are being read correspond. On some occasions, you may not know in advance what the data type of an item is. In such cases, you should find out the data type before doing the read and then select the variable or variables accordingly.

You can do this with the TYP function which is a numeric function that tests the data type of an item without moving the file pointer from that item. Hence, it is possible to use the TYP function to determine the data type of an item and then immediately do a read to get the item itself.

Here are some examples of TYP:

```
TYP(#1,1)
TYP(#9,Test)
TYP(@Disc14,0)
```

The first TYP example shows a file being referred to by its file number, #1. The second 1 in the function is a boolean expression which tells the system whether to skip over EORs or not. 0 means "don't skip" and any other value means "skip".

Thus the data item examined is the item currently being pointed at by the file pointer in file #1. If the item is an EOR, it is ignored and the first item of the next non-empty record is examined.

The second example is similar to the first except that file #9 is being examined and the variable Test is used to determine whether to skip EORs.

The third example shows an I/O path name being used instead of a file number.

The following are the possible values returned by TYP and their meanings:

Value	Data Type
0	An option is missing or a data pointer is lost.
1	Full-precision REAL.
2	Complete String.
3	End-Of-File mark.
4	End-Of-Record mark.
5	Integer-precision number.
6	Short-precision number.
7	Double-precision number.
8	First part of a string.
9	Middle part of a string.
10	Last part of a string.

Number "2", "complete string", means that the system found a complete string. If a string is longer than one record, the system returns "8", "First part of a string", at the first header. When TYP is executed again the system finds the second header for the string. If that portion of the string is also longer than one record, "9", "Middle part of a string", is returned. If the string ended in the next record, the TYP function returns "10", "Last part of a string".

The TYP function is a numeric function returning a numeric result. Therefore, it can be used in numeric expressions the same as may any other numeric function. It is an unusual function, though, in that it requires an access to the I/O system. If you do use it in an expression, and should something go wrong in its attempt to get a value (say, the file was inadvertently not assigned), then you get an error causing the numeric expression to abort.

Since the TYP function requires an access to the I/O system, it cannot be used in output statements, such as PRINT #. This is to prevent a possible "deadlock" situation where the sytem would be trying to read (to fulfill the TYP) and write (to fulfill the PRINT #) at the same time.

Branching on EOR or EOF

It is possible to trap errors caused when a file pointer discovers an EOR or EOF. This is done with the ON END and OFF END statements. This section provides a brief discussion of these statements. For more information, see the "ON Conditions" chapter of this manual.

ON END executes a program branch when an End-Of-Record (random) or any End-Of-File error is returned. The branch can be to another line in the same context via a GOTO, to a subroutine or to a SUB subprogram. It can also execute a line in the same context with the highest priority possible in the computer.

An END condition is caused by an attempt to read or write beyond the end of a record when using random access, by an attempt to read beyond the end of a file, or by an attempt to write when there is no more room in the file and the file cannot be extended.

OFF END disables the ON condition set by ON END. An OFF END # executed without specifying a file number disables ON END conditions for all file numbers.

ON END/OFF END are usable with both I/O path names and file numbers. Here are some examples:

```
    ON END #File_number GOSUB Service_routine
    ON END @File_path_name CALL Serivce_routine
    ON END #1 GOTO 230
    ON END #Fixed_file RECOVER Unexpected_eof
    OFF END #File_number
    OFF END @Path_name
    OFF END #5
    OFF END #
```

Multi-Partition Accessing

It is possible for more than one partition to access a file concurrently. (For more information on partitions, see the "Partitions and Events" chapter). The system schedules the use of a mass storage device so that this can be done. Each partition keeps its own record of the state of the file: for example, its own I/O paths and file pointers. This information is not available to any other partition. This means that although two partitions can both successfully write information into the same file at the same time, one partition may overwrite the data for another partition inadvertantly. To keep partitions from ruining work in progress by another partition, use the LOCK and UNLOCK statements and the IOLOCK function.

The LOCK statement locks out other partitions from use of a file until the LOCKing partition is finished. Its syntax is:

```
LOCK @Io_path
```

Another partition can attempt a LOCK on a file, but its processing is suspended until the file becomes available. If you want your program to attempt a LOCK on a file but you don't want its processing suspended, use this form of LOCK:

```
LOCK @Io_path;CONDITIONAL Status
```

where `Status` is a simple numeric variable or array element which returns the status of the file. If its value is one, the partition successfully LOCKed the file. If its value is zero, the file is currently LOCKED by another partition. Here is a program segment which uses this form of LOCK to gain access to a file as soon as possible.

```
24  LOCK @Bdat_file;CONDITIONAL Status
34  IF Status=0 THEN
44    GOTO 24
54  ELSE
64    GOSUB Output
74  END IF
```

UNLOCK makes a file available for use by other partitions. Once a partition has finished use of a file, UNLOCK should be executed to disable the LOCK. UNLOCK's syntax is:

```
UNLOCK @Io_path
```

A LOCK on a file is also released if any of the following occur in the LOCKing partition.

- The I/O path name is closed explicitly (for example: `ASSIGN @Io_path TO *`).
- The I/O path name is closed implicitly (for example by exiting a subprogram).
- RUN, SHIFT RUN, STOP or SHIFT STOP is executed.
- STOP or END is executed.
- SCRATCH A, SCRATCH C or SCRATCH P is executed.

Trying to UNLOCK a file that has not been LOCKed causes an error.

A partition can LOCK a file more than once. Multiple LOCKs must be UNLOCKed with an equal number of LOCKs. 2 147 483 647 is the maximum number of LOCKs which can be placed on a file.

To determine the LOCK/UNLOCK status of a file, use the IOLOCK function.

The following table summarizes the value returned by IOLOCK.

Value	Meaning
−1	File LOCKed by another partition.
0	File unlocked.
range of DOUBLE>0	File LOCKed N times by this partition.

The IOLOCK value is incremented by each successful LOCK statement specifying the same file. It is decremented by each successful UNLOCK specifying the file. The file remains LOCKed until the IOLOCK value is decremented to zero.

An IOLOCK value of −1 is set to zero when the locking partition executes the last successful UNLOCK statement specifying the file.

Here are some examples of the IOLOCK statement.

```
Lock_count=IOLOCK(@Path_name)
IF IOLOCK(@Name)>0 THEN UNLOCK(@Name)
IF IOLOCK(@Log_file)<0 THEN  GOTO Keep_busy
```

TRANSFER

The TRANSFER statement can be used to transfer unformatted data from a BUFFER to a BDAT file or from a BDAT file to a BUFFER. This can only be done via an I/O path name. For more information about the TRANSFER statement and BUFFERs, see the "Advanced I/O Operations" chapter. Here is the syntax of a simple TRANSFER.

```
TRANSFER @Buffer TO @Bdat_file
```

If this statement is executed, simple data bytes are sent to the BDAT file specified by @Bdat_file from the BUFFER specified by @Buffer.

One of the I/O paths must be to a BUFFER and if the other I/O path is to a file, it must be to a BDAT file.

STATUS, IOSTAT and CONTROL

The STATUS, IOSTAT and CONTROL statements (introduced in the "Advanced I/O Operations" chapter) can be used with ASCII, BCD, BDAT and DATA files which are accessed via I/O path names.

STATUS

STATUS returns the value of one or more I/O registers associated with a file I/O path name. The registers contain values for parameters such as the number of physical records in the file, the length of the file's defined records and the current byte within the record.

Here is an example of using STATUS with an ASCII file.

```
STATUS @Ascii_file,6;Current_byte
```

`@Ascii_file` is the I/O path name of the file you are checking. `6` is the register that you are checking. 6 contains the current byte to be output to within the current physical record. `Current_byte` is a numeric variable which returns the current byte to be output to.

STATUS reads numeric registers sequentially starting with the one specified in the statement. Thus, you can return more than one item at at time by including more variables. For example:

```
STATUS @Bdat_file,1;File_type,Isc,Records,Rlength,Rpointer,Bpointer
```

returns the values of six registers 1 thru 6 because the starting register specified is 1 and there are six variables.

A listing of all the registers available can be found under "ASCII", "BCD", "BDAT" and "DATA" in the "I/O Resources" section of the BASIC Language Reference.

IOSTAT

IOSTAT is a function which returns the value of a single numeric register associated with a file I/O path. This can be very useful when you want to calculate something based upon a numeric status register. For example:

```
Device_selector=SC(@Name)*100+IOSTAT(@Name,3)
```

The registers available are the numeric ones found under "ASCII", "BCD", "BDAT" and "DATA" in the "I/O Resources" section of the BASIC Language Reference.

CONTROL

With the CONTROL statement, you can change the value of one or more status registers. This is very useful for altering a file pointer or restoring the status registers to their original state.

Here is an example of a CONTROL statement.

```
    CONTROL @Data_file;1,1,0,Def_recs,Rec_len,Cur_rec,Cur_byte
```

You can specify a starting register and the data item is then written to it. The succeeding items are written to the following registers.

The HP-IB Interface

Chapter 23

Introduction

This chapter discusses programming techniques unique to the HP 27110A HP-IB interface and introduces several BASIC statements which are used only with that interface. This chapter presumes that you are familiar with the HP-IB addressing methods discussed in the "Introduction to I/O" chapter.

The HP-IB (Hewlett-Packard Interface Bus), or "bus", is Hewlett-Packard's implementation of the Institute of Electrical and Electronic Engineers Standard Digital Interface for Programmable Instrumentation. This chapter discusses techniques involving devices which are already compatible with HP-IB. It does not fully describe the electrical, mechanical and timing specifications of the standard. For information about the standard, consult any of the following documents:

- Institute of Electrical and Electronic Engineers IEEE Std 488-1978
- International Electrotechnical Commission IEC Pub 625-1
- American National Standards Institute ANSI MC1.1

For a detailed understanding of your HP 27110A interface, you should also have available the HP 27132A Technical Reference Package. This package consists of one reference manual for each of the HP-CIO interface cards, including the HP 27110A HP-IB interface. This package is not automatically shipped with each interface card; it must be ordered separately.

A Brief History of the Bus

Prior to the advent of HP-IB, instruments and peripherals were connected to computers with dedicated (unique) connectors, cables, interface cards and software drivers. Dissimilar devices could not share the same interface card. In complex systems, many I/O slots were required and a large portion of system memory was consumed by software drivers.

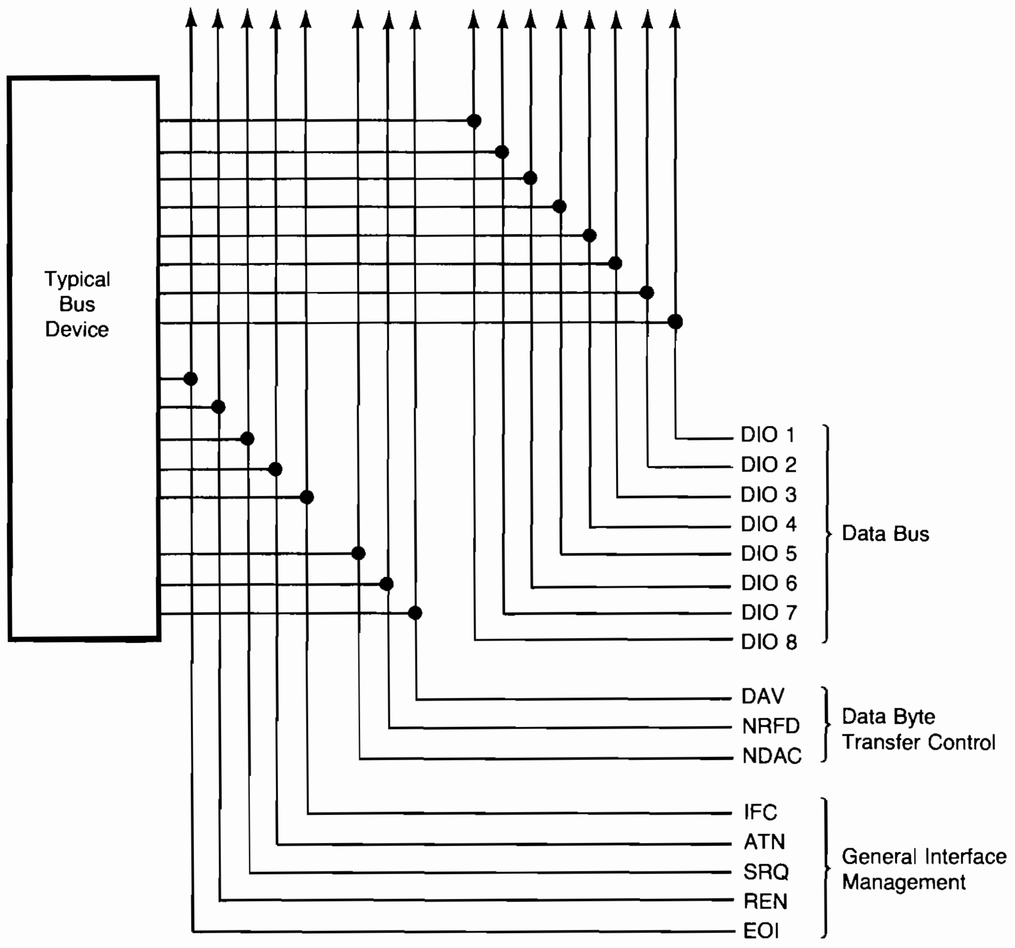
In 1972, in order to standardize HP instrument interfacing, HP introduced the "ASCII Bus". At the same time, HP began participating in various efforts to develop a national and international interface standard. During the next two years, several refinements were made to the ASCII Bus, including a name change in 1974 to "HP-IB". In 1975, the bus definition was adopted by the IEEE as standard 488-1975. The standard was revised in 1978 to clarify potentially ambiguous statements, broaden certain specifications and is now known as standard 488-1978.

Bus Compatibility

Since the publication of IEEE Std 488-1975, hundreds of instruments and computer peripherals have been manufactured that are ostensibly compatible with the IEEE specifications. Vendors other than HP frequently refer to the interface by its IEEE designation or by the expression "General Purpose Interface Bus" (GP-IB). You should be aware that compliance with the IEEE standard is voluntary. HP cannot guarantee that a "GP-IB" or "IEEE-488" compatible device not manufactured by HP will function correctly with your HP 27110A interface card.

General Structure of the HP-IB

The bus transfers data and commands between bus devices on 16 signal lines. This is a **parallel** bus; each device is connected to each line of the bus. The bus lines are shown in the following diagram:



Bus Devices

Any bus device can have one, two or three capabilities: talker, listener and controller.

Talkers are devices which send information on the bus when addressed to talk. Only one device is allowed to talk at one time.

Listeners are devices which accept information from the bus when addressed to listen. One or more devices can listen at any one time.

Controllers are devices which can **reconfigure** the bus by specifying which device is the current talker and which devices are the current listeners.

Only one device is allowed to be the **active controller** (AC) at one time. The active controller can reconfigure the bus and send other bus messages denied to listeners and talkers. You can change the active controller with the `PASS_CONTROL` statement. In some literature the active controller is referred to as the Controller In Charge (CIC).

One bus device is the default active controller at power-up and can become the active controller at any time by sending a special bus message. This is the **system controller** (SC). This capability is generally determined by the design of the device's HP-IB interface or, as is the case with the HP 27110A interface, by a switch on the interface card. You cannot pass system control of a bus.

Each HP 27110A interface card in your computer represents a separate bus. The status of each interface (talker, listener, controller) is independent of the state of the other buses.

Each device (talker, listener, controller) on the bus has one or more primary device addresses. A device address is an integer in the range 0 through 30. This means that a single bus can support 31 addressable or "logical" devices. The number of actual devices is usually less due to the bus cabling rules discussed in the "Bus Cabling" section.

Capability Codes

Each HP-IB compatible device supports some combination of the functions specified in the IEEE 488-1978 Standard. Each capability can be identified by a one or two letter abbreviation followed by a one or two digit number. For example, the codes for your HP 27110A interface are: AH1, C1 through C5, DC1, DT1, L3, LE3, PP1, RL1, SH1, SR1, T5 and TE5.

For many devices, the list of capability codes appears on the device near the HP-IB connector. In general, an omitted code letter or a code number of zero implies no capability for that function.

The BASIC Language Reference manual lists the capability codes required for support of each HP-IB statement. For a complete description of each code, refer to Appendix C of the IEEE Standard.

Bus Message Lines

Eight of the bus lines are used only for the transfer of bus **messages**. A bus message can consist of one or more data or command bytes. A typical data message is a line of text sent by the `OUTPUT` statement. A typical command message is the talker/listener addressing sequence sent by the `OUTPUT` statement prior to the text data.

Bus messages are sent on the Data Input/Output (DIO) lines DIO 1 through DIO 8. Messages are sent on these lines in a bit-parallel, byte-serial manner. Message transfer is asynchronous; that is, an arbitrary amount of time can elapse between the sending of one byte and the next.

Data Byte Transfer Control

Three of the bus lines are used to coordinate the sending of bus messages on the DIO lines. Listener (message accepting) devices use the Not Ready For Data (NRFD) to assure that the talker (message sending) device does not send data before the slowest listener is ready. The talker uses the DAta Valid (DAV) to alert the listener(s) that a message byte is present on the DIO lines. Finally, the listeners use the Not Data ACcepted (NDAC) line to alert the talker that they have accepted (read) the message on the DIO lines.

This "three-wire handshake" using the NRFD, DAV and NDAC lines allows data transfer between one talker and one or more listeners. The listeners need not be able to accept data at the same rate. The speed of the data transfer is regulated by the slowest device (listener or talker) involved in the transfer. Devices not addressed as listeners or talkers do not participate in the three-wire handshake and do not affect the data rate.

You can set some bus devices to configurations known as "listen always", and "talk always". This is usually accomplished with a switch. Such devices always participate in bus data transfer, whether addressed or not. The HP 27110A interface always participates in the handshake of messages it outputs to the bus. This means that you can send bus messages even if no devices are connected to the bus.

No BASIC statements provide direct control of the transfer control lines, although you can test the state of the NDAC line with a `STATUS` statement specifying `HP27110` register 304.

General Interface Management

Five bus lines are used for control of the bus. Two of these lines (IFC and REN) are reserved for use by the device designated as the **system controller**. One line (ATN) is reserved for use by the **active controller**. The remaining two lines (EOI and SRQ) can be asserted by any bus device.

When the controller or devices assert (set) one of these lines to the TRUE (1) or FALSE (0) state, this is known as sending a uni-line message.

Commands vs Data - ATN

ATN means "attention". When the ATN line is TRUE, each device pays attention to the information on the other bus lines, regardless of whether or not the device is addressed as a talker or listener. This line has two purposes:

- When the ATN line is FALSE, the bytes sent on the DIO lines are treated by bus devices as **data**. When the ATN line is TRUE, the bytes sent on the DIO lines are treated by bus devices as bus **commands**. For example, if the ASCII DEL character (code 127) is sent to an HP 2631B printer with ATN FALSE, it is printed as the "▒" symbol. If it sent to the printer with ATN TRUE, it is not printed; instead, the printer returns self-test status to the computer. Only the active controller can assert ATN TRUE.
- The active controller can also use the ATN line, in conjunction with the EOI line, to conduct a bus operation known as a parallel poll. The `PPOLL` statement asserts both ATN and EOI TRUE for a brief period of time without sending bytes on the DIO lines.

Many BASIC statements change the state of the ATN line during their bus sequences. The state of the line is noted in the HP-IB messages table for each HP-IB statement in the BASIC Language Reference manual. You can explicitly control the ATN line with the HP-IB `SEND` statement.

Documentation for early "ASCII Bus" devices may refer to the ATN line as the Multiple Response Enable (MRE) line.

End of Data - EOI
EOI means "end or identify". This line has two purposes:

- When ATN is FALSE, the current talker (controller or device) can assert EOI TRUE to denote that the current data byte on the DIO lines is the last byte in a sequence of bytes. EOI is provided so that devices can send any binary byte value on the DIO lines. No states of the DIO lines are reserved, although many devices use the ASCII linefeed (LF, `CHR$(10)`) character to terminate a message sequence.
- The active controller can use EOI in conjunction with ATN to conduct a parallel poll, as mentioned in the discussion of ATN.

You can assert EOI TRUE in many BASIC output statements and in the `SEND` statement by specifying the `END` keyword. You conduct a parallel poll with the `PPOLL` statement.

Very early "ASCII Bus" devices may not support the EOI line. These devices do require a reserved DIO byte (usually linefeed) to delimit DIO message sequences and they cannot respond to parallel poll.

Resetting the Bus - IFC
IFC means "interface clear". This line has two purposes:

- It clears the current bus configuration. When the system controller asserts this line to the TRUE state, all bus activity ceases. Devices addressed as listeners stop accepting data. The device addressed as the talker stops sending data. The bus devices should not otherwise reset.
- It returns active control of the bus to the system controller. If a device other than the system controller is currently the active controller, it gives up active control of the bus and becomes a simple (talk/listen) device.

There are two statements which affect the IFC line.

- If the interface is system controller, the `RESET` statement causes it to assert IFC TRUE, taking control of the bus. Regardless of whether or not the interface is system controller, `RESET` has other effects which are discussed in the I/O Resources section of the BASIC Language Reference manual.
- If the card is system controller, the `ABORT` statement causes the interface to assert IFC TRUE, taking control of the bus. If the interface is not system controller, `ABORT` does not assert IFC. Instead it sends bus commands which cause all devices to cease listening and cease talking.

Documentation for early "ASCII Bus" devices may refer to the IFC line as the End Output (EOP) line.

Note

Some bus devices, typically test instruments, momentarily assert IFC true at power-up.

Front-panel Control - REN

REN means "remote enable". This line has a single function; it controls the state of the operator controls on many HP-IB test instruments.

When REN is TRUE, listening bus instruments accept programming instructions only from the bus. The instruments are said to be in the REMOTE state. Those buttons and switches on the instruments which represent bus programmable functions are disabled. Controls which do not represent programmable functions may be enabled.

When REN is FALSE, instrument controls are enabled. The instruments are said to be in the LOCAL state.

REN is generally ignored by computer peripherals such as disc drives and plotters. Refer to the device's reference manual for information on the device's response to the REN line.

If the interface is the system controller, there are four conditions which affect the REN line:

- REN is set FALSE at power up.
- The `REMOTE` statement sets REN TRUE.
- If you specify the interface select code of the bus (not a primary device address), the `LOCAL` statement sets REN FALSE.
- If you specify the interface select code of the bus (not a primary device address), the `ABORT` statement sets REN FALSE.

Bus Device Interrupts - SRQ

SRQ means "service request". This line has a single purpose; to provide a method for a bus device to alert the active controller that the device requires service. The meaning of "service" depends on the device and the application.

Any bus device (or combination of devices) can assert SRQ TRUE at any time. The normal method for responding to SRQ is to enable the HP 27110A interface to generate a computer interrupt when it detects SRQ TRUE.

Each requesting device asserts SRQ FALSE when subsequently serially polled (SPOLL) or when the reason for the request goes away.

The HP 27110A interface can also assert SRQ. This is only possible if the interface is acting as a device, rather than as active controller.

Interface Installation

If you have not yet installed your HP-IB interface card, first consult the HP 27110A Installation Manual, part number 27110-90001. This manual covers procedures for installing the HP-IB card in any computer. It does not include information related to the use of the card in the HP 9000 Model 520 BASIC Language System; that is the purpose of this section.

CAUTION

STATIC SENSITIVE DEVICES

The ROM, RAM and Z-8 components used in the HP 27110A are susceptible to damage by static discharge. Refer to the SAFETY considerations at the front of the HP 27132A manual before handling the interface card.

Before installing the card, you need to check its configuration. There are eight switches (S1 through S8) located on the edge of the HP-IB card. Depending on the position of switch S7, you may also need to move a component on the card.

Interface Bus Address - Switches S1..S5

In addition to being bus controller, your HP-IB interface is also a bus device. Like all bus devices, it must have a primary device address in the range 0 through 30. The device address is not the same as the card's interface select code, which is determined by the I/O slot in which you install the card.

You establish the card's default device address with switches S1 through S5. If there are no other controllers on the bus connected to this interface, you should use an address of 30. The HP 27110A interface is shipped with switches S1...S5 set to this address.

Examples of devices capable of acting as bus controller are other computers and calculators. Few computer peripherals and test instruments can act as bus controllers. Those which can generally have an HP-IB capability code in the range C1 through C28. C0 devices cannot act as controllers. This code may appear on the device near its HP-IB connector.

The HP 27110A uses the address set in switches S1...S5 only when the card is acting as a simple talker or listener device. Whenever the card is active controller, it assumes a bus address of 30, regardless of the switch settings. The card also only uses the switch set address as a default value, read at power-up and after `RESET`. You can change the default device address with a `CONTROL` statement specifying register 302.

If the bus to which you are connecting the card has more than one device capable of acting as controller, you may need to select a device address other than 30. If the other controller(s) is (are) HP 27110A interface cards, you must select an address other than 30; in fact, none of the 27110's on the bus should have an address of 30 since the 27110 currently acting as controller assumes an address of 30.

The following table shows the switch settings for addresses 0 through 30. Address 31 is illegal as a device address; it is used in bus addressing messages as the unlisten (UNL) or untalk (UNT) command. "U" is switch up and "D" is switch down.

Device Address	**Switch Settings (S)**				
	1	**2**	**3**	**4**	**5**
0	D	D	D	D	D
1	U	D	D	D	D
2	D	U	D	D	D
3	U	U	D	D	D
4	D	D	U	D	D
5	U	D	U	D	D
6	D	U	U	D	D
7	U	U	U	D	D
8	D	D	D	U	D
9	U	D	D	U	D
10	D	U	D	U	D
11	U	U	D	U	D
12	D	D	U	U	D
13	U	D	U	U	D
14	D	U	U	U	D
15	U	U	U	U	D

Device Address	**Switch Settings (S)**				
	1	**2**	**3**	**4**	**5**
16	D	D	D	D	U
17	U	D	D	D	U
18	D	U	D	D	U
19	U	U	D	D	U
20	D	D	U	D	U
21	U	D	U	D	U
22	D	U	U	D	U
23	U	U	U	D	U
24	D	D	D	U	U
25	U	D	D	U	U
26	D	U	D	U	U
27	U	U	D	U	U
28	D	D	U	U	U
29	U	D	U	U	U
30	D	U	U	U	U
Illegal	U	U	U	U	U

Controller Status-Switch S6

To make the interface **system controller** of the bus, set switch 6 to the UP position. If the interface is the only controller on the bus, it must be system controller or the bus is unusable. If there is more than one controller on the bus, you must decide which is to be system controller.

If another controller is to be system controller, set switch S6 to the DOWN position. With S6 down, the interface is a simple talker/listener device. It cannot send or accept bus data unless addressed by the active controller. It can become the active controller if the current active controller passes control to it.

Data Transfer Rate - Switch S7

One of the enhancements made when the IEEE 488-1975 standard became the IEEE 488-1978 standard involved the maximum potential data transfer rate. To increase the maximum rate from 500 000 to 1 000 000 bytes per second, the 1978 standard allows the operation of the bus in "high speed mode".

High speed mode is provided primarly for buses connected to mass storage devices, such as disc drives. You should use this mode whenever the bus conforms to the following requirements:

- Switch S7 is set to the DOWN (high speed) position and the load resistor pack is installed.
- All of the devices connected to the bus must be compatible with high speed operation. Contact your your local Hewlett-Packard Sales and Service office for information concerning the rate mode compatibility of your HP-IB devices.
- The total length of interconnecting cable must be less than 15 metres and less than or equal to one metre per device load. Device loads are discussed in the "Bus Cabling Limits" section of this chapter.

For use with instruments and computer peripherals other than mass storage devices, you normally use the interface in standard speed mode. This supports data transfer rates up to 500 000 bytes per second with less restrictive cabling rules. You can use most high speed compatible devices on a standard speed bus. To select standard speed mode, set switch S7 to the UP position. The 27110A cannot be configured for low speed operation.

Interface Load Resistor

The use of high speed mode requires that a dual-in-line resistor package (part number 1810-0081) be installed in the socket directly behind the cable connector (J2). Pin 1 on the package must be oriented towards the "U82" marking next to the socket. If this resistor pack has not previously been installed in this socket, it should be found in a nearby (unconnected) socket labelled "LOAD RESISTOR STORAGE".

CAUTION

The pins of the resistor pack can be bent or broken unless the package is handled carefully. You may want to have an electronic technician perform this operation. If you have arranged for an HP Customer Engineer (CE) to install the system or interface, the CE can relocate the resistor pack for you.

The resistor pack can also be installed for standard speed mode and permits the use of longer cables. This technique is discussed in the "Bus Cabling Limits" section of this chapter.

Self Test Mode - Switch S8

This switch should be in the DOWN position.

Bus Systems

A single bus consists of the HP 27110A interface and its cable connected to at least one HP-IB device. The cable supplied with the interface card is intended for connection to a single device. The HP-IB connector on the device end of the cable is a dual connector. The male side of the connector plugs into the female HP-IB connector of the device.

The cable is secured to the device with the floating lock screws. The lock screws are designed to be tightened with the fingers only. Do not use a screwdriver; the screwdriver slots in the lock screws are provided for removal only.

CAUTION

All current HP-IB devices and cables accept and use lock screws with Metric ISO M3.5x0.6 threads. Metric lock screws have a black finish and may be stamped "M". Some HP-IB or "ASCII Bus" devices and cables manufactured prior to 1975 have lock screws with English 6-32 UNC threads. These lock screws have a bright (nickel) finish and cannot mate to metric hardware without thread damage. A conversion kit (HP part number 5060-0138) is available to convert one cable or two devices from English to metric hardware.

Adding Devices to the Bus

Additional devices are added to the same bus by connecting one or more HP-IB cables to the HP-IB connector on the first device. Although you can "stack" HP-IB connectors without apparent limit, you should avoid connecting more than three cables to the same device. This is to prevent damage to the mounting panel of the device should tension be applied to the outer cable.

The normal method for adding devices to a bus is to chain them; the cable for each device is connected to the previous device. There are no limits on the topology of the bus cabling, only on the total cable length. For example, you can also connect devices in a star configuration.

Device Addresses

Each device connected to a single bus normally has a unique primary device address in the range 0 through 30. The address is usually set on a series of switches on the rear panel of the device. Some devices, such as the HP 7970E Magnetic Tape Drive, require removal of an access panel. Others, such as the HP 98034A/B interface used in the HP 9825/35/45 computers, must be disassembled.

Each device capable of talking must be set to a unique address. If two devices attempt to talk at the same time, a bus error or data loss usually results. Only devices with capability codes T0 and TE0 cannot talk. You can inhibit the talk capability of devices which have a "Listen Only" control.

Devices capable of listening can have unique addresses or the same address as another listener on the bus. Since most BASIC I/O statements can specify multiple devices on the same bus, duplicate device addresses are generally not necessary.

When selecting an address for a device, keep the following in mind:

- Do not use address 31. Although you can set this value on devices which have five address switches, it is illegal since it is used as the universal unlisten and untalk commands when sent as an address in a bus command.
- Avoid the use of address 30. This address is used by the HP 27110A interface whenever it is active controller, regardless of how its device address switches are set.
- You can set devices with three address switches (restricted address devices) only to addresses in the range zero through seven. These devices are typically discs, mag tapes and high speed printers. Their drivers assume that they respond to parallel poll by asserting DIO line = 8-Address. If you place other devices on the same bus, you should restrict their addresses to the range zero through seven and never configure them for parallel poll response or they might attempt to assert the same line as one of the restricted address devices.

"Always" and "Only" Configurations

Some devices have switches that configure them for "Listen Always", "Talk Always", "Listen Only" or "Talk Only" operation. A device configured for "Always" operation acts as a listener or talker regardless of whether it is so addressed by the controller. A device configured for "Only" operation cannot talk if it is configured for listen-only, even if addressed as a talker, and vice-versa.

Here are two examples of these configurations. You can set multiple HP 9872C Plotters on the same bus to the same address for obtaining mutiple plots. Set all but one of them to "Listen Only". All plot the data sent to the bus address, but only one can return status and digitizing data to the computer. You can set an HP 9875A Cartridge Tape Unit to "Listen Always" to log all data sent on a bus. You can also set it to "Talk Always" to send recorded data on a bus. The drive "talks" even though there is no controller on the bus to address it.

When the HP 27110A interface is acting as a device (not active controller), you can configure it for listen always or talk always by setting bit 5 or 6, respectively, in control register 302.

You must be cautious with these capabilities because they are not strictly in conformance with the IEEE standard. Even the `ABORT` statement, which asserts IFC, cannot unlisten a device set for "listen always".

HP-IB Cables

Cables for adding devices to a bus are available from Hewlett-Packard in a variety of lengths. Here are the recommended cables for use with your HP 27110A interface:

HP Product or Part Number	Length in Metres	Comments
8120-4310	2.0	HP 27110A interface-to-HP-IB cable
92220R 10833D 10833A 10883B 10833C	0.3 0.5 1.0 2.0 4.0	Has right-angle connector on one end. Early versions of these low-EMI cables may require a 10834A adaptor when used with some HP-IB instruments.
5060-9459 5060-9460	6.0 8.0	Carefully compute and observe maximum system cable length before using.

In order for the electromagnetic interference (EMI) of your system to remain below the FCC and VDE limits to which it was tested, you should use the above cables. The prior 10631A/B/C/D cables are not shielded in the same fashion and do not provide the same level of EMI protection. A bus incorporating these older cables (or non-HP cables) may function correctly, but the continuity of the 2 in 1 shielding of the interface cable is broken.

Early versions of the 10833 cables have a connector housing with a diamond shaped strain relief. This housing prevents the connector from mating properly with some test instruments which have limited space on their rear panels. In such cases, you can use the 10834A adaptor to extend the instrument's HP-IB connector.

Bus Cabling Limits

To obtain low noise and signal distortion on the bus and insure reliable data transfer, the total length of cable for a single bus is limited to 15 or 20 metres. The length may be limited to a smaller figure depending on the number of actual or simulated devices on the bus. The rule used to determine the limits for any bus also depends on whether the bus is used in standard or high speed mode, as summarized in the following table.

Cabling considerations	Standard Speed	High Speed
Absolute total length, metres	20	15
Metres per device load	2	1
Maximum number of device loads	15	15

Each bus device has an active resistive load and a capacitive load on the bus. Of the two, the resistive load is the more important. For either a high or low speed bus, the maximum number of equivalent standard loads is 15.

Test instruments and computer peripherals other than mass storage devices generally represent one standard load. This means that you can place up to 14 instruments on a standard speed bus. Mass storage devices sometimes represent two loads. HP-IB interface cards can represent a varying number of loads. The HP 27110A interface can place one or eight loads on the bus. Consult your local HP Sales Office for information regarding the number of loads represented by an HP-IB device.

Because the seven-load resistor pack is normally installed on the interface card for high speed operation, this limits the additional allowed loads to seven. If the interface is connected to high speed discs which represent two loads, the bus is limited to a maximum of three disc drives.

You can also use the resistor pack in medium speed mode to increase cable lengths. Suppose that you want to place an HP 7585B Plotter at a distance of eight metres from the computer. Without the resistor pack installed, the interface plus plotter represent only two loads, limiting the cable length to four metres. With the resistor pack installed, there are nine loads, and you can use up to 18 metres of cable.

As long as the total cable length is less than the limit, there is no restriction on the amount of cable between any two devices on the bus. If you have a group of 14 HP-IB instruments connected to each other by 0.5 metre cables, the entire "cluster" can be 13 metres from the computer.

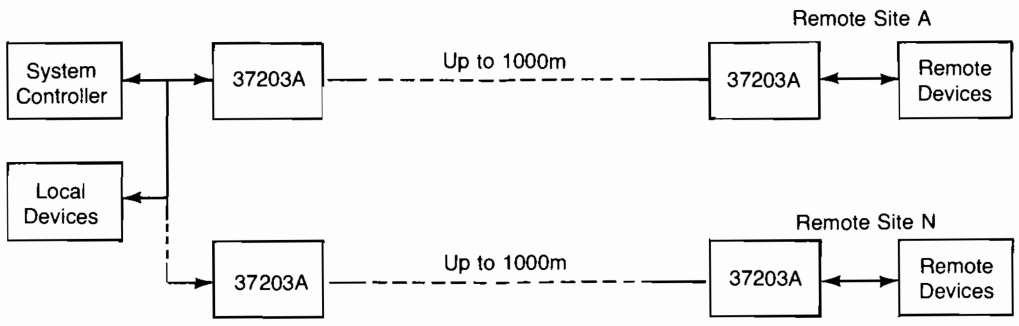
Devices Powered On and Off

For proper bus operation at medium speed, the devices representing at least four of every five loads should be powered on. At high speed, all connected devices must be powered on. In particular, if the load resistor is installed on the interface, no handshake can take place between other devices if the interface is powered off.

When connecting a device to an operating bus, first power-on the new device. If you connect a powered off device to the bus, the handshake may stop until the new device is powered on; also, some devices improperly affect the state of bus lines during power-on and can cause errors or "hang" bus operations.

Bus Extenders

If you need to locate a bus device more than 20 metres from the interface, use a pair of bus extenders. Bus extenders convert the electrical activity on the 16 parallel bus lines to a single stream of serial data bits on a single line or pair of lines. The following illustration shows a bus system using a pair of HP 37203A extenders.



When you use a pair of bus extenders, the bus cabling directly connected to the controller (interface) is known as the **local bus**. The cabling connected to the distant extender is known as the **remote bus**. This use of the terms "local" and "remote" has no connection with the local and remote modes of an instrument.

When you extend a bus, you not only increase the maximum distance between the interface and HP-IB devices, you also increase the maximum number of devices that you can connect to the bus. The local and remote buses are not directly interconnected and each can support 15 loads. Each extender represents one load on the bus to which it is connected.

HP Bus Extenders Available

Hewlett-Packard presently offers two bus extenders, the HP 37201A and HP 37203A. The extender you should use depends on several considerations, such as distance, existing cables, data rate and bus functions required. The two models are summarized in the following table.

Extender Feature	**37201A**	**37203A**
Extender-to-extender connections available	Dual twisted pair cable or RE-232-C/V.25 datacomm	Coaxial or fibre optic cable
Maximum distance between extenders	1000 metres cabled, unlimited datacomm	1000 metres
Maximum data transfer rate	775 bytes per second	50K bytes per second
Data transfer rate mode	Standard speed	Standard speed
Extender device load	1	1
Extender device addresses	1	0
Extender programmable as a device	Yes	No
Pass control to remote device	No	Yes
Conduct parallel poll of remote devices	No	Yes, but see PPOLL section

Bus Mnemonics and Messages

Each bus operation is the transfer of one or more bus **messages**. A message involves the assertion of one or more bus lines to the TRUE or FALSE state. Each message is identified by a three or four letter abbreviation called a mnemonic. Some messages are sequences of other messages.

The messages sent by each HP-IB statement are summarized in a table accompanying each statement in the BASIC Language Reference manual. The messages sent by other I/O statements are summarized under the entries "bus reconfiguration" and "direct bus I/O" in the glossary of that manual. Each mnemonic is defined in the "HP-IB Mnemonics" table in the "Useful Tables" section of the reference manual.

The "General Bus Structure" section of this chapter introduced the five General Interface Management lines (ATN, EOI, IFC, REN and SRQ) and discussed the effect these lines have on bus operations. When a BASIC statement causes the interface card to assert one of these lines, this is called a uni-line message. The bus operations discussed in this section involve the use of more than one bus line and are called multi-line messages.

The multi-line bus messages are the data messages and the command messages. **Data** messages consist of data bytes sent on the DIO lines with the ATN line FALSE. **Command** messages consist of data bytes sent on the DIO lines with the ATN line TRUE. You can send and receive data messages only when the interface is configured as a talker or listener (respectively). You can send command messages only when the interface is active controller. You can receive command messages only when the interface is acting as a device and is not active controller.

The meaning of a command message depends on the value of the data byte. A data byte can have a value in the range 0 through 255 (00000000 through 11111111 binary). Only values in the range 0 through 127 are used. The most significant bit (DIO line 8) is reserved for use as a parity bit. The HP 27110A interface sends commands with odd parity.

Command bytes in the range 0 through 95 are called **primary commands**; bytes in the range 96 through 127 are called **secondary commands**.

Primary Bus Commands

Command bytes in the range 0 through 15 (ASCII characters NUL through SI) are called the **addressed** commands. These commands cause an action only in the device(s) currently addressed as listeners. Only five of the 16 possible byte values in this range are defined as commands (GTL, SDC, PPC, GET and TCT). Commands in this range are sent by the `LOCAL`, `CLEAR`, `PARALLEL POLL CONFIGURE`, `TRIGGER` and `PASS CONTROL` statements.

Command bytes in the range 16 through 31 are the **universal** commands. These commands affect all devices on the bus. Only five of the 16 values in this range are defined as commands (LLO, DCL, PPU, SPE and SPD). Commands in this range are sent by the `LOCAL LOCKOUT`, `CLEAR`, `PARALLEL POLL UNCONFIGURE` and `SPOLL` statements.

Command bytes in the range 32 through 63 are the **listen addresses**. A single listen address is mnemonic LA*; a sequence of listen addresses is called a listen address group (LAG). The computer sends a listen address on the bus by sending the specified device's primary address with bit 5 (DIO line 6) true and bit 6 (DIO line 7) false in the binary representation of that address. In effect, the computer adds 32 to the primary address. When sent as a command, this value causes the specified device to become a listener on the bus. Commands in this range are sent by all I/O statements which reconfigure the bus.

Listen address 31 (command 63) is not a device address. This value is reserved for use as the **unlisten** (UNL) command. UNL causes all current listener devices to cease listening. UNL is sent by all statements which reconfigure the bus, to insure that only the specified devices participate in the new operation.

Command bytes in the range 64 through 95 are the **talk addresses**. A single talk address is mnemonic TA*; a sequence of talk addresses is a talk address group (TAG). The computer sends a talk address on the bus by sending the specified device's primary address with bit 5 (DIO line 6) false and bit 6 (DIO line 7) true in the binary representation of that address. In effect, the computer adds 64 to the primary address. When sent as a command, this value causes the specified device to become a talker on the bus. Commands in this range are automatically sent by all I/O statements which perform bus reconfiguration.

Talk address 31 (command 95) is not a device address. This value is reserved for use as the **untalk** (UNT) command. UNT causes the current talker device to cease talking. UNT is generally not required in bus operations because the current talker device should cease talking if it detects that the active controller has sent any new talk address.

Note

All of the mnemonics in this section denote bus **commands**, which implies that ATN is TRUE. Nonetheless, the HP-IB messages table for each HP-IB statement in the BASIC Language Reference manual shows the state of ATN separately. This is so that you can determine whether ATN is left in the TRUE state at the end of the command sequence.

Ignoring the parity bit, the listen, unlisten, talk and untalk commands are often represented by displayable ASCII characters in the documentation for many HP-IB and IEEE 488-1978 compatible devices. You can correlate character and numeric device addresses by using the "BASIC Character Codes" table in the BASIC Language Reference manual. As an example, the following statements send identical bus messages. Correct parity is added by the SEND statement in each case.

```
100  OUTPUT 425;                          !Output no data
110  SEND 4;MTA;UNL;LISTEN 25
120  SEND 4;TALK 30;LISTEN 31;LISTEN 25
130  SEND 4;CMD 94,63,57
140  SEND 4;CMD "^","?","9"
```

Secondary Commands

The command byte values in the range 96 thru 127 are the secondary commands. Note that value 127 is considered to be undefined. Secondary commands can be sent in a group and are always preceded by a primary command. This is because the meaning of the secondary command depends on the value of the primary command.

The only secondary commands defined by the IEEE 488-1978 standard are those related to parallel polling. Primary command value 5 (PPC) followed by secondary commands in the range 96 thru 103 (PPE) configures a device to respond to parallel poll. PPC followed by secondary command value 112 (PPD) disables a device's parallel poll response. See the "Parallel Polling" section. Other secondary command sequences are device-defined.

Secondaries can be automatically sent by all I/O statements which allow device selectors with secondary addresses. You can also explicitly send a secondary with the `SEND... ;SEC...` statement.

Secondary commands are not generally referred to as command values 96 through 126. They are referred to as secondaries 0 through 30. BASIC I/O statements and `SEND... ;SEC...` automatically set bits 5 and 6 (DIO lines 6 and 7) TRUE in the value specified. Effectively, the system adds 96 to secondaries in the range 0 through 30.

Devices which accept secondaries after a primary listen address are called **extended listeners** (capability code LE). Those which accept secondaries after a primary talk address are called **extended talkers** (capability code TE).

Bus Operations

The rest of this chapter deals with the use of a bus in the BASIC Language System. It introduces the HP-IB statements by discussing the application areas in which they are used. The areas are broadly classified as follows:

- interface and bus status;
- resetting the bus and devices;
- reconfiguring the bus;
- bus I/O considerations;
- instrument programming;
- bus interrupts;
- device status.

Interface and Bus Status

Before beginning bus I/O in a program, you should know the state of the bus. It is particularly important to know if the interface card is active controller or system controller, because only the active controller can perform many HP-IB operations. This section shows how to read bus status and determine if any devices are present on the bus. An alternative method is to force the bus and devices to a known state. This is described in the next section.

Interface Status

There are nine status registers which can tell you a great deal about the bus. The following program prints a report about any HP 27110A HP-IB interface and demonstrates access to these registers. It does not detail the contents of the interrupt registers; these are discussed in the "Bus Interrupts" section.

```
 10 !BUS_STAT: Program to read HP27110A status [RJN] <830111.1306>
 20  LOOP
 30    INPUT "BUS_STAT: Enter select code of HP27110A (or 0)",Bus
 40  EXIT IF NOT Bus
 50    IF (Bus<0) OR (Bus>23) THEN  GOTO Retry
 60 !
 70    ASSIGN @Bus TO Bus;DRIVER "HP27110",RETURN Error
 80    IF Error THEN
 90      PRINT "BUS_STAT: ASSIGN@... ERROR ";Error
100      PRINT "          Select code ";Bus;" is not an HP27110A."
110      GOTO Retry
120    END IF
130 !
140    PRINT LIN(1);"BUS_STAT: HP27110A at select code ";Bus
150    STATUS @Bus,300;R300,R301,R302,R303,R304
160    STATUS @Bus,306;R306,R307,R308
170 !
180    PRINT LIN(1);"Register 300 = ";DVAL$(R300,16)
190    Address=BINAND(R300,31)
200    Sc$=Ac$=" is not "
210    IF BIT(R300,6) THEN Ac$=" is "
220    IF BIT(R300,7) THEN Sc$=" is "
230    PRINT "Controller device address ";Address
240    PRINT "This interface";Sc$;"the system controller."
250    PRINT "This interface";Ac$;"the active controller."
260 !
```

```
270    PRINT LIN(1);"Register 301 = ";DVAL$(R301,16)
280 !
290    PRINT LIN(1);"Register 302 = ";DVAL$(R302,16)
300 !
310    PRINT LIN(1);"Register 303 = ";DVAL$(R303,16)
320    Address=BINAND(R303,31)
330    PRINT "Talker/listener address ";Address
340    IF BIT(R303,6) THEN PRINT "Interface is active controller."
350    IF BIT(R303,7) THEN PRINT "Interface is system controller."
360    IF BIT(R303,9) THEN PRINT "Interface is addressed to talk."
370    IF BIT(R303,10) THEN PRINT "Interface is addressed to listen."
380    IF BIT(R303,15) THEN PRINT "Interface is in REMOTE mode."
390 !
400    PRINT LIN(1);"Register 304 = ";DVAL$(R304,16)
410    Ndac$=Srq$="FALSE."
420    IF BIT(R304,10) THEN Srq$="TRUE."
430    IF BIT(R304,13) THEN Ndac$="TRUE."
440    PRINT "The SRQ line is "&Srq$
450    PRINT "The NDAC line is "&Ndac$
460 !
470    Tstat$=DVAL$(R306,16)
480    PRINT LIN(1);"Register 306 = ";Tstat$[7,8]
490    IF R306<>0 THEN PRINT "Refer to 27132A reference manual."
500 !
510    PRINT LIN(1);"Register 307 = ";DVAL$(R307,16)
520    IF BIT(R307,7) THEN
530      Address=BINAND(R307,31)
540      PRINT "Last secondary address read = ";Address
550      IF BIT(R307,5) THEN
560        PRINT "Preceding primary was a talk address."
570      ELSE
580        PRINT "Preceding primary was a listen address."
590      END IF
600    END IF
610 !
620    PRINT LIN(1);"Register 308 = ";DVAL$(R308,16)
630    IF R308 THEN PRINT "Outbound FIFO frozen due to inbound data."
640 !
650 Retry: !
660  END LOOP
670 !
680  PRINT "BUS_STAT: END"
690  STOP
700  END
```

Status register 306 is updated by driver HP27110 only after an error or exception condition occurs and not after successful I/O operations. Thus, it always reflects the reason for the most recent I/O error.

See the "Reading Secondary Commands" section of this chapter for information about the use of status register 307.

In the previous example, note that the DRIVER expression in the ASSIGN statement is not required. The system automatically configures driver HP27110 (if present) to each HP 27110A interface in the computer. This expression, and the RETURN expression, are used to confirm that driver HP27110 is present and that the specified interface is an HP 27110A. You can use this technique with any interface.

Bus Status

An examination of interface card registers tells you the state of the interface; it does not tell you whether or not there are any devices on the bus. If you are not certain that a device is present on the bus and is at a given address, you can set a timeout for your I/O operations. However, a timeout can occur even if the device is present; for example, it could be the wrong device. Also, bus commands do not cause timeouts unless a device fails. The HP 27110A interface always "handshakes" bus commands, even if there are no other devices on the bus.

Here is a method for determining what devices with basic listener (HP-IB capability codes L1 through L4) are present on the bus. This technique involves addressing each possible device on the bus as a listener and then asserting ATN false by sending a DATA term with no data values. The addressed listener begins the three-wire data transfer handshake by asserting NDAC (not data accepted) and waiting for the controller to assert DAV (data valid), which never happens. While the bus is in this state, the controller examines the NDAC bit (13) of status register 304. Note that this technique cannot detect the presence of extender listener (LE1 through LE4) devices that have no basic listener capability, such as the HP 82901M/S disc drives.

```
 10 !DEV_CHECK: Program to test for devices     [RJN] <821228,1312>
 20 !
 30 !RE-SAVE "DEV_CHECK:INTERNAL"
 40 !
 50 Retry: !
 60  INPUT "DEV_CHECK: Enter select code of HP27110A (or 0)",Bus
 70  IF NOT Bus THEN  GOTO Stop
 80  IF (Bus<0) OR (Bus>23) THEN  GOTO Retry
 90 !
100  ASSIGN @Bus TO Bus;DRIVER "HP27110",RETURN Error
110  IF Error THEN
120    PRINT "BUS_STAT: ASSIGN@... ERROR ";Error
130    PRINT "           Select code ";Bus;" is not an HP27110A."
140    GOTO Retry
150  END IF
160 !
170  ABORT Bus              !Take control of bus
180  IF NOT BIT(IOSTAT(Bus,300),6) THEN
190    PRINT "DEV_CHECK: Interface is not active controller."
200    BEEP
210    GOTO Retry
220  END IF
230 !
240  Listeners=0
250  FOR Address=0 TO 29
260    SEND Bus;UNL;MTA;LISTEN Address;DATA  !Address & drop ATN.
270    IF NOT BIT(IOSTAT(Bus,304),13) THEN   !Check for NDAC.
280      PRINT "   A listener is present at address ";Address
290      Listeners=Listeners+1
300    END IF
310  NEXT Address
320  IF Listeners=0 THEN PRINT "DEV_CHECK: No listeners found."
330  !
340 Stop:PRINT "DEV_CHECK: END"
350  STOP
360  !
370  END
```

Interface Control Registers

There are five control registers defined for interface driver HP27110. The use of these registers is described in the "Serial Polling", "Parallel Polling" and "Using the Interface as a Bus Device" sections of this chapter.

Resetting the Bus

There are three statements which you can use to bring the interface, bus and bus devices to a known state: RESET, ABORT and CLEAR. You can also use data messages to reset some bus devices. The RESET and ABORT statements specify only an interface ("Bus" in the following table); the CLEAR statement can specify either an interface or one or more bus devices ("Device" in the table).

The following table summarizes the interface and bus activity resulting from these statements. The first table assumes that the interface is the system controller.

	RESET Bus	**ABORT Bus**	**CLEAR Bus**	**CLEAR Device**
Bus messages sent	IFC,REN	IFC,REN	DCL	SDC
Bus configuration after statement	None	None	Unchanged	Changed
Subsequent controller status	AC	AC	Unchanged	Unchanged
Interface address as device	Per S1..S5	Unchanged	Unchanged	Unchanged
SPOLL, PPOLL and INTR masks	Set to 0	Unchanged	Unchanged	Unchanged
PPOLL sense byte & delay register	Set to 0	Unchanged	Unchanged	Unchanged

The following table assumes that the interface is not the system controller.

Bus messages sent	**None**	**MTA,UNL**	**DCL**	**SDC**
Subsequent controller status	Unchanged	Unchanged	Unchanged	Unchanged
Freeze bus on CMD parity error	No	Unchanged	Unchanged	Unchanged

Use the RESET statement to reset the interface card to its power-on state and perform an implied ABORT. RESET is useful for clearing the INTR, PPOLL and SPOLL registers and restoring the default device address of the interface.

RESET clears the PPOLL sense and delay register and this may be a disadvantage. If you are using HP 37203A Bus Extenders, another user (typically the AUTOST program) may have set HP27110 register 307 for a longer PPOLL delay. Your program must restore this register to the appropriate value after RESET for proper PPOLL operation. See the "Parallel Polling" section of this chapter for more information.

Use the ABORT statement for any of the following operations.

- Use ABORT to simply unconfigure the bus. All bus devices should unlisten or untalk when they detect the IFC or (interface) MTA and UNL messages.
- If the interface is system controller (SC), but not active controller (AC), use ABORT to seize control of the bus. This is not normally required unless control has been passed to another controller with the PASS CONTROL statement.
- Use ABORT to terminate a data transfer on the bus which does not involve the interface. The procedure for initiating such a transfer is discussed in the "Device to Device Transfers" section of this chapter.

CAUTION

When the interface is system controller, ABORT pulses the IFC line. Some devices incorrectly respond to IFC by performing a device reset operation instead of simply unlistening and untalking; for example, the HP 82905A/B Printer. You should not routinely perform ABORT operations on a bus to which such devices are connected. Device damage can occur if IFC is asserted frequently.

Use the CLEAR statement to set bus devices to a pre-defined state. If you specify an interface select code, CLEAR sends the Device Clear (DCL) bus message to all devices. If you specify one or more device selectors, CLEAR sends the Selective Device Clear (SDC) message to the specified devices.

The response of a device to DCL or SDC is device-dependent. Consult the device documentation. For example, either DCL or SDC resets the HP 2631B printer to its power-on state.

Not all devices respond to the DCL and SDC messages. Check the device's capability code. The following table shows the relationship between the DC capability codes and support for DCL and SDC.

	DC0	DC1	DC2
DCL Support	No	Yes	Yes
SDC Support	No	Yes	No

You can individually reset many devices by sending secondary commands or reserved data values. Using the HP 2631B as an example, you can accomplish the same reset performed with DCL/SDC by sending either of the following command and data messages.

```
420  OUTPUT 406;CHR$(27)&"E";   !Send <ESC> <E>

420  OUTPUT 40616;CHR$(0);      !Send secondary command 16 & null byte
```

Bus Configuration

In order to transfer data between bus devices, one device must be configured as a talker and one or more other devices configured as listeners. This reconfiguration process is performed automatically by I/O statements. For simple I/O, you do not need to know the details of this process. For more advanced operations, you should understand what the I/O statements do and how you can perform the same action with the `SEND` statement.

This section shows how I/O statements automatically reconfigure the bus and how you can use the `SEND` statement to explicitly reconfigure the bus. It also discusses direct bus I/O, which does not involve reconfiguration and ends with the I/O considerations involved when the interface is not the active controller.

Bus Reconfiguration

The following example assumes that there are printers at addresses 6 and 12 on bus interface 4. This section describes the bus messages sent by this statement, the actual bus activity, and shows the equivalent `SEND` sequence.

```
420  OUTPUT 406,412;"HP"            !Print characters "HP"
```

This `OUTPUT` statement performs a typical bus reconfiguration and sends data bytes to the printers. According to the "bus reconfiguration" entry in the BASIC Language Reference manual Glossary, the bus messages sent are:

```
ATN,MTA,UNL,LAG,ATN,DAB
```

What exactly does this mean? The following table shows the sequence of logic states on the ATN, DIO and handshake (NDAC, NRFD, DAV) bus lines. No other bus lines are involved.

Message	ATN Line	DIO Lines	Hand-shake	DIO value and description of bus message sent
ATN	TRUE	xxxxxxxx	No	All devices prepare to accept a bus command. The state of the DIO lines is irrelevant as no handshake occurs.
MTA	TRUE	01011110	Yes	30: My Talk Address; this talk address makes device 30 (the interface) a talker and implicitly makes all other devices cease talking (untalk).
UNL	TRUE	10111111	Yes	31: Unlisten; this reserved listen address makes all devices cease listening.
LAG				Listen Address Group
	TRUE	00100110	Yes	06: Listen address; makes printer 06 a listener.
	TRUE	00101100	Yes	12: Listen address; makes printer 12 a listener.
ATN	FALSE	xxxxxxxx	No	The currently addressed listeners prepare to accept bus data (not bus commands).
DAB	FALSE	01001000	Yes	Data Bytes(s) "H": The ASCII H character.
	FALSE	01010000	Yes	"P": The ASCII P character.
	FALSE	00001101	Yes	The ASCII CR character.
	FALSE	00001010	Yes	The ASCII LF character.

Notice that the listen address group (LAG) could consist of a single listen address and any of its listen addresses could include one or more secondary addresses. The data byte (DAB) message is always at least one data byte and can be many.

If you want to explicitly control the bus, you could perform the same operation as the `OUTPUT` statement by using the following sequence in the `SEND` statement.

```
420  SEND 4;MTA;UNL;LISTEN 6,12;DATA "H","P",CHR$(13),CHR$(10)
```

You do not explicitly specify the state of ATN in the `SEND` statement. When you specify a command term, `SEND` automatically asserts ATN TRUE. When you specify the `DATA` term, `SEND` automatically asserts ATN FALSE.

As you can see, `SEND` is not required for normal bus I/O. The previous example is included to help you understand what `SEND` does. `SEND` is provided so that you can perform less common I/O operations on the bus. The "Introduction to Input" chapter has an example of one such use; sending a unusual command sequence (an UNT followed by a secondary command) to an HP 2631B printer. See the "Ready for Input" section.

Another application involving an unusual bus configuration, and requiring the `SEND` statement, is any I/O transfer in which the computer is neither a talker nor a listener. This topic is discussed in a subsequent section, "Device to Device I/O".

Direct Bus I/O

When you specify one or more device selectors in an I/O statement, the statement reconfigures the bus. If you know the bus configuration, you may not want it changed. You can perform bus I/O using the current configuration by specifying only the bus interface select code in the statement. This is called "direct bus I/O".

This example uses the bus configuration as it was left after the preceding example. The data string is printed on both printers.

```
450  OUTPUT 4;"Using the same configuration."
```

There is a risk in using direct bus I/O. When the interface is active controller, another user or program might execute an I/O statement between your configuring statement and your direct bus I/O statement(s). If the interface is not addressed as a talker (for output) or as a listener (for input), an error occurs when the direct I/O statement executes. To avoid difficulty, statements should always reconfigure the bus or should use one of the cooperative device locking methods discussed at the end of the "Advanced I/O Operations" chapter.

Direct bus I/O is more commonly used when the interface is acting as a device (is not active controller). In this case, an error does not occur if the bus is not correctly configured. The I/O operation waits to become a talker (detects MTA) on output or waits to become a listener (detects MLA) on input.

Computer to Computer Example

In the following direct bus I/O example, two HP 9000 Model 520 computers are interconnected using HP-IB. The HP 27110A interfaces are installed as follows:

	System A	System B
Select code	4	5
System controller	Yes	No
Speed	High	High
Resistor pack	In	Out
Device address	28	29

One interface is set as the system controller and is currently the active controller. This example demonstrates high-speed transfer of a REAL array between computers. Either program can be the first to execute; each waits for the other at their `TRANSFER` statements.

Relevant code in System A:

```
 30  DIM Data$[1024] BUFFER
 40  ASSIGN @Data TO BUFFER Data$
 50  ASSIGN @Bus TO 5                      !Wait until addressed
      .
      .
120  TRANSFER @Bus TO @Data;COUNT 1024     !Read data when available
```

Relevant code in System B:

```
 30  DIM Data$[1024] BUFFER
 40  ASSIGN @Data TO BUFFER Data$
 50  ASSIGN @Cpu TO 429                    !Use direct I/O
      .
      .
120  TRANSFER @Data TO @Cpu;COUNT 1024     !Address & send string
```

CAUTION

When interconnecting HP-IB interfaces which can represent variable device loads, such as the HP 27110A, make certain that the total number of device loads is less than 15. If you exceed 15 device loads, one or both interface cards may be damaged.

Device-to-device I/O

When the interface is active controller it is possible to reconfigure the bus to permit data transfer between bus devices **other** than the interface. For example, you can establish a voltmeter as a talker and a printer as a listener for a data logging application. There are two ways to accomplish this operation.

- You can make the interface a listener by specifying the interface's device address as one of two (or more) listen addresses in an input statement. This method has two disadvantages: you must know how many data bytes to input and ignore, and you must establish a variable or buffer to hold the data.

- You can configure the talker and listeners with the SEND statement, omitting the interface as a listener. This method requires a special technique described in the following paragraphs which configures the interface to sink (handshake and ignore) the data transferred.

When the interface is active controller (unless it a talker or listener) it holds off or freezes the bus handshaking of data bytes (but not commands). To "unfreeze" the handshake you must write into control register 307 the number of data bytes the interface is to permit to pass between the (other) talker and listeners on the bus. The interface must be a listener, but ignores the data.

Here is an example of configuring an HP 3437A System Voltmeter (address 525) to take 100 readings and print them on an HP 2631B Printer (address 506). Each reading (except the last) consists of six alphanumeric characters followed by a comma, a total of 7 × 100 − 1, plus a CR and an LF with EOI, for a total of 701 bytes.

```
100  REMOTE 525                     !3437A to REM state
110  OUTPUT 525;"F1,T1,R3,N100S,D.2S"; !Program 3437A
120  SEND 5;UNL;TALK 25;MLA;LISTEN 6    !Configure bus
130  CONTROL 5,307;701              !Allow I/O on bus
```

If you do not know the total byte count you can specify a large number (up to 32767). The sink operation terminates when the talker asserts EOI. You can also terminate the transfer at any time with the ABORT or RESET statements.

Bus I/O Considerations

This sections discusses considerations unique to the HP 27110A HP-IB interface which apply to simple I/O operations. The two areas covered are:

- obtaining maximum bus performance;
- using the interface as a device.

This section does not deal with considerations unique to test instrument programming or the dedicated HP-IB statements. Those topics are covered in the next two sections.

Performance Considerations

There are two aspects to bus performance, response time and average data transfer rate or throughput. The time between the initiation of an I/O operation and the start of data transfer is the response time. There are areas in which you have choices that affect bus performance, system configuration and selection of I/O statements.

When configuring a system, HP-IB peripherals are divided into two data rate classes, and three response time classes. The data rate class is based on whether or not the device supports the medium or high speed handshake. You should have a separate HP-IB interface for each speed. If you have to mix medium and high speed devices on the same bus, you must set the interface for medium speed operation.

There is less distinction between response times. In general, the classes are:

- immediate response devices, typically disc drives (These devices, even if not ready, always respond to driver requests within a few seconds. An interface with a disc drive connected to it should have only disc drives connected to it. If you have two or more discs used heavily at the same time, you can avoid reduced performance by placing them on separate busses.);
- predictable response devices, typically printers and mag tape drives (These devices can handshake data faster than they can process it. They usually have internal buffers and when the buffer is full, they "hold-off" the bus handshake until space is available in the buffer. During hold-off, no bus I/O can occur, but the duration of the hold-off is predictable. At 180 characters per second, the HP 2631B empties (prints) its 229-byte buffer in 1.3 seconds. The longest HP 7971A tape record is read or written in 0.9 seconds, although a rewind can take 5 minutes. These devices should not be connected to a disc bus unless you always have a reasonable timeout in effect.);
- indefinite response devices; (These devices, such as an externally triggered instrument or graphics tablet, can hold off bus handshake for very long periods of time during input operations. A digitize operation on a 9111A tablet holds off bus activity until the operator presses the stylus switch. These devices should be on a separate bus if long response times are anticipated.)

Throughput on the bus is affected by other bus activity and the type of I/O statement used. On other HP computers, the type of I/O statement used may select different I/O modes; such as interrupt per character, fast handshake or direct memory access (DMA). The HP 9000 Model 520's HP-CIO architecture uses only DMA. To achieve the highest throughput, you should minimize the time the system spends setting up for an operation (overhead) and maximize the amount of data sent by each operation.

The following table list the bus I/O statements in performance groups. For quantitative information on HP 9000 Model 520 BASIC performance, request a copy of the HP 9000 Performance Brief from your HP Sales Representative. This publication (5953-9404) is available without charge.

I/O Statements	**Overhead**	**Transfer Rate**
ENTER...;FORMAT ON ENTER...USING (other than #,-K)	High	Slow
OUTPUT...USING OUTPUTBIN OUTPUT...;FORMAT ON, PRINT PRINT PRINT USING SEND	High	Fast
ENTER...;FORMAT OFF ENTER...USING "#,-K..."	Medium	Fast
OUTPUT...;FORMAT OFF TRANSFER (inbound or outbound)	Low	Fast

Using the Interface as a Device

If you need to connect your interface to a bus which already has another (active) controller, your interface must not be active controller; that is, your interface must be configured to act as a device. Acting as a device is useful when the other controller cannot; for example, if you need to connect your HP 9000 to the HP 12009A or HP 59310B HP-IB interface of an HP 1000 computer.

There are two ways to make your interface a device.

- You can set switch S7 of the HP 27110A interface to the DOWN (non-system controller) position. At power-up or after `RESET`, the interface acts as a device and cannot become active controller unless another controller passes control to it. This configuration is useful when the interface is connected to a bus on which another HP-IB interface is normally system controller.
- You can relinquish active control of the bus. Regardless of whether your interface powers-up as active controller or is passed control, you can give up control with the `PASS CONTROL` statement. The `PASS CONTROL` statement sends the TCT message (CMD 9 or 137) to the specified device. Because the HP 27110A interface handshakes commands regardless of whether another device does, you can pass control to a controller which otherwise could not accept it (C17 through C28 capable devices). If the other controller already "thinks" that it is active controller, you can simply relinquish control by passing control to a non-existent device.

When your interface is a device, you perform I/O by specifying only the interface select code of the bus. Since the interface is not active controller and cannot reconfigure the bus, you cannot specify a device selector. If the active controller of the bus has already addressed your interface as a talker or listener, the output or input operation handshakes data immediately. If the interface is not addressed, the operation waits until the interface receives its talk (MTA) or listen (MLA) address.

When your interface is a device, it can send data messages but not command messages. That is, you can perform data I/O (`OUTPUT, ENTER`) but you cannot use most of the dedicated HP-IB statements. The following table summarizes the effect of using the HP-IB statements.

Statement	**Interface is not System Controller**	**Interface is System Controller**
ABORT	No operation	Assumes active control
CLEAR	Error	Error
LOCAL	Error	Partial function
LOCAL LOCKOUT	Error	Error
PASS CONTROL	Error	Error
PPOLL	Error	Error
PPOLL CONFIGURE	Error	Error
PPOLL RESPONSE	Sets response	Sets response
PPOLL UNCONFIGURE	Error	Error
REMOTE	Error	Partial function
REQUEST	Can assert SRQ	Can assert SRQ
SEND	SEND...;DATA only	SEND...;DATA only
SPOLL	Error	Error
TRIGGER	Error	Error

When acting as a device, the HP 27110A interface can automatically respond to many bus commands and uni-line messages. For those messages which the interface does not transparently support, you can enable the interface to interrupt when they occur. This requires setting bits in the interrupt enable mask, which is discussed later in this chapter. The following table lists the bus messages and describes how the interface supports them.

ATN + EOI - Parallel Poll - If card was enabled to respond to PPOLL with the POLL RESPONSE statement and its address as a device is in the range 0 thru 7, it asserts the corresponding DIO line 1 thru 8. If the address is 8 thru 30, the active controller must configure the card by sending the parallel poll configure (PPC - PPE) message, or you must program control register 306.

DAB - Data Byte(s) - Data is transferred if an input or output statement is pending and the interface is a listener or talker. Otherwise, the presence of one or more data bytes in the card's buffer can interrupt using mask bit 9.

DCL - Device Clear - Not transparent; this message can interrupt using mask bit 0.

EOI - End or Identify - Can terminate item, ENTER statement, be ignored or cause an error, per image specifier.

GET - Group Execute Trigger - Not transparent; this message can interrupt using mask bit 5.

GTL - Go To Local - Not supported

IFC - Interface Clear - Interface unlistens and untalks. This message can also interrupt using mask bits 1 or 8.

LLO - Local Lockout - Not supported

MLA - My Listen Address - Interface becomes a listener. This message can interrupt using mask bit 10.

MSA - My Secondary Address - See SCG.

MTA - My Talk Address - Interface becomes a talker. This message can interrupt using mask bit 11.

PPC - Parallel Poll Configure - Interface is enabled to respond to a parallel poll.

PPD - Parallel Poll Disable - Interface PPOLL response is disabled.

PPE - Parallel Poll Enable - Programs the interface's PPOLL response bit.

PPU - Parallel Poll Enable - Cancels interface's PPOLL response bit.

REN - Remote Enable - Not transparent; this message can interrupt using mask bit 14.

SCG - Secondary Command Group - Not transparent; use the technique described in the next section to read secondary commands. Secondary commands during data input can interrupt using mask bit 7.

SDC - Selective Device Clear - Not transparent; this message can interrupt using mask bit 0.

SPD - Serial Poll Disable - Interface ceases asserting SRQ if true.

SPE - Serial Poll Enable - Interface sends either 01000000 (SRQ true) or 10000000 (SRQ false) on the DIO lines.

SRQ - Service Request - Interface can assert this bus line with the REQUEST statement. If another device asserts SRQ, the interface can interrupt using mask bit 12.

TCT - Take Control - Interface becomes active controller. This message can interrupt using mask bit 15.

UNL - Unlisten - Interface ceases listening.

UNT - Untalk - Interface ceases talking.

Reading Secondary Commands

Because the value of a byte on the DIO lines does not reflect the state of the ATN line, a special technique is required for reading secondary commands. During an ENTER statement, the detection of a secondary command causes an ERROR 485 and the handshake is held off. The secondary is held in status register 307. The following program lines demonstrate reading secondaries.

```
120  ON ERROR CALL Enter_error       !Trap any error
130  ENTER 5;Data$                   !Input as device
   .
   .
810 Enter_error:IF ERRN=485 THEN     !Oops, had an error
820    Secondary=IOSTAT(5,307)       !Read entire register
830  ELSE                            !Error wasn't secondary?
840    DISP ERRM$                    !Normal
850    PAUSE                         !   error messaging
860  END IF
870    SUBEXIT                       !
```

Note that register 307 can also indicate whether it actually contains a secondary (bit 7 = 1) and if the interface was being addressed as a talker (bit 5 = 1) or listener (bit 5 = 0) when the secondary was detected.

Bus Instrument Programming

Although HP-IB is used for supporting a wide variety of computer peripherals, it was originally developed for connecting test instruments to controllers (computers). Consequently, there are bus functions and considerations which largely apply only to instruments. This section describes the use of test instruments on a bus and introduces the `LOCAL`, `LOCAL LOCKOUT`, `REMOTE` and `TRIGGER` statements.

Remote/Local

Unlike most computer peripherals, test instruments are useful even when not connected to a computer. They are usually equipped with "front panel" controls and can be controlled by their human operator. When connected to a computer, the question arises of who is in control of the instrument. There are three HP-IB concepts which apply to this issue:

- **LOCAL** - The instrument is under the control of its operator. All front panel controls are enabled. Instruments in the local state generally ignore bus commands and data until programmed into the remote state. An instrument in local handshakes bus commands but does not handshake bus data.
- **REMOTE** - The instrument is under HP-IB control. All possible front panel controls except "LOCAL" are disabled. The operator can return the instrument to the local state with the return-to-local switch/button. Non-programmable (typically mechanical) front panel controls, such as the power switch, are not disabled when the instrument is in the remote state.
- **LOCAL LOCKOUT** - This bus message programs an instrument to disable its return-to-local control. The operator cannot manually return the instrument to the local state. If the instrument is not in the remote state, local lockout does not take effect until the instrument is programmed to the remote state.

Most HP-IB instruments power-up in the local state. They ignore bus messages until programmed to the remote state. An instrument enters the remote state when it is addressed to listen and the REN bus line is true. You do this with the `REMOTE` statement.

`REMOTE` sets REN true. If you specify only an interface select code, only those instruments currently addressed as listeners enter the remote state. If you specify one or more device selectors in the `REMOTE` statement, it reconfigures the bus and only the specified devices enter the remote state.

The `LOCAL` statement sets REN FALSE. if you specify only an interface select code, only those instruments currently addressed as listeners enter the local state. If you specify one or more device selectors in the `LOCAL` statement, it reconfigures the bus and only the specified devices enter the local state.

The `LOCAL LOCKOUT` statement sends the LLO universal bus command. You can specify only an interface select code. All instruments which support LLO and are in the remote state lock out their "return-to-local" control.

The following example assumes that there is an HP 3437A System Voltmeter at address 25 on bus 5.

```
510  REMOTE 525              !Program 3437A to remote
520  LOCAL LOCKOUT 5         !Local out "LOCAL" control
530  OUTPUT 525;"F1,T1,R1";  !Typical device instructions
```

Instrument Instructions

The previous example programs the HP 3437A Voltmeter to its ASCII mode, internal triggering and 1.0 volt range. It does this with the data string "F1,T1,R1". This type of bus data is called device or instrument instructions. Unlike HP-IB printers, HP-IB instruments do not normally print or display bus data sent to them; they act on the data instead.

Most instruments do not handshake bus data unless they are in the remote state (as well as being addressed as listeners). If the HP 3437A is not in the remote state, it does not participate in the three-wire handshake and therefore does not respond to the instructions. (The HP-IB interface does handshake the data, preventing a timeout error.)

Instrument instructions are sent on the bus as data messages (ATN FALSE) and not as bus commands. You can send data messages with `OUTPUT`, `OUTPUTBIN`, `OUTPUT...USING` or `SEND...;DATA...`. You can also use the `PRINT` and `PRINT USING` statements if `PRINTER IS` is assigned to the instrument(s).

Each instruction is generally a string consisting of a one or two letter mnemonic followed by a numeric modifier. The string is composed of ASCII alphabetic and numeric characters. If you include more than one instruction in a single output statement, you may need to delimit each instruction with a punctuation character. For example, the HP 3437A does not require a delimiter, but does allow a comma for clarity. The following example programs the HP 3437A to take one reading.

```
110  OUTPUT 525;"T1,N1S,D,0S";        !"S"="Store"
120  ENTER 525;Reading                !Input reading
```

Some devices, such as the HP 3437A, also support a "packed binary" instruction set. You must create these instructions by setting and clearing bits in binary data bytes. The following code sends a binary program (B) to the HP 3437A to set it to Packed Binary mode (F0), ENAB RQS off (E0), Internal Trigger (T1), 0.1 V Range (R1), NRDGS 100 (R100S) and a 0.5 second Delay (D.5S).

```
 10  DIM Instr$[7],Readings$[200]
  .
  .
710  Instr$[1,1]=CHR$(IVAL("00000101",2))   ! F0, E0, T1, R1
720  Instr$[2,2]=CHR$(IVAL("00000001",2))   ! 0100 Readings
730  Instr$[3,3]=CHR$(IVAL("00000000",2))
740  Instr$[4,4]=CHR$(IVAL("00000000",2))   ! 0.0200000 Sec delay
750  Instr$[5,5]=CHR$(IVAL("00100000",2))
760  Instr$[6,6]=CHR$(IVAL("00000000",2))
770  Instr$[7,7]=CHR$(IVAL("00000000",2))
780  OUTPUT 525;"B"&Instr$[1,7];                      !Send binary program
790  ENTER 525 USING "#,-K,200A";Readings$[1,200]     !Get 100 byte-pairs
```

The advantage of packed binary is that fewer data bytes are required to program the device. This is important when you must program the device frequently or it is returning a large volume of data. In unpacked (ASCII) mode, the bus I/O might consume enough time to slow down the test. The disadvantage of packed binary is that the creation of instructions and the interpretation of the returned data is more complex than with ASCII.

Some instruments require instruction terminator characters. For example, the HP 3437A does not store ENAB RQS ("E..."), NRDGS ("R...") or DELAY ("D...") instructions until terminated with an "S" character.

Other instruments do not act on or store instructions until they receive a request terminator. For example, the HP 2240A and 2250 Measurement and Control Processors can accept long sequences of (MCL) instructions, but do not act on them until you send an exclamation character ("!").

Most HP-IB instruments ignore unrecognized characters; others do not, such as the HP 3495A Scanner, and may respond in an inappropriate manner to the CR-LF default EOL sequence sent by output statements. Unless the CR-LF is required by the instrument, you should suppress the automatic EOL.

Using EOI

As the "General Interface Management" section indicated, the End Or Identify (EOI) line has two purposes. This section discusses how it is used by the current talker to delimit (end) a data message.

The cases in which you must be most conscious of EOI are those which involve binary data transfers; that is, those operations where the value of a data byte on the DIO lines does not necessarily represent an ASCII character. EOI allows the talker to send any sequence of 8-bit data values and denote which byte is the last by asserting EOI true while handshaking that byte.

On input, if you read a binary value of 00001010 from an instrument, the number builder treats it as an ASCII linefeed (LF) character, which is the default DELIM character. This can cause premature input termination or an error. To ignore LF, but not EOI, perform the input operation with the `ENTER...USING` statement, using an image specifier of `-K`.

During input, EOI is not an END condition; however, there are image specifiers which provide special processing of EOI.

- Use `#` to terminate input only when the variable list is satisfied. EOI can terminate input data items but `ENTER` does not continue reading bytes scanning for DELIM/EOI after reading a value for the last input item. Input is very slow unless you also specify -K.
- Use `%` to terminate input immediately on detection of EOI. An error occurs if the EOI occurs within a fixed-width field, such as a substring.
- Use `+` when you want the `ENTER` statement to continue reading data bytes after satisfying the last variable and cease reading data bytes only on detection of EOI. EOI must occur within 256 bytes of the last used data byte.
- Use `-` when you want `ENTER` to ignore EOI as a statement terminator. `ENTER` terminates only on `DELIM`. EOI is still recognized as an item terminator.

Many devices assert EOI with the last data byte even when the data is ASCII and is also terminated by carriage-return linefeed (CR LF) or simply LF. EOI is asserted with the LF. If you are entering the data into a string, the CR may be stored in the string as character data if your program does not account for this possibility. If you use the the -K specifier, `ENTER` also treats the LF as string data.

On output, EOI is not asserted unless you explicitly specify it. There are four ways to assert EOI with a data byte.

- For I/O path name, PRINTER IS and PRINTALL IS output, you can specify an END term in the EOL attribute expression. This asserts EOI with the last character of each EOL sequence.
- In the OUTPUT and OUTPUT...USING statements, you can specify an END term after the last data item. This asserts EOI with the last data byte output. END also suppresses the automatic EOL sequence.
- In the TRANSFER statement, the EOR(...END...) and END attributes assert EOI with the last outbound data byte in each record.
- In the SEND statement, the DATA...END expression asserts EOI with the last data byte of the preceding DATA expression.

Instrument Triggering

Some bus devices respond immediately to instructions, others respond when sent an "act now" instruction, like the exclamation character. Most test instruments require a special bus command, the Group Execute Trigger (GET) message.

There are advantages to using triggering even when it is optional. Once you have sent instructions to an instrument, you may want to have it perform its task several times without having to repeat the instructions. If you are programming two or more instruments, you may want to send (different) instructions to each, but have them perform the task simultaneously. You can perform these bus operations with the TRIGGER statement.

The HP 3437A voltmeter samples immediately (after delay) in internal trigger (T1) mode and only upon an electrical signal or GET command in external trigger (T2) mode. It also has a maximum sampling delay of one second. The following example uses triggering to take a voltage reading after a ten second delay.

```
110  REMOTE 525                      !Program 3437A to remote
120  LOCAL LOCKOUT 5                 !Lock in remote
130  OUTPUT 5;"D.0S,N1S,E0S,R2,T2,F1"; !Program 3437A
140  ON DELAY 10 GOTO Trigger        !Set up delay
150  WAIT                            !Wait for ON... cond.
160 Trigger:TRIGGER 5                !Trigger one reading
170  ENTER 525;Reading               !Read result
```

You can specify either an interface select code or one or more device selectors in the TRIGGER statement. If you specify the interface, the GET message is sent to all current listeners. If you specify one or more device selectors, the specified devices are addressed to listen before the GET message is sent. The previous example used bus triggering because it is slightly faster than reconfiguring the bus on each TRIGGER.

Other Instrument Considerations

Some instruments require very precise instruction formats and do not tolerate extraneous spaces and other punctuation; for example, the HP 1980A/B Oscilloscope does not allow extra spaces. For this reason, unless you know the device to be free-field tolerant, you should use the OUTPUT...USING or PRINT USING statement to send instructions and should avoid the default fields of the simple OUTPUT and PRINT statements.

Some early HP-IB devices have instruction sequences which are unconventional; for example, the HP 8660A/C Synthesized Signal Generator demands its numeric sequences in reverse character order. You can use either `VAL$` or `OUTPUT` (to a string) to create the string numeric, reverse it with the `REV$` function and then send it with `OUTPUT`.

Some instruments are sensitive to parallel polling; for example, the HP 3497A Data Acquisition/ Control Unit cannot assert SRQ during a parallel poll. This is a consideration with the HP 27110A interface because it may conduct a continuous parallel poll (set and leave EOI and ATN true) when the bus is idle. See the "Parallel Polling" section of this chapter for more information.

HP-IB Interrupts

The HP 27110A interface can interrupt the computer on the occurrence of any of 12 bus conditions. Your program can respond to the interrupt with an `ON INTR` statement or a `WAIT FOR INTR` statement. Which condition, or combination of conditions cause the interrupt depends on the value of the interrupt mask register that you specify in the `ENABLE INTR` statement. Here is the "bit map" of the mask register from the "I/O Resources" section of the BASIC Language Reference manual.

31-24	23-16	15	14	13	12	11	10	8	7	6	5	4-2	1	0
0	PPOLL	CIC	REM	0	SRQ	TLK	LTN	DAT	NCC	MSA	PTY	0	IFC	DCL

Each of the conditions represented by these bit fields can interrupt only if the corresponding mask bits are one. Each field is described separately in the following paragraphs.

PPOLL - If you set any of these bits to one, the interface conducts a continuous parallel poll when there is no other bus activity; that is, it sets both the ATN and EOI lines true and monitors the DIO lines. If any devices set one or more of the DIO lines true in response to the parallel poll, and the exclusive-or of those lines (1..8) and the PPOLL sense bits in control register 306 correspond to any in the mask bits (16..23), an interrupt occurs. See the "Parallel Polling" section for more information.

CIC - This bit can cause an interrupt when the interface becomes active controller (controller in charge) of the bus. Another bus device can cause this interrupt by passing control (sending the TCT message). If the interface is system controller, your computer can cause this interrupt by taking control with the `ABORT` statement (but not the `RESET` statement). `ABORT` can also cause a bit 0 (IFC) interrupt.

REM - This bit can cause an interrupt when the interface is programmed to "remote" by another controller; that is, when it is addressed to talk or listen and the REN line is true. There is no corresponding interrupt when the interface is programmed to "local", but you can periodically examine the remote/local state of the interface with bit 15 of status register 303. The remote/local bus messages have no other effect on the interface; these functions are supported only by applications software. There is no interrupt or status register support for the local lockout (LLO) message.

SRQ - When the interface is active controller this bit can cause an interrupt when any device(s) on the bus (including the HP 27110A interface) asserts the SRQ line true. SRQ is the "generic" HP-IB interrupt and is the only way most devices can interrupt your computer. There is an example of this common bus interrupt in the "Servicing Interrupts" section of the "Advanced I/O Operations" chapter of this manual. When the interface is acting as a device this bit cannot interrupt. Each device asserting SRQ de-asserts it when subsequently polled with `SPOLL` or if the condition within the device goes away.

TLK - When the interface is acting as a device this bit can cause an interrupt when the interface is addressed as a talker (receives MTA). There is no corresponding interrupt when the interface is unaddressed as a talker, but you can periodically examine the talker status of the interface with bit 9 of status register 303.

LTN - When the interface is acting as a device this bit can cause an interrupt when the interface is addressed as a listener (receives MLA). There is no corresponding interrupt when the interface is unaddressed as a listener, but you can periodically examine the listener status of the interface with bit 10 of status register 303. This bit can also interrupt if the interface receives a universal identify (UNT,SEC Primary_address) sequence.

DAT - When no there is no input operation (`ENTER`, `ENTERBIN`, `ENTER...USING`) active on the interface, this bit can cause an interrupt when the interface is a listener and another device sends it one or more data bytes. The bus handshake is frozen, so the incoming data is not lost and can be read by a subsequent input statement. This condition is cleared by any input operation which reads all of the available data bytes.

NCC - When the interface is active controller and loses control by passing it or by having it taken away by the system controller, this bit can cause an interrupt. If control is taken away, the IFC bit can also cause an interrupt.

MSA - When the interface is addressed as a listener, this bit can cause an interrupt if any secondary command is received. If an input statement is active, a BASIC ERROR 485 occurs. In either case, the bus handshake is frozen and the secondary command is stored in status register 307. This condition is cleared when the next byte is read from the bus.

PTY - When acting as a device this bit can interrupt if a bus command is received without odd parity. You can independently use control register 305 to program the interface to freeze the handshake on bad parity. This condition is cleared when the next byte is read from the bus.

GET - This bit can interrupt if the interface receives a Group Execute Trigger command while acting as a device.

IFC - This bit can cause an interrupt when the system controller asserts the IFC bus line.

DCL - This bit can cause an interrupt when the interface receives either a Device Clear (DCL) or Selective Device Clear (SDC) message from the active controller.

In addition to the mask register there are two interface status registers related to interrupts. When an interrupt occurs, register 301 contains the reason for the interrupt. This register has the same binary bit assignments as the mask register. Register 302 contains the most recently written mask register. If your interrupt service subprogram needs to re-enable the interface for different conditions than those that caused the interrupt, it can read register 302, change the appropriate bits, and write the new mask out with an `ENABLE INTR` statement. This avoids having to place the mask in COM or pass it as a parameter to the subprogram.

The following program demonstrates enabling the HP 27110A interface for all interrupts (except PPOLL) and then reporting the reason for the interrupt.

```
 10  OFF INTR Bus                         !Cancel any existing ENABLE INTR
 20  Mask=IVAL("1101111111100011",2)      !Interrupt on any condition
 30  ENABLE INTR Bus;Mask                 !Write out mask
 40  ON INTR Bus,15 RECOVER Intr          !On intr, exit simple WAIT
 50  WAIT                                 !WAIT FOR INTR locks keyboard
 60 Intr:R301=IOSTAT(Bus,301)             !Read interrupt reason
 70  IF BINAND(R301,65535) THEN           !Get bits of interest
 80    PRINT LIN(1);"The last interrupt was caused by detection of..."
 90    IF BIT(R301,0) THEN PRINT " a CLEAR (DCL)"
100    IF BIT(R301,1) THEN PRINT " an ABORT or RESET (IFC)"
110    IF BIT(R301,5) THEN PRINT " a command without odd parity"
120    IF BIT(R301,6) THEN PRINT " a secondary command (MSA)"
130    IF BIT(R301,7) THEN PRINT " I lost active control"
140    IF BIT(R301,8) THEN PRINT " a data byte was sent to me"
150    IF BIT(R301,10) THEN PRINT " I am addressed as a listener"
160    IF BIT(R301,11) THEN PRINT " I was addressed as a talker"
170    IF BIT(R301,12) THEN PRINT " a service request (SRQ)"
180    IF BIT(R301,14) THEN PRINT " remote enable is true (REN)"
190    IF BIT(R301,15) THEN PRINT " I am active bus controller"
200  ELSE
210    PRINT "Interrupt for unknown reasons; mask=";DVAL$(R301,16)
220  END IF
230  END
```

Bus Device Status

During bus operations, you may need to know the state of a device. This requirement most commonly arises when the interface detects that one or more devices have asserted SRQ and the interface interrupts the computer. When this happens you need to determine which device interrupted and why.

When an interrupt occurs, the following steps are suggested for servicing the interrupt.

1. If all of the devices which can assert SRQ also support parallel poll (PPOLL) (capability PP1 or PP2) and are properly configured, you can perform a `PPOLL` to determine which devices to serial poll.
2. Conduct a serial poll (`SPOLL`) of each device which can assert SRQ and obtain its status byte. The status byte normally indicates whether or not the device asserted SRQ, and if so, why.
3. Perform the task requested by the device. This may require obtaining extended device status to further qualify the reason for the SRQ.
4. Re-enable interrupts with the `ENABLE INTR` statement.
5. Resume the interrupted task (if appropriate).

The following sections discuss steps one through three. You can also use these operations in non-interrupt applications; such as, determining that a device is ready before initiating an I/O transfer.

Parallel Polling

When the active controller conducts a parallel poll (PPOLL), each bus device having PP1 or PP2 capability may return one bit of status information. This section discusses the following aspects of parallel polling:

- how the poll is conducted;
- what the device's response means;
- how the device is configured to respond.

To conduct a parallel poll, the IEEE 488-1978 standard requires the controller to assert the ATN and EOI lines to the true state for at least 100 nsec and then read the state of the DIO lines. The reading of the DAB response is done without bus handshake, so the polled devices need not be currently addressed as talkers. The HP 27110A interface can conduct a parallel poll by two methods, as a result of the `PPOLL` function or while the bus is idle.

If you use the `PPOLL` function, the interface holds the ATN and EOI lines true for at least 25 μsec and then returns the contents of the DIO lines as a data byte value. The 25 μsec delay is adequate for a local bus. If you are polling devices on a remote bus, you must increase the delay to account for the signal propagation time through the HP 37203A bus extenders. You can increase this time exponentially writing a byte value into the low order byte of control register 306. Refer to the HP 37203A manual for appropriate time values from which to calculate the exponent.

If you configure the card to perform the idle poll, the ATN and EOI lines are left true indefinitely. You can enable idle polling two ways.

- If you use the `ENABLE INTR` statement to write a mask which has one or more bits set to one in the PPOLL field, the interface conducts a poll whenever the bus is idle. To disable this poll you must write a mask which has this field set to zero, and perform any bus operation which sets ATN false; such as:

  ```
  SEND Bus;DATA
  ```

- When any bus operation leaves the ATN line in the true state, the interface also asserts EOI true, conducting a continuous parallel poll. Input and output statements do not leave ATN true; most HP-IB statements do. You can explicitly leave ATN true with the statement:

  ```
  SEND Bus;CMD
  ```

  To disable the idle poll, use the statement:

  ```
  SEND Bus;DATA
  ```

During both polling operations the result returned by `PPOLL` and the result compared to the PPOLL interrupt mask field is first exclusive-or'd with the sense-byte field of control register 306. In the `PPOLL` function case the returned byte is given by the following algorithm, assuming `Cr306` = control register 306.

```
BINEOR(Dio_dab,SHIFT(Cr306,8))
```

In the PPOLL interrupt case, the interrupt can occur only if the result of the following algorithm is non-zero:

```
BINAND(BINEOR(Dio_dab,SHIFT(Cr306,8),SHIFT(Intr_mask,16)))
```

Because the default value of control register 306 is zero, the exclusive-or operation generally has no effect and you can ignore it.

If a device is configured to respond and has reason to respond, the parallel poll causes it to set one of the DIO lines to the true (1) state. Typically each of up to eight devices uses a different line. The meaning of this line assertion is device dependent, although in general it means that a condition which could cause SRQ to exist within the device. More than one device can assert the same DIO line. Only one needs to assert a line to one for the line state to be one. This feature is useful if you have more than eight devices on a bus and want to have groups of similar devices share the same line.

Which DIO line the device asserts depends on its parallel poll configuration. You can program the PPOLL response of devices with PP1 capability with the `PPOLL CONFIGURE` statement. This statement specifies a byte value which signifies both the DIO line the device is to use and whether the device is to assert the line or not. `PPOLL CONFIGURE` sends the PPC and PPE messages.

PP2 devices are configured either by local controls or are permanently configured, usually to use the DIO line corresponding to: `8 - (Primary_address MOD 8)`

You can disable the PPOLL response of PP1 devices with the `PPOLL UNCONFIGURE` statement. If you specify only the interface select code of the bus, `PPOLL UNCONFIGURE` sends the PPU message to all devices. If you specify one or more device selectors, it sends the PPC and PPD messages only to those devices.

The following is an example of parallel polling. It assumes that there is an HP 7580A Plotter and HP 2631B printer at addresses 5 and 6 (respectively) on bus 4.

```
100  PPOLL UNCONFIGURE 4                !Disable all devices
110  PPOLL CONFIGURE 405;IVAL("1101",2) !Set 7580 for DIO 6
120  PPOLL CONFIGURE 406;IVAL("1110",2) !Set 2631 for DIO 7
130  PRINT IVAL$(PPOLL(4),2)            !Print poll response
140  SEND 4;DATA                        !De-assert ATN
```

When acting as a device, the 27110A interface can respond to a parallel poll conducted by the active controller. You can enable and disable the interface's response with the `PPOLL RESPONSE` statement. The DIO line that the card uses is determined in one of two ways:

- If the interface's address as a device is in the range 0 through 7, its default DIO line for poll response is 8 through 1 respectively. If the address is above 7, the interface does not respond to parallel poll unless its response is programmed by the active controller.
- The active controller can program the interface's response by sending the PPC and PPE messages. You cannot use the `PPOLL CONFIGURE` statement to self-program the interface; the configuration must be sent by another controller.

Serial Polling

Parallel polling allows all bus devices to respond simultaneously to a poll but has the limitation that each device can return only one bit of status information. Another type of polling available with HP-IB is serial polling. A serial poll returns 8 bits (one byte) of status information from a single device. Serial polling is a feature of the service request (SR) bus capability and is supported by all devices which can assert SRQ (code SR1).

You conduct a serial poll with the `SPOLL` function, which specifies one and only one device selector of a bus device. Unlike parallel polling, serial polling addresses the device as a talker and the status data is transferred using the bus handshake. This means that there must be device at the address. You can establish whether or not there is a device at the address using the "device check" technique of the "Bus Status" section of this chapter. You may also have established that there is a device at the address via parallel poll. If you are not sure that an SR1 device exists at the address, establish a short timeout with an `ON TIMEOUT` statement.

An SR1 device responds to a serial poll regardless of whether it is asserting SRQ. Generally, each bit in the status byte has a separate meaning and often one of them (bit 6, DIO line 7) indicates whether the device is requesting service. In most cases each of the other bits indicates a condition within the device which can cause SRQ. In many cases you can output a mask device instruction which enables some combination of these conditions to cause SRQ. If the device has SRQ asserted, it de-asserts it when polled with `SPOLL`, or if the condition which cause the SRQ ceases.

The following example programs an HP 3437A System Voltmeter to generate SRQ only on data ready. It also programs the reading to occur after a one second delayed internal trigger.

```
 10  OFF INTR 5
 20  OUTPUT 525;"T2,E45,R3,F1,N1S,D.99999995";
 30  ENABLE INTR 5;IVAL("100000000000",2)
 40  TRIGGER 5
 50  Start=TIMEDATE
 60  WAIT FOR INTR 5
 70  PRINT "Waited";TIMEDATE-Start;"seconds."
 80  Status=SPOLL(525)
 90  IF BIT(Status,6) THEN PRINT "3437A asserted SRQ"
100  Mask=BINAND(Status,7)
110  PRINT "ENAB RQS mask was";Mask
120  Rqs_status=BINAND(SHIFT(Status,3),7)
130  PRINT "RQS STATUS is";Rqs_status
```

Extended Device Status

In addition to parallel and serial polling (which are defined by the IEEE standard), many bus devices support device instructions or secondary addresses which return additional status information, often referred to as "extended status". The messages required to request extended status vary considerably from device to device. Here are examples of obtaining extended status from the HP 2631B and 7580A.

```
110  ENTER 50614;Status$[1,1]        !Get HP-IB I/O status
120  !
130  Esc$=CHR$(27)
140  Dc1$=CHR$(17)
150  OUTPUT 506;Esc$&"&s1^"&Dc1$;    !Request strap status on 2631
160  ENTER 506;Strap$[1,10]          !Read strap status
170  !
180  OUTPUT 505;"OA;"                !Request position & pen on 7580
190  ENTER 505;Status$               !Read "x,y,P0/1"
```

You must refer to the device documentation for the precise syntax of the status message available with the device.

Appendix A
Utilities

BANNER Utility

This program prints poster-size messages horizontally down the page of the internal thermal printer forming a banner.

To run the BANNER program:

1. Load the BASIC Language System.
2. Load the BANNER utility and press (RUN).
   - If you are using the original flexible discs with which your system came, insert the disc labeled "BASIC Utilities Programs 2 of 3" in the disc drive and type:

     ```
     LOAD "BANNER:INTERNAL"          (EXECUTE)
     (RUN)
     ```
3. The allowable characters are printed on the display.
4. You are then asked if you want the banner to be printed in inverse video. If you type Y, the letters of the message are white and the background is black. If you type N, the background is white and the letters are black.
5. Next, type the message you want printed. The message may consist of any of the characters listed on the display. When done, enter the message by pressing (RETURN).
6. After the message is input, the program starts printing the banner.
7. If the program is stopped before the banner is completely printed, the printer needs to be reset by typing the following statements:

   ```
   PRINTER IS PRT (EXECUTE)
   PRINT CHR$(27)&"E" (EXECUTE)
   ```

CONVERGE Utility

The CONVERGE utility realigns the color focus of various parts of the HP 98770B internal high-performance color CRT display. If the white color on the display no longer looks white, but instead the composite colors (red, green and blue) are apparent, it is time to converge the screen.

To run the CONVERGE program:

1. Load the BASIC Language System.
2. Load the CONVERGE utility and press **RUN**.
   - If you are using the original flexible discs with which your system came, insert the disc labeled "BASIC Utilities Programs 1 of 3" in the disc drive and type:

     LOAD "CONVERGE:INTERNAL" **EXECUTE**
     RUN
3. Open the door to the right of the HP 98770B CRT to expose the 39 alignment locations and the alignment tool. The 39 locations are organized in rows by adjustment number (1 thru 13) and in columns by color (red, blue, green). Remove the alignment tool by pressing on its top.
4. After pressing **RUN**, a small white plus (+) is displayed in one area of the display. A number 1 and the characters "=>" appear in the right-hand column of the display. The + is used to determine whether that portion of the display is in alignment or if some adjustment needs to be made. The number and "=>" reminds you which row of alignment locations should be used to adjust the convergence.
5. Adjusting the alignment locations moves the plusses across the display according to the following rules.
   a. The red plus moves along a diagonal line from lower left to upper right.
   b. The blue plus moves along a vertical line.
   c. The green plus moves along a diagonal line from upper left to lower right.

   The easiest way to merge the plusses is to first merge the green and red plusses. This makes a yellow plus which should be located in line with the vertical portion of the blue plus. Then the blue plus is merged with the yellow, forming a white plus.

Note

A white plus with shaded (colored) edges rather than completely separate plusses, needs very little adjustment. Even when the three colored plusses are perfectly merged, a small amount of colored "fringing" may appear around the edges. This is normal.

6. After the plus is satisfactorily aligned, press **CONTINUE**. Another set of plusses appears in a different segment of the display. Continue the procedure stated in Step 5 for each location, two thru thirteen.
7. After a single plus has appeared for each location, all the plusses are printed to the display so you can examine its overall quality. To clear the display press **CONTINUE**. The display clears and the message "CONVERGE is done." is output. If the convergence is still not correct, re-run the program.

8. If at any point the program terminates before the "CONVERGE is done." message appears, it is necessary to execute the following statements to reset the printer and re-format the display.

   ```
   PRINTER IS CRT (EXECUTE)
   RESET SCREENS (EXECUTE)
   ```

   If the program terminates normally, there is no need to type in these statements.

CERTIFY TAPE Utility

The CERTIFY_TAPE utility re-certifies a DC tape. Note that certifying a tape causes all data on the tape to be destroyed.

To run the CERTIFY_TAPE Program:

1. Load the BASIC Language System.
2. Load the CERTIFY_TAPE utility and press (RUN).

   - If you are using the original flexible discs with which your system came, insert the disc labeled "BASIC Option Binaries Disc 2" in the disc drive and type:

     ```
     LOAD "CERTIFY_TAPE:INTERNAL" (EXECUTE) (RUN)
     ```

3. You are asked to enter the volume specifier.
4. Make sure the tape to be certified is in the tape drive and the tape drive is on-line.
5. Enter the volumr specifier of the tape drive.
6. The certify process can take from 20 to 160 minutes to complete, depending on the length and quality of the tape.

DEGAUSS Utility

The DEGAUSS utility demagnitizes the HP 98770B internal high-performance color CRT's color mask. This should be done after prolonged continuous use if the display appears fuzzy or irregular.

To run the DEGAUSS program:

1. Load the BASIC Language System.
2. Load the GRAPHICS and HP98770 option files (shipped on the BASIC 3D Graphics Option Binary disc).
   - If you are loading them from the original discs, type:

     ```
     LOAD BIN "GRAPHICS:INTERNAL" EXECUTE
     LOAD BIN "HP98770:INTERNAL" EXECUTE
     ```
3. Load the DEGAUSS utility and press RUN.
   - If you are using the original flexible discs with which your system came, insert the disc labeled "BASIC Utilities Programs 1 of 3" in the disc drive and type:

     ```
     LOAD "DEGAUSS:INTERNAL" EXECUTE
     RUN
     ```
4. The message "`DEGAUSSING THE SCREEN.`"appears on the display as the program is executing. There is a three second delay and then the screen blanks. When the degaussing process is done, the message "`DEGAUSS is done.`" appears on the display.

DISC_BACKUP Utility

This program duplicates and transfers the entire contents of a disc to another mass storage media.

It can transfer information from one drive to another or duplicate a disc using one drive. A smaller disc can be copied to a larger disc but the larger disc then can only contain the same amount of information as the smaller. Any disc of information can be copied to a ¼ inch streaming tape but only one disc of information may reside on one tape.

Note

Because the information is copied sector by sector, a complete transfer must occur for the new media to contain valid information. If the transfer is incomplete, errors result.

Before using this program you should read the "Mass Storage Organization" and "Working with Files" chapters of this manual.

To run this program:

1. Load the BASIC Language System.
2. Load the MS option.
   - If you are using the original flexible discs with which your system came, insert the flexible disc labeled "BASIC Binaries" in the internal disc drive and type:

     `LOAD BIN "MS:INTERNAL"` **EXECUTE**
3. Load the appropriate optional drivers for your device(s). These are also shipped on the "BASIC Binaries" disc. The following table shows the available choices.

If you want to copy to or from this device:	Load this driver:
CS80	None
HP 82901	HPIB_FLEX
HP 82902	HPIB_FLEX
HP 8290x	HPIB_FLEX
HP 9885	HP9885
HP 9895	HPIB_FLEX
INTERNAL	None
MEMORY	MEMORY_VOL

4. Load the DISC_BACKUP utility.
   - If you are using the original discs, insert the disc labeled "BASIC Utilities 1 of 3" and type:

     `LOAD "DISC_BACKUP:INTERNAL"` **EXECUTE**
5. Make sure that the disc to be backed up and the media to receive the copy are on-line.
6. Press **RUN**.

The Current Status Menu

The following is an example of what is displayed after pressing RUN or the NEXT MENU softkey.

```
DISC_BACKUP Program  Rev 2.00

This program duplicates or transfers entire medias of information.

CURRENT STATUS FOR DEVICES:

Listed below is the current status for each device. To change any of the
devices listed below, press the corresponding key. When no further changes
are desired, press the  BEGIN BACKUP  key.

                         SOURCE -      :CS80,7,1



                         DESTINATION -  :CS80,7,1





Select an action and press the corresponding key.

                                                                      *
                       CHANGE   CHANGE            END            BEGIN
                       SOURCE   DESTIN           PROGRAM         BACKUP
```

This display shows you the device to be copied and the device to do the copying. These are known as the SOURCE and DESTINATION devices respectively. The default SOURCE and DESTINATION devices are both "`:CS80,7,1`" which is the internal flexible disc drive.

The four softkeys do the following.

CHANGE SOURCE – When you press this key, another menu, the SOURCE Device Specification Menu, is displayed for specifying a new SOURCE device.

CHANGE DESTIN – When you press this key, another menu, the DESTINATION Device Specification Menu, is displayed for specifying a new DESTINATION device.

END PROGRAM – Pressing this softkey causes the program to terminate, printing:

```
DISC_BACKUP is done.
```

BEGIN BACKUP – Pressing this softkey starts the copying of the SOURCE disc to the DESTINATION device.

Before copying, the program asks you if you want the copy to be verified. This means the program performs a read-after-write operation. This takes longer than a non-verified copy.

Next the program asks you if you want multiple copies of the source media. This means that after the first copy procedure is done, you can replace the first DESTINATION media with another and the same copy procedure occurs.

After this, the copy procedure begins.

The Device Specification Menu

Upon pressing the CHANGE SOURCE or CHANGE DESTINATION softkeys, the Device Description Menu is displayed. Here is an example of the display after pressing CHANGE SOURCE.

```
DEVICE SPECIFICATION
For the SOURCE DEVICE, `:CS80,7,1'
Press the keys as needed to change the appropriate portions of the
device specifier. When all changes have been made, press the NEXT MENU key.

                         Device Driver  CS80

                         Select Code    7

                         Subunit        1

                         Volume         None

                         Volume Label   None




Select an action and press the corresponding key.

                                                                      *
               CHANGE   CHANGE    CHANGE   CHANGE  CHANGE            NEXT
               DEVICE  SEL CODE  SUBUNIT   VOLUME  LABEL             MENU
```

`Device Driver` specifies the type of disc to access.

`Select Code` specifies the select code and the address at which the device is found. This includes the primary and secondary addresses.

`Subunit` specifies which drive to use if there are multiple units at the same select code.

`Volume` specifies which volume of a subunit is accessed.

`Volume Label` specifies the volume label and password. The format of this is: `LABEL` *volume label*< > where the angle brackets are optional and indicate that a password has been specified.

The six softkeys do the following.

CHANGE DEVICE – When you press this key, the following occurs.

For each device, SOURCE and DESTINATION, a menu of the possible choices is displayed. A selection is made by using the up and down arrow keys or the soft keys labeled UP and DOWN to move the selection indicator. The final choice is made by pressing RETURN or the softkey labeled RETURN.

The menus for driver selection contain the following choices:

INTERNAL	This choice refers to the internal flexible disc if present. If the flexible disc is not present, then INTERNAL refers to the internal hard disc.

MEMORY	Memory can be used as a mass storage device by selecting a memory volume specifying which of the 32 possible volumes is to be accessed. These 32 volumes are numbered 0 thru 31. You must initialize the meory either within the program or by executing the following statement: `INITIALIZE ":MEMORY,0,`memory volume`,`size where memory volume specifies which memory volume is to be initialized; size specifies how many 256 byte records the memory volume may contain.

Note

To duplicate a disc using one drive, MEMORY is not needed as a device. The SOURCE and DESTINATION devices should have the same media specifier.

CS80	This choice refers to any of the CS80 discs. These discs include HP 7908, HP 7911, HP 7912, HP 7914, internal flexible disc, and the internal hard disc.
HP9895 HP9885 HP82901	These choices refer to the devices they name. The specification of the other fields for these devices is done using the Device Specification Menu described above.
OTHER	If you wish to specify a device not previously listed, choose this menu item. You are then requested to input the device specifier. The next menu item prompts you to input the select code, drive number and volume number.
SPECIFYING DEVICE AND PATH	With this option, you can input the entire file specifier at one time. The path, device driver, select code, subunit, volume number, volume label and passwords, as needed, are input together. The entries made here can be modified as needed using the Device Specification Menu.

CHANGE SEL CODE – Upon pressing this softkey, you are asked to input the select code of the device. The default is 7.

CHANGE SUBUNIT – Upon pressing this softkey, you are asked to input the subunit of the device. The default is 0.

CHANGE VOLUME – Upon pressing this softkey, you are asked to input the volume of the device to use. The default is 0.

CHANGE LABEL – Upon pressing this softkey, you are asked to input the volume label of the device to use. After this, you are asked for the password to use. The volume password is never listed by the program. Its existence is indicated by angle brackets after the volume label.

NEXT MENU – This softkey sends you back to the Current Status Menu.

DISC_COPY Utility

This utility has two functions: it can copy files from one disc to another and it can purge a file or set of files on a disc.

More specifically, this utility can copy all files in a directory or subtree. You can also selectively copy files by specifying a device and a directory path.

If you are copying from one SDF volume to another you can preserve the file structure. This means that the directory structure is transferred from the SOURCE device to the DESTINATION device; a subtree can be copied from one directory to another. Files which are at different levels on the SOURCE device have the same relationships on the DESTINATION device.

You can also copy without regard for the file structure. This means that all files copied are placed at the same level in the DESTINATION directory.

You can also purge all files on a disc, purge all files in a subtree, or purge files selectively.

The following restrictions are made by the program.

1. Although the program copies or purges any number of files, they are copied/purged in blocks of 160.
2. Files under a directory that is READ protected are not copied or purged unless the volume password is present or the directory password is supplied through a path as part of the device specifier.
3. Files lower than six levels in an SDF volume cannot be copied or purged.
4. A directory cannot be purged unless it contains no files.
5. When copying to a LIF volume, the length of the file name being copied must be ten or less characters long, otherwise the file cannot be copied. No directory files can be copied.
6. When copying to a 9845 formatted volume, the length of the file name being copied must be six or less characters long, otherwise the file cannot be copied. No directory files can be copied.

Before using this program, you should read the "Mass Storage Organization" and "Working with Files" chapters of this manual.

To run the DISC_COPY program:

1. Load the BASIC Language System.
2. Load the MS option.
   - If you are using the original flexible discs with which your system came, insert the flexible disc labeled "BASIC Binaries" in the internal disc drive and type:

     `LOAD BIN "MS:INTERNAL"` **EXECUTE**

3. Load the appropriate optional drivers for your device(s). These are also shipped on the "BASIC Binaries" disc. The following table shows the available choices.

If you want to purge files on or copy to or from this disc:	**Load this driver:**
CS80	None
HP 82901	HPIB_FLEX
HP 82902	HPIB_FLEX
HP 8290x	HPIB_FLEX
HP 9885	HP9885
HP 9895	HPIB_FLEX
INTERNAL	None
MEMORY	MEMORY_VOL

4. If you are working with LIF volumes, load the option, LIF_DISC.

   - If you are using the original flexible discs, type:

   ```
   LOAD BIN "LIF_DISC:INTERNAL" (EXECUTE)
   ```

   If you are working with 9845 formatted volumes, load the option, 9845_DISC.

   - If you are using the original flexible discs, type:

   ```
   LOAD BIN "9845_DISC:INTERNAL" (EXECUTE)
   ```

5. Load the DISC_COPY utility.

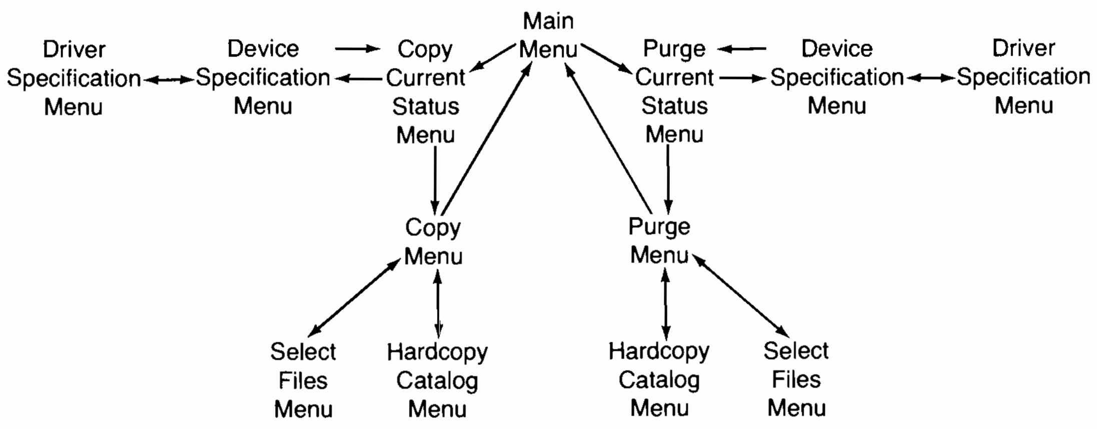
   - If you are using the original discs, insert the disc labeled "BASIC Utilty Programs 1 of 3" and type:

   ```
   LOAD "DISC_COPY" (EXECUTE)
   ```

6. Make sure that the disc whose files are to be copied or purged is on-line.
7. Press (RUN).

Program Flow

The following diagram shows the flow of the softkey menus in this utility.



Each of the menus is explained in the following sections.

Main Menu

This is the first menu that appears after you press RUN and represents the MAIN program. The following is displayed.

```
Select an action and press the corresponding key.

                                                                   *
          COPY           PURGE                            EXIT
                                                          PROGRAM
```

The three softkeys have the following functions.

COPY – Pressing this softkey specifies that you want to do a file copy and invokes the Current Status Menu.

PURGE – Pressing this softkey specifies that you want to do a file purge and invokes the Current Status Menu.

EXIT PROGRAM – This key causes the program to terminate, printing:
`DISC_COPY Done`

Current Status Menu

The following is an example of the display when this menu is invoked.

```
CURRENT STATUS FOR DEVICES:

Listed below is the current status for each device. To change any of the
devices listed below, press the corresponding key. When no further changes
are desired, press the NEXT MENU key.
                     SOURCE -       :CS80,7,1,0


                     DESTINATION -  :CS80,7,0,0


                     OVERFLOW -     None


                     PRINTER -      Internal

Select an action and press the corresponding key.

                                                                   *
      CHANGE   CHANGE          CHANGE   CHANGE             NEXT
      SOURCE   DESTIN          OVERFLOW PRINTER            MENU
```

This display shows you the device whose files are to be purged or which is to be copied and the device to do the copying. The first device is known as the SOURCE device and the second device is the DESTINATION device. It may also show an OVERFLOW device and a PRINTER.

The display contains the current values for each device. If COPY was selected the status of each of the devices is shown. If PURGE was selected in the Main Menu, only the the status of the SOURCE device and the PRINTER is shown on the display. These are updated as the values for the devices are changed.

The five softkeys do the following.

CHANGE SOURCE – Pressing this softkey invokes the Device Specification Menu for the SOURCE device. Use this key when you want to change the file specifier of the file(s) to be copied or purged. The default SOURCE device is `":CS80,7,1"`.

CHANGE DESTIN – Pressing this softkey invokes the Device Specification Menu for the DESTINATION device. Use this key when you want to change the file specifier of the directory to receive the copied files. The default DESTINATION device is `":CS80,7,0"`.

CHANGE OVERFLOW – Pressing this softkey invokes the Device Specification Menu for the OVERFLOW device. This specification is only important when copying files and the original DESTINATION device is full. If the original DESTINATION device becomes full and there are still files to be copied, the remainder of the files are copied to the OVERFLOW device. The default value for this device is NONE.

CHANGE PRINTER – Pressing this softkey invokes the Device Specification Menu for the PRINTER device. This device is used to specify where hardcopy messages are printed. These messages include the number of files copied or purged, the reasons why files were not copied or purged, the source and destination files as they are being copied (if the LOG feature is turned on as files are being copied or purged), and the hard copy catalogs if they are requested. The initial value for this specification is "INTERNAL". This value refers to the optional internal thermal printer.

NEXT MENU – Pressing this softkey invokes the Copy Menu or Purge Menu depending upon whether the COPY or PURGE softkey was pressed in the Main Menu. Before proceeding to the Copy Menu or Purge Menu, the program checks any unchanged devices. If the specification for SOURCE device was not changed, the program then checks for the device and reads the catalog. If any errors occur, the program returns to the Current Status Menu. If the DESTINATION device or the OVERFLOW device was not changed, each is checked to make sure that it can be accessed. If any errors occur, the program returns to the Current Status Menu.

Device Specification Menu

This menu is used after a device has been selected and further specifications are needed. The following is an example of the display when this menu is invoked.

```
DEVICE SPECIFICATION
For the SOURCE DEVICE, `:CS80,7,1,0'
Press the keys as needed to change the appropriate portions of the
device specifier. When all changes have been made, press the NEXT MENU key.

                         Device Driver   CS80

                         Select Code     7

                         Subunit         1

                         Volume          0

                         Volume Label    None

                         Path            None


                         Subset          None

Select an action and press the corresponding key.

                                                                             *
|| CHANGE || CHANGE  || CHANGE  || CHANGE ||| CHANGE || CHANGE || CHANGE || NEXT ||
|| DEVICE ||SEL CODE || SUBUNIT || VOLUME ||| LABEL  || PATH   || SUBSET || MENU ||
```

This display shows you the current file specifier or media specifier for the device chosen in the Current Status Menu.

`Device Driver` specifies the type of device to access.

`Select Code` specifies the select code and address at which the device is found. This includes the primary and secondary addresses.

`Subunit` specifies which drive to use if there are multiple units at the same select code.

`Volume` specifies which volume of a subunit is accessed.

`Volume Label` specifies the volume label and password. The format of this is: `LABEL` *volume label*< > where the angle brackets are optional and indicate that a password has been specified.

The eight softkeys do the following.

- **CHANGE DEVICE** – Pressing this softkey invokes the Driver Specification Menu. Do this when you want to change the type of device specified in this menu.
- **CHANGE SEL CODE** – Upon pressing this softkey, you are asked to input the select code of the device. The default is 7.

CHANGE SUBUNIT – Upon pressing this softkey, you are asked to input the subunit of the device. The default is 0.

CHANGE VOLUME – Upon pressing this softkey, you are asked to input the volume of the device to use. The default is 0.

CHANGE LABEL – Upon pressing this softkey, you are asked to input the volume label of the device to use. After this, you are asked for the password to use. The volume password is never listed by the program. Its existence is indicated by angle brackets after the volume label.

CHANGE PATH – Upon pressing this softkey, you are asked to input a string which is used to determine what files to copy or purge. Only the files that begin with that specific string can be copied or purged. If all files available are desired, this string should be left blank. If there are directories in the volume, they are always selected for easy recognition of the directory to which the file belongs. This softkey is only applicable if you pressed the CHANGE SOURCE softkey in the Current Status Menu.

CHANGE SUBSET – Upon pressing this key, you are asked to input a file pathname. If you are not interested in all of the files on a disc, but are only interested in the files contained in a specific directory, that directory can be specified. Since the program can handle a maximum of six levels, the directory pathname can have a maximum of five levels. If all the files on a disc are of interest, no pathname should be specified.

NEXT MENU – This softkey sends you back to the Current Status Menu after checking the accessability of the device specified.

Driver Specification Menu

For each device, SOURCE, DESTINATION, OVERFLOW and PRINTER, a menu of the possible choices is displayed. A selection is made by using the up and down arrow keys or the soft keys labeled UP and DOWN to move the selection indicator. The final choice is made by pressing RETURN or the soft key labeled RETURN.

The SOURCE, DESTINATION and OVERFLOW menus for driver selection contain the following choices:

INTERNAL — This choice refers to the internal flexible disc if present. If the flexible disc is not present, then INTERNAL refers to the internal hard disc.

MEMORY — Memory can be used as a mass storage device by selecting a memory volume specifying which of the 32 possible volumes is to be accessed. These 32 volumes are numbered 0 thru 31. You must initialize the memory either within the program or by executing the following statement:

```
INITIALIZE ":MEMORY,0,<memory volume>",<size>
```

where:
<memory volume> specifies which memory volume is to be initialized

<size> specifies how many 256 byte records the memory volume may contain

CS80	This choice refers to any of the CS80 discs. These discs include HP 7908, HP 7911, HP 7912, HP 7914, the internal flexible disc, and the internal hard disc.
HP9885 HP9895 HP82901	These choices refer to the devices they name. The specification of the other fields for these devices is done using the Device Specification Menu described above.
OTHER	If you wish to specify a device not previously listed, choose this menu item. You are then requested to input the device specifier. The next menu item prompts you to input the select code, drive number and volume number.
SPECIFYING DEVICE AND PATH	With this option, you can input the entire file specifier at one time. The path, device driver, select code, subunit, volume number, volume label and passwords, as needed, are input together. The entries made here can be modified as needed using the Device Specification Menu.
NONE	This choice appears only for the OVERFLOW device specification. Choosing this as the OVERFLOW device implies that when copying, if the DESTINATION device becomes full and there are more files, the files not copied are printed with an device overflow error message.

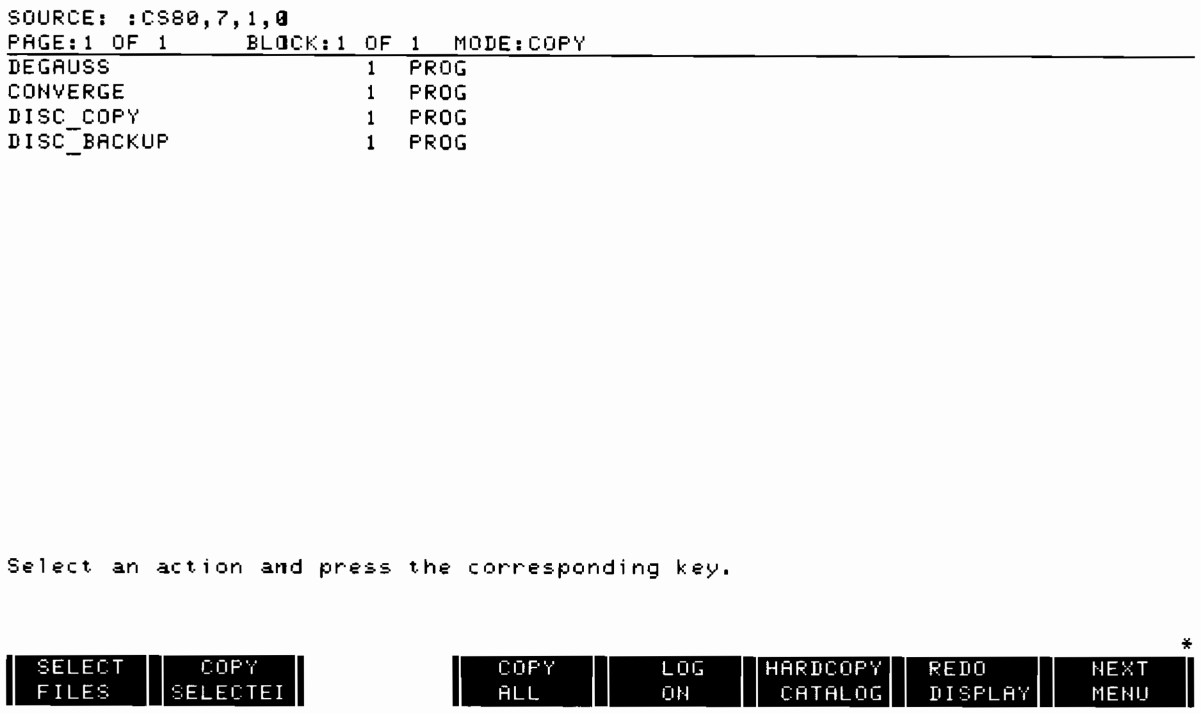
The choices for the Printer Menu and their meanings are:

INTERNAL	This specifies that the optional internal thermal printer is to be used.
EXTERNAL	This specifies that an external printer is to be used. You then input the select code for the printer.
NONE	This choice implies that no printer is available and all messages are printed to the display.

After a device has been selected, you are returned to the Device Specification Menu.

Copy Menu

The following is an example of what is displayed when this menu is invoked.



```
SOURCE: :CS80,7,1,0
PAGE:1 OF 1      BLOCK:1 OF 1  MODE:COPY
DEGAUSS                    1  PROG
CONVERGE                   1  PROG
DISC_COPY                  1  PROG
DISC_BACKUP                1  PROG









Select an action and press the corresponding key.
```

At the top of the display the header shows the current file specifier of the SOURCE device and the position within that device as well as the current page, the total number of pages, the current block and the total number of blocks. The rest of the display contains the page of files specified by the current block and page. For each file,there is the name of the file, the level of the file in the directory structure, the type of the file and whether the files are to be copied or purged. If no selection of files have been made then this column is blank.

With the Copy Menu, you can select a number of options before copying files. At this point the SOURCE, DESTINATION, OVERFLOW and PRINTER should have already been selected and specified.

The following is an explanation of the keys and their functions available in this menu.

SELECT FILES – When this key is pressed, the Select Files Menu is invoked.

COPY SELECTEI – After files have been selected for copying, this key causes the files to be copied.

COPY ALL – Pressing this key causes all the files specified by the SOURCE device to be copied to the DESTINATION device.

LOG ON – This softkey turns file logging on or off. With the log feature you can obtain a hardcopy listing of the files copied as they are being copied. The log must be turned on before the copying starts. The printer must be set at something other than NONE to obtain a log. If the log feature is turned off, the LOG ON key is displayed to enable you to turn the log on. If the log is turned on, the LOG OFF key is displayed so that the log feature may be turned off.

HARDCOPY CATALOG – This softkey invokes the Hardcopy Catalog Menu.

REDO DISPLAY – If the display is disturbed while the program is running, this softkey brings the display back.

NEXT MENU – When this softkey is pressed, the program returns to the Main Menu.

Purge Menu

The following is an example of what is displayed when this menu is invoked.

```
SOURCE: :CS80,7,1,0
PAGE:1 OF 1      BLOCK:1 OF 1  MODE:PURGE
DEGAUSS                    1  PROG
CONVERGE                   1  PROG
DISC_COPY                  1  PROG
DISC_BACKUP                1  PROG









                                                                          *
 SELECT   PURGE                 PURGE     LOG    HARDCOPY   REDO     NEXT
 FILES   SELECTEI                ALL      ON     CATALOG   DISPLAY   MENU
```

At the top of the display the header shows the current file specifier of the SOURCE device and the position within that device as well as the current page, the total number of pages, the current block and the total number of blocks. The rest of the display contains the page of files specified by the current block and page. For each file, there is the name of the file, the level of the file in the directory structure, the type of the file and whether the files are to be copied or purged. If no selection of files has been made then this column is blank.

With the Purge Menu, you can select a number of options before purging files. At this point the SOURCE of the files to be purged and the PRINTER should have already been selected and specified.

The following is an explanation of the softkeys and their functions.

SELECT FILES – When this softkey is pressed, the Select Files Menu is invoked.

PURGE SELECTEI – After files have been selected for purging, this key causes the files to be purged.

PURGE ALL – Pressing this softkey causes all the files specified by the SOURCE device to be purged, regardless of block.

LOG ON – This softkey turns file logging on or off. With the log feature you can obtain a hardcopy listing of the files as they are purged. The log must be turned on before the purging is started. The printer must be set at something other than NONE to obtain a log. If the log feature is turned off, the LOG ON key is displayed to enable you to turn the log on. If the log is turned on, the LOG OFF key is displayed so that the log feature may be turned off.

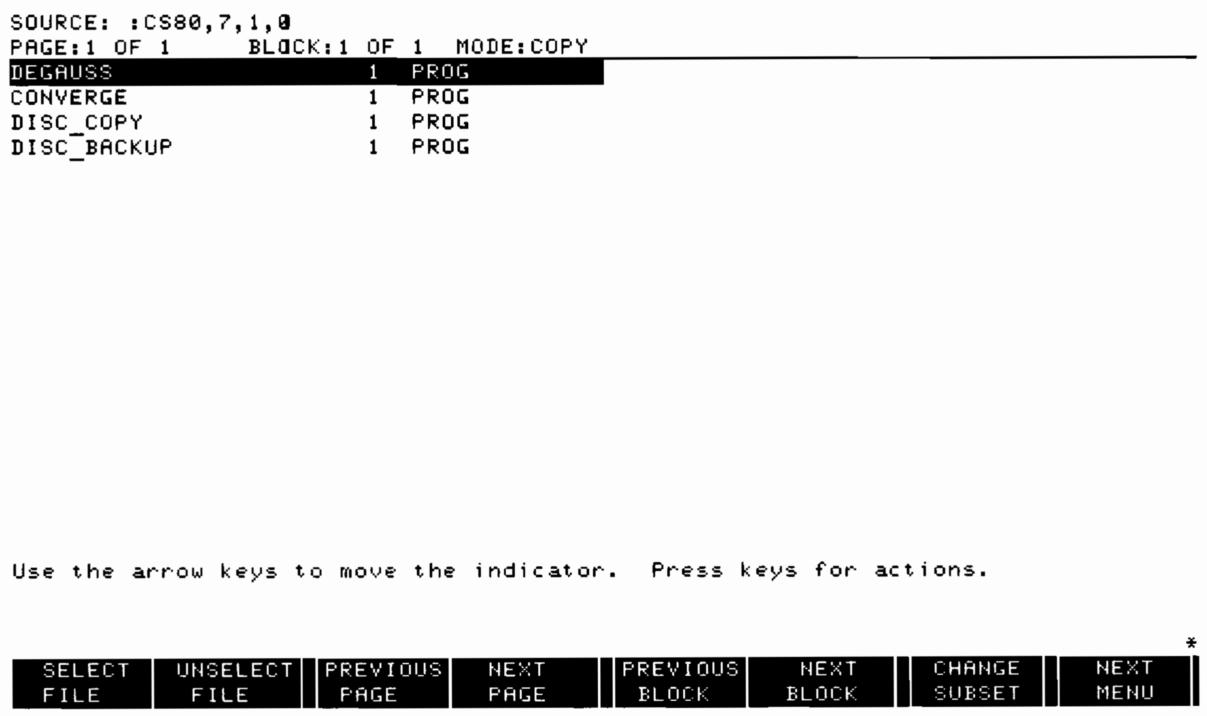
HARDCOPY CATALOG – This softkey invokes the Hardcopy Catalog Menu.

REDO DISPLAY – If the display is disturbed while the program is running, this softkey brings the display back.

NEXT MENU – When this softkey is pressed, the program returns to the Main Menu.

Select Files Menu

The following is an example of what is displayed when this menu is invoked.



```
SOURCE: :CS80,7,1,0
PAGE:1 OF 1     BLOCK:1 OF 1  MODE:COPY
DEGAUSS                   1  PROG
CONVERGE                  1  PROG
DISC_COPY                 1  PROG
DISC_BACKUP               1  PROG




Use the arrow keys to move the indicator.  Press keys for actions.

                                                                   *
 SELECT   UNSELECT  PREVIOUS   NEXT   PREVIOUS   NEXT    CHANGE    NEXT
 FILE     FILE      PAGE       PAGE   BLOCK      BLOCK   SUBSET    MENU
```

This menu is used to pick which files on the SOURCE device are to be copied or purged.

The eight softkeys do the following.

SELECT FILE – Pressing this key causes the file shown by the indicator to be selected for copying or purging. Whether the file has been selected for copying or purging is indicated on the same line as the file. The indicator is then moved to the next file down.

The file selector may also be moved by using the arrow keys. These keys only move the selector within the same page. The function of these keys is to move the indicator in the direction of the arrow.

UNSELECT FILE – Pressing this key causes the file shown by the indicator to be unselected. This means that if the file was selected for copying or purging, it is no longer selected. If a file was not selected for copying or purging then nothing is changed. The indicator is then moved to the next file down.

PREVIOUS PAGE – Because there are only 32 files on the display at one time, NEXT PAGE causes the next page of files to be printed to the display. Within one block there are a maximum of 160 files or 5 pages of files. The NEXT PAGE softkey can only get new pages within the same block. To get pages other than those in the current block, either the PREVIOUS BLOCK or NEXT BLOCK softkey must be pressed. The indicator specifying which page is the current page and how many pages there are in the block is located at the top of the display.

NEXT PAGE – This key is the complement of the NEXT PAGE key. Pressing this key causes the previous page of files to be printed to the display. It functions under the same constraints as the NEXT PAGE key.

PREVIOUS BLOCK – Pressing this key causes the next set of 160 files on the disc to be made available for copy or purge selection by the user. The number of blocks available is printed at the top of the display in the header. When a new block is selected, any files that were previously selected in another block are no longer available. This means that if files are to be copied from multiple blocks, each block of files must be brought into memory and the desired files must be selected and copied before the next block of files is read.

NEXT BLOCK – Pressing this key, causes the previous block of 160 files to be read. The same restrictions listed under NEXT BLOCK apply.

CHANGE SUBSET – A SUBSET is used to build a catalog containing the files that begin with a particular string. The directories on a disc always appear even if they do not begin with the specified string because if the directories did not appear you could not tell which directory the file was under. Pressing this key allows you to change the SUBSET specified. It may be changed to another SUBSET or may be changed to none. In the case of no SUBSET, all files on the on the disc are available for selection.

NEXT MENU – Pressing this key returns the program back to the Copy Menu or the Purge Menu depending on which one called the Selection Menu.

Hardcopy Catalog

This menu allows you to obtain a hardcopy listing of the files on the disc. At this point, the SOURCE, DESTINATION and OVERFLOW devices have been selected. A catalog may be obtained for any of the three devices.

The five softkeys do the following.

CATALOG SOURCE – Pressing this softkey causes a catalog of the SOURCE device to be printed on the currently designated printer. If there is no printer designated, then a message is printed to the CRT showing this.

CATALOG DESTIN – Pressing this softkey causes a catalog of the DESTINATION device to be printed. If there is no printer designated, then a message is printed to the CRT showing this.

CATALOG OVERFLOW – Pressing this softkey causes a catalog of the OVERFLOW device to be printed. If there is no printer, a message is printed to the CRT showing this.

CHANGE PRINTER – Pressing this softkey invokes the Device Selection Menu for the PRINTER.

NEXT MENU – Pressing this softkey returns the program to the Copy or Purge Menu depending on which one you were in.

MAG_TAPE_COPY Utility

This program moves files to and from the HP 7970 or the HP 7971 tape drive.

The program capabilities are divided into three categories: BACKUP - copying information to tape; RETRIEVE - copying information from tape to disc; and CATALOG which produces a listing of the blocks and files contained on a tape.

There are two types of BACKUP. First, files can be selected for backup by selecting a maximum of 160 files. These files once selected can be copied to tape. Second, all the files on a disc can be copied to tape in 160 file blocks.

Similarly there are two types of RETRIEVE. First, files can be selected from a tape block by selecting the files to be retrieved and then retrieving them. Secondly, all the files in a particular tape block can be retrieved.

Lastly, tapes can be cataloged. Either one block or the entire tape can be cataloged at one time.

Note that files written with the HP 9845 Tape Utilities can be retrieved using the MAG_TAPE_COPY program but these files are then useable only with the HP 9000 Series 500 Model 520 BASIC Language System.

Before using this program, read the "Mass Storage Organization" and "Working with Files" chapters of this manual.

In addition, if you intend to use a tape that contains old data to be overwritten, execute the following.

```
ASSIGN @Tape Drive I/O Pathname TO Address;DRIVER "HP7970"
CONTROL @Tape Drive I/O Pathname, 105; 0
CONTROL @Tape Drive I/O Pathname, 102; 100
```

This rewinds the tape and erases the first twenty-five feet, allowing a new header to be placed on the tape.

To run the MAG_TAPE_COPY program:

1. Load the BASIC Language System.
2. Load the MS option.
   - If you are using the original flexible discs with which your system came, insert the flexible disc labeled "BASIC Binaries" in the internal disc drive and type:

     ```
     LOAD BIN "MS:INTERNAL" (EXECUTE)
     ```
3. Load the IO option.
   - If you are using the original discs, type:

     ```
     LOAD BIN "IO:INTERNAL" (EXECUTE)
     ```
4. Load the HP7970
   - If you are using the original discs, type:

     ```
     LOAD BIN "HP7970:INTERNAL" (EXECUTE)
     ```
5. Load the appropriate optional drivers for your device(s). These are also shipped on the "BASIC Binaries" disc. The following table shows the available choices.

If you want to copy to or from this disc:	**Load this driver:**
CS80	None
HP 82901	HPIB_FLEX
HP 82902	HPIB_FLEX
HP 8290x	HPIB_FLEX
HP 9885	HP9885
HP 9895	HPIB_FLEX
INTERNAL	None
MEMORY	MEMORY_VOL

6. If you are working with LIF volumes, load the option, LIF_DISC.
   - If you are using the original flexible discs, type:

     ```
     LOAD BIN "LIF_DISC:INTERNAL" (EXECUTE)
     ```

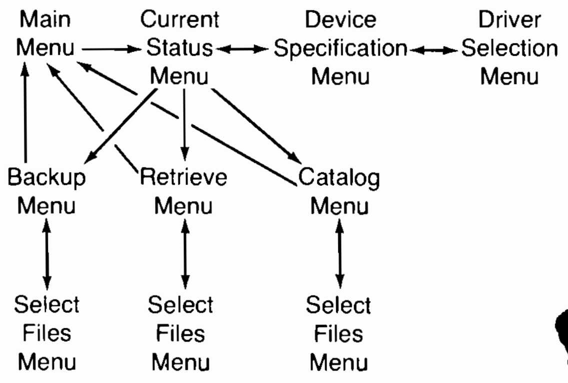
   If you are working with 9845 formatted volumes, load the option, 9845_DISC.
   - If you are using the original flexible discs, type:

     ```
     LOAD BIN "9845_DISC:INTERNAL" (EXECUTE)
     ```
7. Load the MAG_TAPE_COPY utility.
   - If you are using the original discs, insert the disc labeled "BASIC Utilities 1 of 3" and type:

     ```
     LOAD "MAG_TAPE_COPY:INTERNAL" (EXECUTE)
     ```
8. Make sure the disc and tape drive to be used are on-line.
9. Press (RUN).

Program Flow

The following diagram shows the flow of the softkey menus in this utility.



Each of the menus is explained in the following sections.

Main Menu

This is the first menu that appears after you press (RUN). The following is displayed.

```
SELECT AN ACTION

Pick the action you wish to perform.

                         BACKUP    - copy files from a disc to tape
                         RETRIEVE  - copy files from tape to disc
                         CATALOG   - produce tape table of contents
                         EXIT      - exit program










Use UP(↑) and DOWN(↓) keys to move indicator. Press RETURN to make a selection.

                                                                                   *
                    UP↑        DOWN↓                 RETURN
```

The four selectable actions have the following functions.

BACKUP - specifies that you want to copy files from a disc to a tape.

RETRIEVE - specifies that you want to copy files from a tape to a disc.

CATALOG - specifies that you want to catalog a tape.

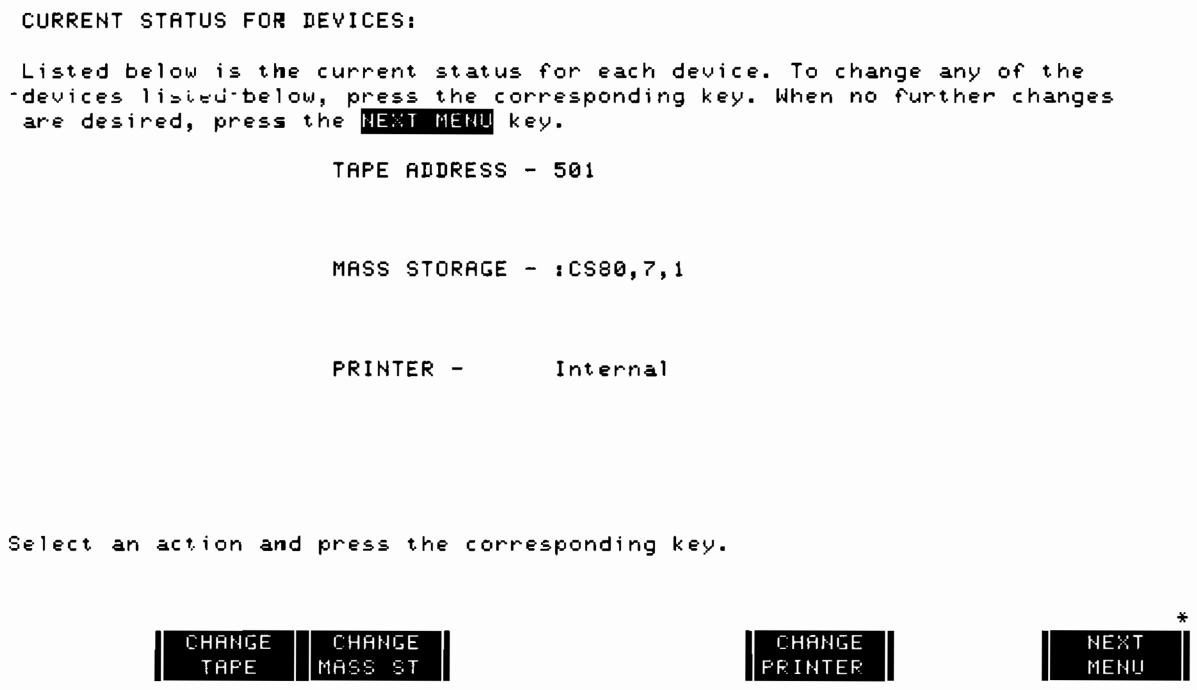
EXIT - causes the program to terminate.

To make a choice from the above list, move the indicator using the labelled keys or the the up and down arrow keys. To select the entry shown by the indicator, either press the RETURN key or the soft key labelled RETURN.

After pressing RETURN or RETURN, the program invokes the Current Status Menu.

Current Status Menu

The following is an example of the display when this menu is invoked.



```
 CURRENT STATUS FOR DEVICES:

 Listed below is the current status for each device. To change any of the
-devices listed-below, press the corresponding key. When no further changes
 are desired, press the NEXT MENU key.
                          TAPE ADDRESS - 501


                          MASS STORAGE - :CS80,7,1


                          PRINTER -      Internal




 Select an action and press the corresponding key.

                                                                          *
           CHANGE   CHANGE                          CHANGE           NEXT
            TAPE    MASS ST                         PRINTER          MENU
```

This display shows you the current address of the tape device, the current media specifier of the disc device and the current printer device, if any. If BACKUP or RETRIEVE was selected, the status of all the devices is shown. If CATALOG was selected on the original menu, then only the the status of the TAPE drive and the PRINTER is shown on the display. These are updated as the values for the devices are changed.

The four softkeys do the following.

- CHANGE TAPE – Upon pressing this softkey, the program asks you to input the new tape address.
- CHANGE MASS ST – Pressing this softkey invokes the Device Specification Menu.
- CHANGE PRINTER – Pressing this softkey invokes the Printer Menu.
- NEXT MENU – Pressing this softkey invokes either the Backup Menu, the Retrieve Menu or the Catalog Menu, depending upon which function was selected in the Main Menu.

Device Specification Menu

This menu is used after a device has been selected and further specifications are needed. The following is an example of the display when this menu is invoked.

```
DEVICE SPECIFICATION
For the MASS STORAGE DEVICE:":CS80,7,1"
Press the keys as needed to change the appropriate portions of the
device specifier. When all changes have been made, press the NEXT MENU key.

                        Device Driver   CS80

                        Select Code     7

                        Subunit         1

                        Volume          None

                        Volume Label    None

                        Path            None


                        Subset          None

Select an action and press the corresponding key.

                                                                          *
| CHANGE | CHANGE   | CHANGE  | CHANGE | | CHANGE | CHANGE | CHANGE | NEXT |
| DEVICE | SEL CODE | SUBUNIT | VOLUME | | LABEL  | PATH   | SUBSET | MENU |
```

`Device Driver` specifies the type of device to access.

`Select Code` specifies the select code and address at which the device is found. This includes the primary and secondary addresses.

`Subunit` specifies which drive to use if there are multiple units at the same select code.

`Volume` specifies which volume of a subuit is accessed.

`Volume Label` specifies the volume label and password. The format of this is: `LABEL` *volume label*< > where the angle brackets are optional and indicate that a password has been specified.

Path allows the specification of a directory in conjunction with the the actual device. If you are not interested in all of the files on a disc, but are only interested in the files contained in a specific directory, that directory can be specified. This path can only be five levels deep, because the program can handle at most six levels. If all the files on a disc are of interest, no path should be specified. This field is also used to copy files to different levels in the directory structure.

Subset allows only the files that begin with a specific string to be available for BACKUP. If all files available are desired, this field should be left blank. If there are directories on the device, they are selected all the time so that you can tell which directory a file belongs to. This field is only applicable to the MASS STORAGE DEVICE.

The eight softkeys do the following.

CHANGE DEVICE – Pressing the softkey invokes the Driver Specification Menu. Do this when you want to change the type of device specified in this menu.

CHANGE SEL CODE – Upon pressing this softkey, you are asked to input the select code of the device. The default is 7.

CHANGE SUBUNIT – Upon pressing this softkey, you are asked to input the subunit of the device. The default is 0.

CHANGE VOLUME – Upon pressing this softkey, you are asked to input the volume of the device to use. The default is 0.

CHANGE LABEL – Upon pressing this softkey, you are asked to input the volume label of the device to use. After this, you are asked for the password to use. The volume password is never listed by the program. Its existence is indicated by angle brackets after the volume label.

CHANGE PATH – Upon pressing this key, you are asked to input a file pathname. If you are not interested in all of the files on a disc, but are only interested in the files contained in a specific directory, that directory can be specified. Since the program can handle a maximum of six levels, the directory pathname can have a maximum of five levels. If all the files on a disc are of interest, no pathname should be specified.

CHANGE SUBSET – Upon pressing this softkey, you are asked to input a string which is used to determine what files to operate upon. Only the files that begin with that specific string are backed up or retrieved. If all files available are desired, this string should be left blank. If there are directories in the volume, they are always selected for easy recognition of the directory to which the file belongs.

NEXT MENU – This softkey sends you back to the Current Status Menu.

Driver Selection Menu

For the MASS STORAGE device and the PRINTER, a menu of the possible choices is printed to the screen. A selection is made by using the up and down arrow keys or the soft keys labelled UP and DOWN to move the selection indicator. The final choice is made by pressing RETURN or the soft key labeled RETURN.

The Mass Storage Menu for driver selection contains the following choices.

INTERNAL	This refers to the internal flexible disc, if present; otherwise INTERNAL refers to the internal hard disc.
MEMORY	Memory can be used as a mass storage device by selecting a memory volume specifying which of the 32 possible volumes are to be accessed. These 32 are numbered 0 thru 31. The memory must also be initialized either within the program or by executing the following statement: `INITIALIZE ":MEMORY,0,`<memory volume>`",`<size> where <memory volume> specifies which memory volume is initialized and <size> specifies how many 256 byte records memory volume can contain
CS80	This choice refers to any of the CS80 discs. These discs include HP7908, HP7911, HP7912, HP 7914, the internal flexible disc and the internal hard disc.
HP9885 HP9895 HP82901	These choices refer to the the devices they name.
OTHER	If you wish to specify a device not previously listed, choose this menu item. You are then requested to input the device specifier. The next meu item prompts you to input the select code, drive number and volume number.
SPECIFYING DEVICE AND PATH	With this option, you can input the entire file specifier at one time. The path, device driver, select code, subunit, volume number, volume label and passwords as needed are input together. The entries made here can be modified as needed using the Device Specification Menu.

The choices for the Printer Menu and their meanings are:

INTERNAL	This specifies that the optional internal thermal printer is to be used.
EXTERNAL	This specifies that an external printer is to be used. You then input the select code for the printer.
NONE	This choice implies that no printer is available and all messages are printed to the display.

After a device has been selected, the program returns to the Device Specification Menu.

Backup Menu

This menu contains the choices for the different types of backup.

The four selectable actions have the following functions.

BACKUP ALL FILES	Each file specified by the current MASS STORAGE device is copied to tape in blocks of 160 files. Files in a read protected directory are not copied unless the volume password is specified. Directories and passwords are not copied to tape.

BACKUP SELECTED	The files already selected for backup are copied to tape.
SELECT FILES	When this softkey is pressed, the Select Files Menu is invoked.
EXIT	When this softkey is pressed, the program returns to the Main Menu.

Retrieve Menu

This contains choices for the different types of retrieval.

The four selectable actions have the following functions.

RETRIEVE ALL FILES	All of the files in the chosen tape block are copied to the current MASS STORAGE device.
RETRIEVE SELECTED	The files selected from the current tape block are transferred to the current MASS STORAGE device.
SELECT FILES	Selects files from the current tape block to be copied to disc.
EXIT	When this softkey is pressed, the program returns to the Main Menu.

Catalog Menu

This menu is used to select the different forms of cataloging.

The four selectable actions have the following functions.

WHOLE TAPE	Reads the entire tape and prints the volume header, block headers and file headers for the files contained in the blocks.
ONE BLOCK	The file headers are printed for the block specified.
NEW REEL	Mounts and catalogs a new reel of tape.
EXIT	When this softkey is pressed, the program returns to the Main Menu.

Select Files Menu

When this menu is invoked, the following is displayed.

```
MASS STORAGE:  BINS:CS80,7,0
PAGE:1  OF  2      BLOCK:1  OF  1   Mode:BACKUP
TRANSLATOR                1  PROG          HP7970                 1  BIN
MAG_TAPE_COPY             1  PROG          HPIB_FLEX              1  BIN
ASYNC_0000                1  PROG          CIPERLP                1  BIN
ASYNC_SOURCE              1  PROG          HP27110                1  BIN
IO                        1  BIN           IMAGE_DBM              1  BIN
MEMORY_VOL                1  BIN           DEGAUSS                1  PROG
MS                        1  BIN           CONVERGE               1  PROG
LIF_DISC                  1  BIN           DISC_COPY              1  PROG
9845_DISC                 1  BIN           DISC_BACKUP            1  PROG
HP9885                    1  BIN           BANNER                 1  PROG
HP27112                   1  BIN           SET_NON_VOL_MEM        1  PROG
XREF                      1  BIN           GRAPHICS               1  BIN
CAT_BOOT                  1  PROG          HP98770                1  BIN
BUILD_BOOT                1  PROG          HP97060                1  BIN
ERRORS_ENG                1  BIN           HP97062                1  BIN
SERIAL                    1  BIN           GUTIL                  1  PROG

Use the arrow keys to move the indicator.  Press keys for actions.

                                                                                 *
 SELECT  | UNSELECT || PREVIOUS |  NEXT  || PREVIOUS |  NEXT  || CHANGE ||  NEXT
  FILE   |   FILE   ||   PAGE   |  PAGE  ||  BLOCK   | BLOCK  || SUBSET ||  MENU
```

This menu is used to pick the files on the source device which are to be copied. This same menu is used for selection from either a tape block or MASS STORAGE device.

The eight softkeys do the following.

SELECT FILE – Pressing this key causes the file shown by the indicator to be selected for backup or retrieval. Whether the file has been selected for backup or retrieval is indicated on the same line as the file. The indicator is then moved to the next file down.

The file selector may also be moved by using the arrow keys. These keys only move the selector within the same page. The function of these keys is to move the indicator in the direction of the arrow.

UNSELECT FILE – Pressing this key causes the file shown by the indicator to be unselected. This means that if the file was selected for backup or retrieval, it is no longer selected. If a file was not selected for backup or retrieval then nothing is changed. The indicator is then moved to the next file down.

PREVIOUS PAGE – Because there are only 32 files on the display at a time, pressing this key causes the next set of 32 files to be printed to the display. Within one block there is a maximum of 160 files or 5 pages of files. The NEXT PAGE key retrieves only the new pages within the same block. To get pages other than those in the current block, either the PREVIOUS BLOCK or NEXT BLOCK key must be pressed. The indicator at the top of the display specifies which page is the current page and how many pages there are in the block.

NEXT PAGE – This key is the complement of the NEXT PAGE key. Pressing this key causes the previous page of files to be printed to the display. It functions under the same constraints as the NEXT PAGE key.

PREVIOUS BLOCK – Pressing this key makes available for selection the next set of 160 files on the disc. This key is only applicable to the MASS STORAGE device. The current block and the number of available blocks is printed at the top of the display in the header. When a new block is selected, any files that were previously selected in another block are no longer available. This means that if files are to be backed up from multiple blocks, each block of files must be brought into memory and the desired files must be selected and backed up before the next block of files is read. This key is not applicable for tape blocks.

NEXT BLOCK – Pressing this key, causes the previous block of 160 files to be read. The same restrictions listed under NEXT BLOCK apply. This key is not applicable for tape blocks.

CHANGE SUBSET – A SUBSET is used to build a catalog containing the files that begin with a particular string. The directories on a disc always appear even if they do not begin with the specified string so you can tell which directory the file is under. Pressing this key allows you to change the SUBSET specified to another SUBSET or to no SUBSET. In the case of no SUBSET, all files on the disc are available for selection. This key is not for tape blocks.

NEXT MENU – Pressing this key returns the program to the Backup Menu or to the Retrieve Menu, depending on which one called the Selection Menu.

Statements Accessing the Tape Drive

If the MAG_TAPE_COPY program does not satisfy your need, the following statements can be used to access the tape drive.

- To assign the pathname for the tape drive, use:

  ```
  ASSIGN @<pathname> TO <select code>;DRIVER<driver name>,<other attributes>
  ```

  where:

  <pathname> specifies the path name.

  <selectcode> specifies the select code and the primary address.

  <driver name> "HP7970" or HP7971".

  Refer to the ASSIGN @... statement in the BASIC Language Reference manual for more information on other attributes. These attributes include an input and an output conversion table so that EBCDIC tapes can be read and converted to ASCII.

  Here is an example of using ASSIGN @...

  ```
  ASSIGN @T TO 501; DRIVER "HP7970",BUFFERSIZE 2048
  ```

- To space across EOF marks, use:

  ```
  CONTROL @<pathname>,100;<repeat value>
  ```

  where:

  <pathname> is the pathname used in the ASSIGN @... statement.

  <repeat value> indicates how many EOF's will be spaced across. If <repeat value> is greater than zero the tape moves forward. If <repeat value> is less than zero the tape moves backward.

Here are some examples:

`CONTROL @T,100;2` spaces across two EOFs going forward,
`CONTROL @T,100;-2` spaces across two EOFs going backward.

- To space across records, use:

  `CONTROL @<pathname>,101;<repeat value>`

  where:

  <pathname> is the pathname used in the ASSIGN @... statement.

  <repeat value> specifies the number of records skipped. If <repeat value> is greater than zero, the tape moves forward. If <repeat value> is less than zero, the tape moves backward.

  For example:

  `CONTROL @T,100;2`

  spaces across two records going forward.

- To write a 3.75 inch gap, use:

  `CONTROL @<pathname>,102;<repeat value>`

  where:

  <pathname> is the <pathname> used in the ASSIGN @... statement.

  If <repeat value> equals zero means no gap is written. If <repeat value> is greater than zero, the specified number of gaps is written. For example:

  `CONTROL @T,102;40`

  writes 40 3.75 inch gaps. Note that this statement can be used to erase a tape, if security warrants it.

- To write end of file (EOF):

  `CONTROL @<pathname>,103;<repeat value>`

  where:

  <pathname> is the <pathname> used in the ASSIGN @... statement.

  If <repeat value> equals zero, no EOF is written. If <repeat value> is greater than zero, the specified number of EOF's is written.

- To read one or more records - backwards, use:

  `CONTROL @<pathname>,104;<repeat value>`

  where:

  <pathname> is the same name used in the ASSIGN @... statement.

  <repeat value> is the number of records to be read and is in the range -32 768 thru 0.

- To rewind the tape and set offline, use:

  `CONTROL @`<pathname>`,105;`<offline value>

  where:

  <pathname> is the name used in the ASSIGN @... statement.

  If <offline value> is equal to zero, the tape is just rewound. If <offline value> is not equal to zero, the drive is set offline.

- To transfer data from tape to a string, use:

  `ENTER @`<pathname>` USING"#,-K";`<data string>

  where:

  <pathname> is the name used in the ASSIGN @... statement.

  <data string> is the string into which the information from the tape is placed.

- To transfer data from a string to tape, use:

  `OUTPUT @`<pathname>` USING "@,-K";`<data string>

  where:

  <pathname> is the name used in the ASSIGN @: statement.

  <data string> is the variable which contains the information written to tape.

  Refer to the I/O Drivers section of the BASIC Language Reference manual for more information for the HP 7970 Tape Drive.

SET_NON_VOL_MEM Utility

This program resets two locations of non-volatile memory: the stack count and the default MASS STORAGE IS.

The stack count represents the number of finstrates that should be in your computer's processor stack. Every time your system is powered up, it counts the number of finstrates in the stack and compares this with the stack count in non-volatile memory. If fewer finstrates are found than expected, a finstrate has either been removed or has failed.

The default MASS STORAGE IS specifies what mass storage device is used at power up when a device is referred to implicitly in a mass storage statement. For example, suppose you execute:

```
COPY "File1" TO "File2"
```

Since no device is specified in the statement, the default mass storage device is searched for `File1`. You may also change the default mass storage device using the MASS STORAGE IS statement. This does not reset the value in non-volatile memory. This default is used until the machine is powered down. When the machine is powered up again, the value in non-volatile memory is the default.

To run the SET_NON_VOL_MEM program:

1. Load the BASIC Language System.
2. Load the SET_NON_VOL_MEM program.
   - If you are using the original flexible discs with which your system came, insert the flexible disc labeled "BASIC Utilities Programs 3 of 3" in the disc drive and type:

     `LOAD "SET_NON_VOL_MEM:INTERNAL"` (EXECUTE)
     (RUN)
3. After pressing (RUN), there are three soft key labels that appear.

RESET STACK – This softkey updates the non-volatile memory location that contains the number of finstrates.

SET MSI – After pressing this softkey, the program prompts you to input the new default mass storage device. The program then checks to see if the device is present. If it is not, an error occurs. If no error occurs, the non-volatile memory location containing the default media specifier is updated.

EXIT PROGRAM – Pressing this softkey causes the program to terminate.

TRANSLATOR Utility

The TRANSLATOR Utility automatically translates most of the statements in an HP 9835/45 BASIC program, which are not accepted by or whose meaning has changed on the HP 9000 Model 520, to statement(s) which produce the same effect.

Before you run the TRANSLATOR program, check the increment between line numbers in your HP 9835/45 program and if possible renumber (REN) the lines so that this increment is five or greater. If your program is too large to do this, renumber with the largest increment possible. The TRANSLATOR occasionally inserts one extra line and only rarely inserts three extra lines to provide an equivalent statement(s).

In addition you must have your HP 9835/45 program(s) on one of the following media.

HP 9885 Disc
HP 9895 Disc
HP 88140 Tape
HP 7970E Tape

To run the program:

1. Connect the mass storage device containing the programs to be translated to your new computer. If the files are on tape, move them to a disc. Use the MAG_TAPE_COPY utility if it is a 7970E tape.
2. Load the BASIC Language System.
3. Load the binaries from the "BASIC Binaries" disc necessary to access your HP 9835/45 file (These are typically 9845_DISC and HPIB_FLEX. It is also strongly recommended that you load ERRORS_ENG. The TRANSLATOR does not require options which ultimately allow your program to run, for example, the graphics option to translate a graphics program.).
4. Insert Disc 3 of the BASIC Utility Programs into the disc drive and type:

   `LOAD "TRANSLATOR",1` (EXECUTE)

   Loading the program takes about 20 seconds. When this process is completed the "*" at the lower right of your CRT disappears and the following messages appear.

   ```
   Copyright Hewlett-Packard 1983. All rights reserved.
   9845 BASIC to 9000 BASIC Translator Program Version 1.00
   ```

   The message `==>Initializing Translator` appears on your CRT as TRANSLATOR begins.
5. The program now interactively prompts you to answer the following questions.

   ```
   Enter source file name:
   ```

   Input the file specifier of the file you want to translate. The program responds with:

   ```
   ==>Getting source file size.
   ```

   and then asks:

   ```
   Line number of first source file line to translate
   (default--beginning of file):
   ```

   Input the line number in the file to be translated that you want the translator to start at. If you want the translator to start at the beginning of the file, just press (RETURN).

This line number need not match exactly a line number in your program. Translation begins with the first line whose number is greater than or equal to your specified line number. If you do specify a line number which results in any lines from your source file being skipped, the TRANSLATOR does not insert LEXICAL ORDER IS ASCII at the beginning of your output file. For more information, refer to LEXICAL ORDER statements in the "HP 9835/45 Compatibility" appendix of this manual.

6. Next the program requests:

```
Enter output file name:
```

Input the file specifier of the file to receive the translated program. The program responds with:

```
==>Creating output file.
```

The output file created is a DATA file that is twenty percent larger than the input file. This is to allow room for expansion due to translation in disc formats which do not have extendable files.

7. Next, the program asks:

```
Listing type?
```

and supplies three softkeys.

`Major changes` – produces a list of all lines in the program which affect program behavior and which require your attention upon translation.

`All changes` – produces a list of all lines changed upon translation

`Complete listing` – produces a list of all lines in your program whether they are changed or not upon translation.

8. Next, the program asks:

```
List to?
```

and provides six softkeys to choose the output device(s) to receive the translator listing upon translation.

`Internal Printer` – This softkey specifies the internal thermal printer as the output device.

`External Printer` – Upon pressing this softkey, the program asks you for a device selector for an external printer to use as the output device.

`File only` – Upon pressing this softkey, the program asks you for a file specifier of a file to use as the output device.

`File & CRT` – Upon pressing this softkey, the program asks you for a file specifier of a file to use as the output device along with the internal CRT.

`File & int print` – Upon pressing this softkey, the program asks you for a file specifier of a file to use as the output device along with the internal thermal printer.

`File & ext print` – Upon pressing this softkey, the program asks you for a device selector for an external printer to use as an output device. It then asks for a file specifier of a file to use as an additional output device.

If you select any option which includes a listing to a file, it is strongly recommended that you specify a file on a disc with a structured directory format (SDF). It is impossible to determine in advance how large the listing file will be, and an SDF file can then be extended as necessary.

9. Next, the program displays:

```
Select items you would like translator support in modifying:
```

In addition to the substitutions which the TRANSLATOR performs automatically, you can also have the program make substitutions for changed file names, mass storage unit specifiers (MSUS), select codes and certain key numbers in ON KEY statements.

Translating File Names

There are three primary reasons for making file name substitutions: first is for protect code or password handling; second is for accommodating a directory structure for an SDF disc; and third is to eliminate illegal characters.

If you select "File Names" you are prompted with:

```
Original file name:  (empty line to terminate)
```

You enter the file name you would like the TRANSLATOR to change for you, with no enclosing quotes. For example:

```
X RETURN
```

The TRANSLATOR then prompts you for its replacement and places the original file name in the keyboard input area of the display so you can modify it. For example, if you want to add the password "MINE" enter:

```
X<MINE> RETURN
```

Protected Files

When protected files are accessed on your new computer, the password is given with the file name enclosed in angle brackets "< >". (See **file specifier** in the glossary.) The sole exception to this rule is the PROTECT statement, which, in its simplest form is unchanged from the HP 9845 PROTECT statement. (See the PROTECT keyword in the BASIC Language Reference manual.) The TRANSLATOR does not automatically change formats, but on encountering any statement with a file protect code, except PROTECT, it drops the protect code and assumes that it will be taken care of by a user specified file substitution.

For example, with no file name replacement for "Myfile", the TRANSLATOR changes the statement:

```
RE-STORE "Myfile","Prtect"
```

to

```
RE-STORE "Myfile"
```

and issues a diagnostic that the password has been subsumed into the file name.

Assuming the sample substitution for "X" in the previous example, the TRANSLATOR would change:

```
    RE-STORE "X","MINE"
```

to

```
    RE-STORE "X<MINE>"
```

Note that the TRANSLATOR would then also make the following changes:

`RE-STORE "X","YOURS"` to `RE-STORE "X<MINE>"`

and

`LOAD "X",1` to `LOAD "X<MINE>",1`

In the first case the TRANSLATOR effectively changes the password; in the second case it adds a password which is not strictly necessary.

Directory Structures for an SDF Disc

Filename substitutions can be used to reference files which have been moved to some directory other than the root directory on an SDF disc. For example, if "X" is a protected DATA file which resides on device ":F8" and which will reside on ":CS80,7" in directory "Mydir" on your new computer, change `X` to `Mydir/X<password>` using the File Names softkey and `:F8` to `:CS80,7` using the MSUS softkey.

TRANSLATOR would change:

```
    ASSIGN #1 TO "X:F8",Return,"password"
```

to

```
    ASSIGN #1 TO "Mydir/X<password>:CS(0,7"; RETURN Return
```

Illegal Characters

The slash (/) and less than (<) were legal characters in a file name on the HP 9845, but are not on your new system. In your new system, the slash is used to indicate containment in a directory and the less than to begin a password. The TRANSLATOR does not flag occurrences of these characters.

MSUS

With mass storage unit specifier (MSUS) substitutions you can change an msus because a device is identified differently on your new computer or because a totally new type of device will be used.

Msus change requests are made in the same way as for file names. On each input, enter the leading colon and do not enclose the msus in quotes. For example:

```
    Original MSUS name:  (empty line to terminate)
    :T15 (RETURN)
```

The TRANSLATOR then prompts you for a replacement. You can enter, for example:

```
    :INTERNAL (RETURN)
```

Another example:

`:T15` can be changed to `:CS80,7`.

This causes references that formerly used the right hand tape cartridge to now use the internal Winchester disc drive.

`:F8` can be changed to `:CS80,7,1`.

References that used the external HP 9885M drive now use the internal flexible disc drive.

Changing :H8 to :HP9895,4 would still use an HP 9895 flexible disc drive but which is now on select code 4.

Only those references to file names or msus' which are a single string literal are looked at for replacement. Variables as well as concatenation of literals or a literal and a variable cause the TRANSLATOR to bypass file name and MSUS replacements.

Select Codes

If you press the SFK under "Select codes" two types of replacement are possible:

- interface select codes (ISC);
- individual device selectors.

Here are the default interface select code replacements used by the TRANSLATOR:

	Interface Select Codes		
Old	0	13	16
New	6	1	1

The rest of the interface select codes remain the same, for example,

`PRINTER IS 7,3` translates to `PRINTER IS 703`

If you specify changing the standard HPIB select code "7" on the HP 9845 to "5" on your new system:

`PRINTER IS 7,3` becomes `PRINTER IS 503`
`PRINTER IS 7,1` becomes `PRINTER IS 501`

You can also change the individual device address with the "Device Selector". For example, if you specify that "503" be changed to "202" then:

`PRINTER IS 5,3` becomes `PRINTER IS 202`.

In a situation where both an individual device selector and an ISC replacement apply, the individual device selector takes precedence. If you request that ISC "7" be changed to "5" and device selector "703" changed to "202", then "7,3" changes to "202" not "503".

ON KEY Re-map

Both the HP 9845 and your new system have 32 special function keys (SFKs) on the keyboard, numbered 0 thru 31, (16 thru 31 being the shifted versions of 0 thru 15). The HP 9845C and the HP 98770B and HP 98780B CRTs also have eight SFKs on the CRT bezel which are trappable by the ON KEY statement. The HP 9845C CRT SFKs are equivalent to keyboard keys 24-31, but on your new system they are equivalent to keys 0 thru 7. By selecting either CRT SFK under this re-map prompt you are asking the TRANSLATOR to automatically change all occurrences of ON KEY #n to ON KEY (n MODULO 24). This effectively maps key numbers in the range 24 thru 31 into the range 0 thru 7.

ON KEY re-map is useful if your program uses CRT SFKs, but not if your program expects to use the SHIFTed keyboard keys.

No More

The program cycles back to the message:

```
    Select items you would like translator support in modifying:
```

until you press the SFK under "No More" or all four replacement types have been requested.

After this the TRANSLATOR program begins to change the incompatible lines, indicating on your CRT which line is currently being translated.

Translator Diagnostics

If it is possible that the meaning of the translated statement differs slightly from the original, the TRANSLATOR issues a diagnostic message(s) with the changed line. If the TRANSLATOR can not convert a statement to some new equivalent, it comments the line out and in most cases, gives suggestions on how you can create create an equivalent statement sequence. Changes, diagnostics and suggestions are output to the current listing file and/or device. Some of these may refer to the appendix "Compatibility with the HP 9835/45" by the message:

```
    Refer to Differences/TRANSLATOR Actions Appendix for details.
```

The old line is indicated with a "#" immediately following the line number; the translated line has a blank following its line number. An example of a change which performs the identical operation follows:

```
    960# MAT Resources=ZER
    960  MAT Resources=(0)
```

An example of a TRANSLATOR action on a statement for which there is no equivalent is:

```
    410# REWIND ":T15"
    410 !==> 9000 REWIND ":T15"
    ***** REWIND--no equivalent
    ***** Line commented out.
```

The suggested modifications are listed below this line.

If the line has a label, the comment mark "!" is placed after the label. These lines are always marked with `!==>9000` and can therefore always be found easily using the FIND command in the editor.

There are two lists at the end of your diagnostic output.

1.
   ```
   The following items, which may affect program behavior, were
   encountered at least once during translation:
   ```

   A listing of these items follows. For example,

   ```
   ***** DROUND--Results may differ due to binary arithmetic.
   ***** GRID--Current Position is updated to intersection.
   ```

2.
   ```
   Relational equality or inequality operator found in the
   following lines. List includes =  <=  >=  <>, but excludes >
   and <. May be a concern due to use of binary instead of BCD
   arithmetic on the 9000.
   ```

   A list of line numbers follows. This list includes all lines where =, <=, >= or <> operators are found. In most cases the list only includes comparisons of numeric expressions. However, multiple assignment statements are also included and it is possible that some occurrences of string comparisons may be included.

The message "TRANSLATION COMPLETE" and the summary statistics, "changed mm of nnn source lines. Added xx lines", appear on your CRT when the translation process is finished.

The list of diagnostic messages and possible solutions for each line should help you to complete the final stage of altering your program.

Appendix B
HP9835/45 Compatibility

This appendix outlines the differences you may encounter when a program which was written for the HP 9835/45 is used on your new system. General considerations, the actions of the TRANSLATOR Utility program, graphics actions and an alphabetical keyword listing are described.

General Considerations

Some translation of your HP 9835/45 program is necessary because of changes in these major areas:

- a change from binary coded decimal (BCD) to binary arithmetic[1];
- a change in the order of evaluation of expressions[1];
- support for integer arithmetic[1];
- changed precedence of unary minus and NOT operators[1];
- significant new graphics capabilities;
- differences in select code formats;
- differences in image specifiers for IMAGE statements and ENTER...USING.

Select Code Formats

There is a single consistent method of HP-IB addressing for both device address on the bus and subcommands on your new system. For a precise definition of this format, see "device selector" in the Glossary. Many statements in the IO option may require modification to this new format. Such statements as PRINTER IS and PRINTALL IS are also affected. For example:

```
    PRINTER IS 7,3
```

is now:

```
    PRINTER IS 703
```

The TRANSLATOR utility makes the necessary select code format changes to your program if your addresses are always given by constants. It also makes many of the changes needed if they are variables.

IMAGE Statements

(also for PRINT...USING, OUTPUT...USING and ENTER...USING)

For the most part, these statements are compatible with the equivalent statement on the HP 9845. The outbound direction is completely compatible, except that the current EOL sequence is substituted for CR-LF at the end of the statement. The inbound direction, however, has undergone several changes, which are detailed in the following table.

1 Refer to the "Numeric Computations" and "Bit Manipulations" chapters for a discussion of these areas.

9845 Spec	New Equiv	Semantics
F	K	Freefield numeric input, decimal point radix
N	D	Fixed field numeric input, decpt radix
[n]G	[n−1]DR	Fixed field numeric input, comma radix (5G = 4DR, etc. 1G=1D)
T	K	Freefield string
+	−	Terminate statement on LF only
%	#	Terminate statement as soon as list satisfied

The following notations are used throughout:

RNEQ	remove, no equivalent;
SFK	special function key;
msus	mass storage unit specifier.

TRANSLATOR Utility Actions

This section describes the actions of the TRANSLATOR Utility program, which is on disc 3 of the BASIC Utility Programs. The implementation of the TRANSLATOR is outlined in the "Utilities" appendix of this manual.

File Names and Msus

In the discussion of the translation of specific keywords, the use of:

```
   <file spec>
→  <new file spec>
```

or any stand alone occurrence of an msus:

```
   <msus>
→  <new msus>
```

indicates that the translation occurs by the method described under File Name and Msus replacements in the TRANSLATOR Utility section of the "Utilities" appendix.

For statements where a password appears, the password is merely dropped, all necessary passwords must be incorporated into the file name using the method above. The single exception to this is PROTECT, where the password is in the same place on your new system.

The characters "<" and "/" were legal in file names on the HP 9835/45 but are illegal on your new system. The TRANSLATOR utility does not check for these.

The following set of statements require only translation of file names, msus' and passwords.

```
   COPY <file spec 1> TO <file spec 2>[,<password>]
→  COPY <new file spec 1> TO <new file spec 2>

   GET <file spec>[,<add line>[,<exec line>]]
→  GET <new file spec>[,<add line>[,<exec line>]]
```

LOAD <file spec>[,<exec line>]
→ LOAD <new file spec>[,<exec line>]

LOAD KEY <file spec>
→ LOAD KEY <new file spec>

MASS STORAGE IS <msus>
→ MASS STORAGE IS <new msus>

PROTECT <file spec>,<password>
→ PROTECT <new file spec>,<password>

PURGE <file spec> [,<password>]
→ PURGE <new file spec>

RENAME <file spec1> TO <file spec2> [,<password>]
→ RENAME <new file spec1> TO <new file spec2>

[RE-]SAVE <file spec>[,<password>][,<begin>[,<end>]]
→ [RE-]SAVE <new file spec>[,<begin>[,<end>]]

[RE-]STORE <file spec>[,<password>]
→ [RE-]STORE <new file spec>

STORE BIN <file spec>
→ STORE BIN <new file spec>

VOLUME DEVICES ARE <msus list>
→ VOLUME DEVICES ARE <new msus list>

Select Code and HP-IB Addressing

These are the notations and definitions for device selector replacements:

- <isc> interface select code in range of 0-16, no address allowed;
- <sc,ad> select code with optional address (This can be an <isc>, as above, or can also include HPIB address in the form:

  <isc>,<addr>

  where <addr> is the HPIB address);
- <dev> select code (This can be an <isc>, as above, or can also include HPIB address in the form:

  <isc> × 100 + <addr>

  where <addr> is the HPIB address);
- <scs> select code sequence. An <scs> can be an interface select code followed by zero or more HPIB addresses, with or without subcommands, or it can be a sequence of one or more select codes of the form <isc> × 100 + <addr>, also with or without subcommands.

Whenever any of the above items is found in a program, it is translated according to the rules described in the alphabetical keyword listing. The replacements in individual statements are indicated by <new isc>, <new sc,adr>, <new dev> and <new scs>.

If all of the expressions comprising a select code or address of any of the above forms are constants, then translation is performed according to the keyword listing rules.

Graphics Differences

Your new system's BASIC Graphics Option offers significant new graphics capabilities while providing a high degree of compatibility with HP 9845 BASIC Graphics. Most HP 9845 graphics programs only require minor syntax modifications prior to being executed on your new system. Some HP 9845 graphics programs may require reprogramming to give acceptable results.

Actions necessary to move HP 9845 graphics programs to your new system are divided into two categories: "Graphics Programming Concept Changes" and "Minor Program Changes". The first category includes several graphics programming concept changes which may force substantial revision to some HP 9845 graphics programs. These changes are discussed in the following section. The second category includes minor program changes which may be necessary to achieve HP 9845-like behavior on your new system. These changes are discussed on a keyword by keyword basis under "Keyword Listing".

Graphics Programming Concept Differences

Logical View Surface

Your new system supports multiple display devices in a device-independent manner. User coordinate data is transformed into images represented in logical device coordinates (also known as Graphics Display Units or GDUs). GDUs can be thought of as the dimensions of an internal, "logical view surface". The images created on the logical view surface are mapped onto each enabled physical device. Thus output statements are transformed to a single internal representation and then scaled to fit onto various real output devices.

There is a single "viewing operation" to define how UDU is to be transformed to GDU. It is not possible to establish separate viewing operations for separate devices. Moreover, the viewing operation is unaffected by PLOTTER IS and LIMIT statements.

Thus there is exactly one image and exactly one GDU space. On the HP 9845, there could be multiple images AND multiple GDU spaces.

The mapping of the logical view surface to devices is normally a best-fit, non-distorting mapping. That is, the logical view surface is mapped to the largest possible rectangular area of the physical device with the same aspect ratio as the logical view surface. It is possible, however, to specify that the full limit area of the physical device be used (using PLOTTER IS or LIMIT). In this case, the mapping would scale the logical view surface to fit exactly on the device limit area. This full-fit mapping does not, however, alter the range of GDUs for that device. It merely "streches" the logical view surface to fit the entire limit area.

Note

Programs that rely on the 9845 capability to produce multiple, simultaneous, independent views cannot be transported to your new system. Such programs must be rewritten to output plots sequentially to each plotter.

The following changes are recommended to minimize differences in program behavior. First, begin each program with GINIT. This terminates all graphics peripherals and resets the viewing specification and all attributes to the default state. Second, add DISTORT to all PLOTTER IS, LIMIT and GRAPHICS INPUT IS statements. The anisotropic GDU mapping resulting from DISTORT may be undesireable in some applications. Finally, reset attributes and viewing specification, as necessary, following PLOTTER IS statements. It may be possible to precede each PLOTTER IS by a GINIT statement to reset attributes and viewing specifications to the default state. GINIT also terminates all graphics devices, thus this technique cannot be used if multiple graphics devices are being used.

Modal Attributes
Primitive attributes (PEN, LINE TYPE, FILL color, CSIZE, and so on) are globally maintained and device independent. It is not possible to set different attributes on different devices. For example, the following program sequence results in a line drawn using PEN 7 on both plotters.

```
PLOTTER 703 IS OFF
PLOTTER 705 IS ON
PEN 3
PLOTTER 705 IS OFF
PLOTTER 703 IS ON
PEN 7
PLOTTER 705 IS ON
DRAW 100,150
```

The only way to achieve the effect of device-dependent attributes is to repeat graphics output to each device after setting attributes desired for that device.

Modal attributes are also unaffected by PLOTTER IS and LIMIT statements. The only way to reset attributes to their default state is using the GINIT statement.

Current Position Updates
The current position is maintained in UDU and is not altered when the viewing operation changes. On the HP 9845 the (device-dependent) current position is maintained in device coordinates. Thus the current position may be different after a change to the viewing specification. For example, the value of X,Y after the sequence below is 1000,2000 NOT the GDU coordinates of the pen position on the plotter.

```
WINDOW -1000,1500,-1000,3750
DRAW 1000,2000
SETGU
WHERE X,Y
```

Plotter Array Contents
Output to a plotter array differs in three major ways.

First, the number of array entries and sequence of these entries may not be identical to the HP 9845. For example, PLOTTER A(*) IS ON outputs all current attributes, including area fill color. Programs which access specific rows in these arrays must be changed to use the correct index value.

Second, the coordinates are placed in the array in UDU (after application of the modelling transformation), rather than GDU as in the HP 9845. Programs which use MAT PLOT to display these arrays must be changed to establish the correct viewing specification.

Finally, the array must be in COMMON. It may be necessary to initialize arrays since COM arrays are not automatically initialized.

Keyword Listing

Actions taken by the TRANSLATOR utility program are boxed and use the symbol "⟶" to indicate how the TRANSLATOR modifies a statement. An HP 9835/45 statement or partial statement precedes the "⟶"; the result of the modification made by the TRANSLATOR follows.

In TRANSLATOR action boxes, two comments, "Diagnostic given." and "One time diagnostic.", appear numerous times with no further information on what the diagnostic means. In both cases, the diagnostic issued by the TRANSLATOR contains the essence of the semantic differences as explained in the unboxed discussion of the keyword. Some diagnostics also refer to this appendix or to the BASIC Language Reference for further details. "Diagnostic given." also means that the diagnostic is issued with **each** statement where the condition occurs. "One time diagnostic." means that individual occurrences are not noted, but that the diagnostic is issued with the other summary diagnostics at the end of the translation.

+, −,NOT:

The precedence of unary +, unary -, and NOT have changed from the 9835/45. Unary plus and minus now have the same priority as binary plus and minus. NOT has priority between relational operators and AND. Some expressions require parenthesization to obtain equivalent results.

The TRANSLATOR parenthesizes where necessary to account for lowered precedence of unary forms. Due to simplified parsing assumptions, some occurrences of unary minus and NOT are parenthesized unnecessarily, but no necessary parenthesizations are missed.

ABORTIO:

ABORTIO becomes ABORT.

```
     ABORTIO <isc>
 ⟶  ABORT <new isc>
```

ALPHA:

```
   ALPHA
──► ALPHA ON
```

ASSIGN:

Only one form of ASSIGN # is available on your new system:

```
ASSIGN #<file num> TO <file spec> [;RETURN <rtn>]
```

On the HP 9835/45, the return variable was a code. On your new system, it is the number of the error that would have been reported. Also remember to look at protect codes, file names, and msus' for changes.

```
     ASSIGN #<file num> TO <file spec>[,<rtn>[,<password>]]
or   ASSIGN <file spec> TO #<file num>[,<rtn>[,<password>]]
──►  ASSIGN #<file num> TO <new file spec>[;RETURN <rtn>]
```

Diagnostic - the RETURN variable is now the error number that would have been reported rather than the the old error code.

```
     ASSIGN #<file num> TO *
or   ASSIGN * TO #<file num>
──►  ASSIGN #<file num> TO *
```

AUTO:

There is no equivalent for AUTO on your new system.

AXES:

Current position is updated to intersection.

One time diagnostic.

BUFFER #:

There is no equivalent for BUFFER # on your new system. All files are buffered.

Line commented out.

CARD ENABLE:

See CONTROL MASK.

```
    CARD ENABLE <isc>
──► ENABLE INTR <new isc>
```

CAT,
CAT #,
CAT TO:

There is a single uniform syntax for all CAT statements on your new system:

```
CAT [<source>] [TO <dest>] [;<options>]
```

A CAT statement on the HP 9835/45 which looks like this:

```
CAT"X:F8"
```

looks like this on your new system:

```
CAT ":HP9885,8";SELECT "X".
```

A statement which looks like this on the HP 9835/45:

```
CAT ":H8,1"
```

looks like this on your new system:

```
CAT ":HP9895,801"
```

An example of a CAT TO statement and its translation is:

```
CAT TO A$(*),I,J;Dev$
```

becomes

```
CAT Dev$ TO A$(*);SKIP I, COUNT J, NO HEADER
```

NOTE: CAT format has changes. In CAT...TO skip count may be affected and return count is also affected. Also don't forget to check the msus' and the file names for additional changes.

```
   CAT
—► CAT

   CAT "<selector><msus>"
—► CAT "<new msus>";SELECT "<selector>"

   CAT #<sc,ad>[;"<selector><msus>"[,<hdr sup>]]
—► CAT["<new msus>"] TO <new sc,ad>[;SELECT"<selector> "[,NO HEADER]]

   CAT TO <id>$(*)[,<skip>[,<var>]][;"<selector><msus> "[,<hdr sup>]]
—► CAT ["<new msus>"] TO <id>$(*)[; [SELECT"<selector>"]
       [,SKIP<skip>[,COUNT<var>] [,NO HEADER]]
```

Diagnostic - skip count and return count affected.

If <hdr sup> is not given, it defaults to NO HEADER for CAT TO... For CAT #..., NO HEADER is not added.

If <hdr sup> is given:

<hdr sup>	**translated to**
0	ignored
1	NO HEADER
else	diagnostic given using default header option for statement.

CHECK READ:

```
   CHECK READ
—► CHECKREAD ON
```

CHECK READ OFF:

```
   CHECK READ OFF
—► CHECKREAD OFF
```

CLEAR:
Check select code sequence.

```
   CLEAR <scs>
—► CLEAR <new scs>
```

CLIP:

Does not allow MIN parameters to be greater than MAX parameters.

> One time diagnostic.

COL(A):

Use SIZE(A,RANK(A)) unless RANK(A) is 1, then use 1.

```
    COL(A)
──► SIZE(A,RANK(A))
```

> Diagnostic - use 1 if RANK(A) is 1.

COLLECT:

RNEQ

> Line commented out. One time diagnostic.

CONFIGURE:

Use SEND. If using TALK or LISTEN options behavior may differ. Refer to the BASIC Language Reference for details.

```
    CONFIGURE <isc>[TALK=<exp>][LISTEN =<exp list>]
──► SEND <new isc>;MTA;UNL[;TALK<exp>][;LISTEN <exp list>]
```

> One time diagnostic.

CONTROL MASK:

The only method of specifying an interrupt mask on your new system is by the use of ENABLE INTR. Since this may enable interrupts at an undesirable time, CONTROL MASK should probably be dropped and the corresponding CARD ENABLE replaced with ENABLE INTR and specifying the the mask at that place. Also, check the device register for the proper mask.

```
    CONTROL MASK <isc>;<num exp>
──► ENABLE INTR <new isc>;<num exp>

    CONTROL MASK <isc>;<str exp>
──► ENABLE INTR <new isc>;DVAL(<str exp>,2)
```

> Diagnostic - check the new device register specification for proper mask. Translation may enable interrupts for this device earlier than on the HP 9835/45. Check to see if this enabling statement needs to be moved.

CONVERGE:

> Line commented out.
> Diagnostic - use provided BASIC utility.

CONVERT:

RNEQ (See CONVERT attribute in the ASSIGN @... statement in the BASIC Language Reference and the note under ENTER in this keyword list.)

```
Line commented out.
Diagnostic - try using CONVERT attribute in the ASSIGN @... statement.
```

CREATE:

CREATE becomes CREATE BCD.

Another file type may be equally or more appropriate; your new system also allows CREATE for DATA, ASCII, BDAT and PROG type files.

```
   CREATE <file spec>,<nrec>[,<reclen>]
→  CREATE BCD <new file spec>,<nrec>[,<reclen>]

   One time diagnostic - some other file type may be more appropriate.
```

CRT, Writing To:

The CRT escape sequences on the HP 9835/45 are area dependent. The CRT escape sequences on your new system are time dependent.

Example:

Start a new line. Write 5 characters to CRT. Turn inverse video on. Write 10 more characters. Turn inverse video off. Write 5 more characters. So far, no difference.

Difference 1: Now, return to the beginning of that line and turn inverse video on. On the HP 9835/45, the first 15 characters are now in inverse video, because inverse video on is an attribute of character position 1 and inverse video off is an attribute of character position 16. On your new system, however, the first 5 characters are still in normal video, since you have only enabled the (time dependent) mode of inverse video but written no characters in that mode.

Difference 2: Move to position ten of the line and write 1 character (ignoring any actions in Difference 1 above). On the HP 9835/45 the newly written character is in inverse video because it was written into an inverse video field. On your new system, that single character in the middle of the inverse field is in normal since it was written in the normal mode and it has no effect on the other nine characters in the field.

PRINT PAGE:

PRINT PAGE to the CRT on the HP 9835/45 clears the screen and the scroll buffer. On your new system it clears the screen by scrolling, but does not explicitly destroy information in the scroll buffer.

CRT ON/OFF:

> Line commented out.
> Diagnostic - CRT on your new system has its own memory and processor. Limiting access is unnecessary.

CRT SFK's:

The SFK's which can be labeled on the HP 9835/45, are keys 24 thru 31. They are keys 0-7 on your new system.

> See ON KEY in this appendix for TRANSLATOR action information.

CSIZE:

Character cell height equal to zero is not allowed. Slant angle is independent of the character cell aspect ratio.

> One time diagnostic.

CURSOR:

CURSOR becomes READ LOCATOR.

Status string is now device independent. No special data is needed for LIGHT PEN, HP 9111 and so on.

CURSOR cannot be used to read point saved by ON GKEY event. Replace CURSOR in ON GKEY with GKEY statement. (See DIGITIZE in this keyword list for more information.)

```
   CURSOR <x>,<y>[,<stat$>]
→  READ LOCATOR <x>,<y>[,<stat$>]
```

> One time diagnostic.

DBBACKUP,
DBRECOVER:

Removed, use the BACKUP and RECOVER options of the Utilities subsystem of QUERY.

> Line commented out with diagnostic.

DBCREATE,
DBERASE,
DBPURGE:

```
DBxxx base$[;maint$][,sets$] becomes
DBxxx base$[;[MAINT maint$][[,]SETS sets$]
```

File specifier references need to change to conform with the new MSUS syntax. The ROOT file is not purged in a DBPURGE statement without a ROOT parameter.

```
    DBxxx <file spec>[;<maint>][,<sets>]
──► DBxxx <file spec>[;MAINT<maint>][[,]SETS<sets>]
```

Diagnostic given with DBPURGE.

Note: The TRANSLATOR cannot change file names since they are always variables.

DBCLOSE,
DBDELETE,
DBFIND,
DBGET,
DBINFO,
DBOPEN,
DBPUT,
DBUPDATE:

The syntax of file and device specifiers have changed and the status array must be a ten element array of type DOUBLE.

One time diagnostic.

Note: The TRANSLATOR cannot change file names since they are always variables.

DECIMAL:

```
    DECIMAL(<exp>)
──► DVAL(DVAL$(<exp>,10),8)
```

DEFAULT ON/OFF:

On your new system, DEFAULT ON has no effect on INTEGER or DOUBLE operations; errors still occur for any operations performed in these types.

One time diagnostic.

DEGAUSS:

RNEQ

Line commented out.

DEL FN:

```
    DEL FN<function name> [TO END]
──► DEL SUB FN<function name> [TO END]
```

DEV$:

Equivalent on your new system is SYSTEM$("MSI").

```
    DEV$
──► SYSTEM$("MSI")
```

DIGITIZE:

Status string is now device independent. No special data is needed for LIGHT PEN, HP 9111 and so on.

The DIGITIZE statement is now completely independent of ON KEY. It is not possible to use DIGITIZE (or CURSOR) to return the locator position at the time of the GKEY event. When the event occurs, an event report is saved on an internal event queue. The GKEY statement must be used to read the information saved in the event queue. Thus, the first use of DIGITIZE (or CURSOR) in ON GKEY service routines should be replaced by the GKEY statment.

DIGITIZE from ARROW KEYS does not automatically place an echo (tracking cross) on the internal CRT. The echo is only enabled if TRACK is on for the ARROW KEYS and CRT.

One time diagnostic.

DIM, REAL, SHORT, INTEGER:

These statements on the HP 9835/45 allow expressions for bounds. Your new system only allows integer constants. If variable bounds are necessary, put those arrays and/or strings in ALLOCATE statements, and move them to the appropriate location in the context.

If any non-constant expressions in bounds declarations are found, the line is commented out with a diagnostic.

DROUND, PROUND:

The form of these two statements is the same on your new system as on the HP 9835/45. However, the change from BCD to binary math may cause results to differ from those results obtained on the HP 9835/45.

One time diagnostic.

DUMP GRAPHICS:

If specifying a device, use the new "device selector" format.

The upper and lower limits of the area to dump are interpreted in GDU. Programs which specify these limits in UDU requires modification to calculate the appropriate GDU value.

Change:

```
DUMP GRAPHICS Lower, Upper
```

To:

```
Udu_x=0               !any x in the 2D window works
ASKGU Udu_x,Lower;Gdu_x,Gdu_lower
ASKGU Udu_x,Upper;Gdu_x,Gdu_upper
DUMP GRAPHICS; Gdu_lower,Gdu_upper
```

```
    DUMP GRAPHICS [<ymin>[,<ymax>]]
--> DUMP GRAPHICS [;<ymin>[,<ymax>]

    DUMP GRAPHICS #<sc,ad>[;<ymin>[,<ymax>]]
--> DUMP GRAPHICS #<new sc,ad>[;<ymin>[,<ymax>]]
```

One time diagnostic.

ENTER:

If using option WHS, BINT, or WINT, use a BYTE or WORD attribute in the ASSIGN @... statement.

There is no equivalent for options BFHS, WFHS, DBMA, WDMA, or BYTE. Numeric array as source is not supported.

If TRL is used, try using the DELIM attribute in the ASSIGN statement, although there will be some differences. The TRL character was used **in addition to** line feed; DELIM replaces line feed. The TRL character was not stripped from entered strings, but the DELIM character is.

Note

Here and in many other situations, the use of some attribute in an ASSIGN @... statement is suggested. If I/O is being done to or from a device, behavior should be acceptable after the appropriate qualifications have been accounted for. An additional qualification is necessary, however, when ENTERing from or OUTPUTing to a string. In order to execute:

```
ASSIGN @<name> TO <string>
```

the string must be a BUFFER. BUFFER variables are treated as circular buffers whose pointers to the next location to ENTER from or OUTPUT to is maintained between successive invocations of:

`ENTER @<name>` and `OUTPUT @<name>`

This differs from ENTER <string> and OUTPUT <string> where each invocation always starts at the beginning of a string.

```
    ENTER <scs>[<type1> <type2> <count>]
        [USING<image> NO FORMAT];<list>
──► ENTER <new scs> [USING<new image>];<list>
```

<type1> ::= WHS | BINT | WINT
<type2> ::= BFHS | WFHS | DBMA | WDMA

Any occurrence of <type1> or <type2> is removed.
If <type1> is found, the TRANSLATOR suggests using the BYTE (for BINT) or WORD (for WHS or WINT) attribute in the ASSIGN @... statement. If <type2> is found, the line is commented out.

```
    ENTER <source> [BYTE][USING<image>];<list>
──► ENTER <source> [USING <new image>];<list>
```

If <source> is a numeric array element, RNEQ. If <source> is a string variable, leave as is. If the BYTE attribute is included, RNEQ.

For ENTER...USING, if the image given is a single string literal, it is always translated.

In either case, if <list> begins with TRL(<exp>), it is removed and a diagnostic is given.

EOI:

EOI is not supported directly. You may add the EOI expression to the OUTPUT statement followed by ";END".

```
    EOI <isc>,<exp>
──► OUTPUT <new isc> USING "#,B";<exp>;END
```

EOL:

RNEQ Consider using EOL attribute in the ASSIGN statement. You need to be aware that your new system's EOL sequence is output everywhere the HP 9845 would have output a CR-LF. See note under ENTER in this keyword list.

Line commented out with a diagnostic.

ERRL:

The parameterless function ERRL on the HP 9835/45 which returns the line number on which an error occurred has been changed to a Boolean function with one parameter. The parameter is a line number or line label; the function returns 1 if the error occurred in the referenced line, 0 if not.

Simple uses of ERRL are translated easily. For example, ERRL=100 becomes ERRL(100) and ERRL<>100 becomes NOT ERRL(100). More complex uses require more care.

```
      ERRL=<integer constant>
or    <integer constant>=ERRL
──►   ERRL(<integer constant>)

      ERRL<><integer constant>
or    <integer constant><>ERRL
──►   (NOT ERRL(<integer constant>))

      More complex expressions are commented out with a diagnostic.
```

EXIT ALPHA:

```
      EXIT ALPHA
──►   ALPHA OFF
```

EXIT GRAPHICS:

```
      EXIT GRAPHICS
──►   GRAPHICS OFF
```

FCREATE,
FPRINT,
FREAD:

There is no direct equivalent for these statements on your new system. One solution, which requires the I/O option is:

FCREATE use CREATE BDAT <file>,<size>

```
FPRINT use ASSIGN @Fprint TO <file>
           OUTPUT @Fprint;<array ident>
           ASSIGN @Fprint to *

FREAD use ASSIGN @Fread to <file>
          ENTER @Fread;<array ident>
          ASSIGN @Fread TO *
```

They could also be replaced with READ # and PRINT # statements. In either case, however, these substitutions will not achieve DMA speeds, nor will they automatically write and read array bounds as would the original statements.

```
      Line commented out with a diagnostic.
```

Filenames:

The following characters are not allowed in your new system's filenames:

<
:
/

See the opening discussion in this appendix and in the TRANSLATOR Utility in the "Utilities" appendix for more information.

FOR...NEXT:

Use of non-integer valued bounds and/or step sizes in a FOR...NEXT statement may cause different results. Some FOR loops may execute fewer or more passes due to binary arithmetic. Refer to the "Structured Programming" chapter in this manual.

The HP 9835/45 allows multiple NEXT statements and does dynamic matching of FOR/NEXT pairs. On your new system, however, the matching is done statically and multiple NEXT statements are not allowed nor are improperly constructed FOR...NEXT pairs.

If any non-integer valued bound or step size is detected, a diagnostic is given.

GLOAD:

The GRAPHICS plotter must be enabled. GLOAD loads only the visible portion of the frame buffer.

One time diagnostic.

GRAPHICS:

```
    GRAPHICS
──► GRAPHICS ON
```

GRAPHICS INPUT IS:

Use the new device selector format; change the device identifier of "DIGITIZER", "TABLET", or "9872A" to "HPGL".

```
    GRAPHICS INPUT IS [<sc,ad>]<input$>[,<parm>]
──► GRAPHICS INPUT IS [<new sc,ad>]<new input$>[,<parm>]
```

If <input$> is "DIGITIZER", "9872A" or "TABLET", it is changed to "HPGL".

```
GRAPHICS INPUT IS <id>(*)
```

RNEQ - A diagnostic is given.

GRAPHICS INPUT IS ON/OFF

Use the new device selector format.

```
    GRAPHICS INPUT <sc,ad> IS ON/OFF
──► GRAPHICS INPUT <new sc,ad> IS ON/OFF

    GRAPHICS INPUT <id>(*) IS ON/OFF
```
RNEQ - A diagnostic is given.

GRAPHICS INPUT IS A(*):

RNEQ

Line commented out with diagnostic.

GRAPHICS INPUT A(*) IS ON/OFF:

RNEQ

Line commented out with diagnostic.

GRID:

The current position is updated to intersection.

One time diagnostic.

GSTAT:

RNEQ; may use SYSTEM$.

Line commented out with diagnostic.

GSTORE:

GRAPHICS plotter must be enabled.

GSTORE stores only the visible portion of the frame buffer.

One time diagnostic.

HOLE:

On SDF discs, files need not be contiguous; available space can be found using:

```
CAT TO A$(*)
```

On LIF discs there is no equivalent.

Line commented out with diagnostic.

IMAGE:

See guidelines at beginning of keyword list.

```
    IMAGE <spec list>
──► IMAGE <new spec list>
```

The IMAGE statement is scanned for possible translations. This scan considers three kinds of image specifiers:

1. outbound only - can never be used for input;
2. inbound only - can never be used for output;
3. bi-directional- can be used for both input and output.

If no outbound only specifiers are found, and at least one inbound only specifier is found, any occurrence of a changed specifier (F, N, T, + and %) is translated.

If no inbound specifier is found, and at least one outbound only specifier is found, the statement is assumed to be for output and is ignored.

If both inbound only and outbound only specifiers are found, the statement is not modified, but a diagnostic about conflicting specifiers is given.

If only bi-directional specifiers and a + , which needs translation if used for input, are found, the statement is not modified and a diagnostic about ambiguity is given.

INTEGER:

See DIM.

Integer Math:

Some INTEGER (and DOUBLE) operations may overflow on your new computer that would not have on the HP 9845. Since your new computer performs INTEGER (and DOUBLE) arithmetic on 32 bit integer intermediate values wherever reasonable, some intermediate results may overflow where they would not on the HP 9845, where all arithmetic operations were performed in full REAL precision.

IOFLAG,
IOSTATUS:

RNEQ

Line commented out.

IOR:

Change to BINIOR. This requires the IO option.

```
    IOR(<num exp>,<num exp>)
──► BINIOR(<num exp>,<num exp>)
```

Some INTEGER (and DOUBLE) operations may overflow on your new computer that would not have on the HP 9845. Since your new computer performs INTEGER (and DOUBLE) arithmetic on 32 bit integer intermediate values wherever reasonable, some intermediate results may overflow where they would not on the HP 9845, where all arithmetic operations were performed in full REAL precision.

KBD$:

Diagnostic-key codes have changed.

KEY LABELS:

No direct equivalent. Try using ENTER SCREEN (6)...

Line commented out with diagnostic.

LABEL KEY #:

```
   LABEL KEY #<exp> <str exp>
──► ON KEY (<exp>)-24 LABEL <str exp>
```

In most cases LABEL KEY # is used in conjunction with an ON KEY # statement. With the ON KEY... LABEL... statement on your new computer these two actions can be merged. This is the assumption made by the TRANSLATOR in making this partial translation. Since the TRANSLATOR cannot guess the CALL/GOSUB/ GOTO action desired, the line is then commented out, assuming you will complete the statement appropriately.

LABEL KEYS:

```
   LABEL KEYS <str exp>
──► OUTPUT SCREEN(6);CHR$(12);<str exp>
```

Note that this translation requires the IO option. An alternative, which does not require the IO option is:

```
Save_printer=VAL(SYSTEM$("PRINTER IS"))
PRINTER IS SCREEN(6)
PRINT PAGE,<str exp>
PRINTER IS Save_printer
```

LASTBIT:

RNEQ

Line commented out.

LAXES:

The current position is updated to the intersection.

One time diagnostic.

LETTER:

Only one string may be entered. That string is output to all enabled plotters.
LETTER uses the current echo type.
LETTER does not do implicit GRAPHICS ON.

One time diagnostic.

LEX:

See LEXICAL ORDER operations.

Line commented out.

LEXICAL ORDER operations:

LEXICAL ORDER IS now affects all string compares, thus the LEX function is no longer needed and no longer supported. This does mean, however, that great care must be taken in running any program which uses LEXICAL ORDER, since explicit string compares now are performed in the current lexical order, rather than in ASCII as on the HP 9845.

LEXICAL ORDER IS STANDARD becomes LEXICAL ORDER IS ASCII. On your new system LEXICAL ORDER IS STANDARD sets the lexical order according to the keyboard language.

At power-up and SCRATCH A, LEXICAL ORDER is reset to that corresponding to the keyboard language. SCRATCH P effects the same reset for that partition. Thus, for equivalent operation, each program which runs on your new system with other than an ASCII or KATAKANA keyboard, a LEXICAL ORDER IS ASCII statement should be executed at the beginning of **every** program.

All of the lexical orders provided in table form with the HP 9845 Advanced Programming ROM are now available by keyword on your new system. They are equivalent on your new system, with two exceptions:

1. Character codes 127-160 and 223-255 formerly collated with sequence number 0. They now collate sequentially above what would formerly have been the highest sequence number in each lexical order. See the "Useful Tables" section of the BASIC Language Reference for more detailed information.
2. Character code 255, which represented a "don't care" character in the ASCII table now collates as a legitimate character in its normal ASCII sequence.

Any lexical table other than the standard ones will have to be translated, since the table format is now different. See the BASIC Language Reference manual under LEXICAL ORDER IS for details.

`LEXICAL ORDER IS ASCII` is inserted at the beginning of the MAIN program.

```
  LEXICAL ORDER IS STANDARD
→ LEXICAL ORDER IS ASCII
```

`LEXICAL ORDER IS <name>(*)` is commented out with a diagnostic.

LGRID

The current position is updated to the intersection.

One time diagnostic.

LIMIT:

Maps the Logical View Surface to P1-P2 area in non-distorted, best-fit manner. If full use of P1-P2 area is desired, add ";DISTORT"

DISTORT is added.

LIN:

On your new system, the argument to LIN is the number of EOL sequences to output; arguments less than or equal to zero do nothing.

```
  LIN(<exp>)
→ LIN(<new exp>)
```

If <exp> is a positive integer constant, it is left alone.
If <exp> is a negative integer constant, the minus sign is removed and a diagnostic is given that behavior may differ.
If <exp> is the constant 0, it is replaced with 1.
If <exp> is anything alse, a diagnostic is given stating that non-positive arguments do nothing.

LINE TYPE:

The length of tick marks on line types 9 and 10 is different. HPGL repeat length is implemented.

One time diagnostic.

LINK:

RNEQ Try using LOADSUB.

Line commented out with diagnostic.

LOAD ALL:

RNEQ

Line commented out.

LOAD BIN

The LOADBIN statement is supported on your new system, but its behavior is significantly different; each LOADBIN performs an implicit SCRATCH A.

> Line commented out.

LOAD SUB:

The closest equivalent to LOAD SUB will be LOADSUB ALL. Starting line number and increment cannot be given on your new system. Starting and ending SUB numbers cannot be translated; selective loading can only be done by giving the SUB or FN name.

```
    LOAD SUB<file spec>[,<line>[,<inc>]][;<start sub> [,<end sub>]]
──► LOADSUB ALL FROM <new file spec>
```

If <file spec> only is given, it is translated as above.
If <line> number is included, it is translated as above with a note that your new system renumbers automatically.
If <start sub> number is included, the line is commented out with a diagnostic.

LOCAL:

```
    LOCAL <scs>
──► LOCAL <new scs>
```

LOCAL LOCKOUT:

Translate select code sequence.

```
    LOCAL LOCKOUT <isc>
──► LOCAL LOCKOUT <new isc>
```

LOCATE:

Use VIEWPORT instead.

At least one point must lie within the limit area. VIEWPORT is truncated to Logical View Surface. The VIEWPORT effect on viewing operation is immediate.

```
    LOCATE [<list>]
──► VIEWPORT [<list>]
```

LORG:

The LORG statement now applies justification to the entire character cell. On the HP 9845 justification was based on the character within the cell. The following sequence may be used to achieve justification which is identical to the HP 9845. These translations assume that Org already has an integer value and is not relying on the rounding which the system performs. Otherwise, precede the translation with:

```
Org=INT(Org+.5)
```

to simulate rounding.

Change:

```
    LORG Org
```

To:

```
    XJust=(((Org-1) DIV 3)/2)
    If Org<4 THEN XJust=XJust+(1/9)/St_length
    If Org>6 THEN XJust=XJust-(2/9)/St_length
    YJust=((Org-1) MOD 3)/2)*(8/15)+4/15
    LORG XJust,YJust
```

where:

`St_length` is the length of the string.

To achieve justification that is within a pixel of the HP 9845 x justification, but independent of string length:

Change:

```
    LORG Org
```

To:

```
    XJust=(((Org-1) DIV 3)/2)
    YJust=(((Org-1) MOD 3)/2)*(8/15)+4/15
    LORG XJust,YJust
```

```
    LORG <exp>
--> LORG(INT((<exp>)-.5)DIV 3)/2,(INT((<exp>)-.5)MODULO 3+1)*4/15
```

MAT A = CON[(...)],
MAT A = ZER[(...)],
MAT A = IDN[(...)]:

All statements of the form:

`MAT <id>=<mat exp>`

are the same on your new system with the following exceptions:

```
    MAT A=CON(<new dim>)
--> REDIM A(<newdim>)
    MAT A=(1)

    MAT A=ZER(<new dim>)
--> REDIM A(<new dim>)
    MAT A=(0)

    MAT A=IDN(<new dim>)
--> REDIM A(<new dim>)
    MAT A=IDN
```

MAT AIPLOT,
MAT APLOT,
MAT ARPLOT:

MAT must be removed. An array specifier, (*) is required if Array(start:end) is not present. Does not update the Current Position.

```
    MAT AIPLOT<aref>
→   AIPLOT<new aref>
    where <aref> means add "(*)" to an array name by itself, but leave as is if array
    name with bounds.

    MAT APLOT<aref>
→   APLOT<new aref>
    where <aref> translates as above

    MAT ARPLOT<aref>
→   ARPLOT<new aref>
    where <aref> translates as above

    A one time diagnostic about the current position is given in all cases.
```

MAT IPLOT,
MAT PLOT,
MAT RPLOT,
MAT SYMBOL:

MAT must be removed. An array specifier, (*), is required if Array(start:end) is not present.

```
    MAT<key><aref list>[,FILL]
→   <key><new aref list>[,FILL]
    where <key>::=IPLOT|PLOT|RPLOT|SYMBOL and <aref list> is one to three
    occurrences of <aref> translated as above
```

MAT INPUT,
MAT READ,
MAT READ #:

If any array is redimensioned in the MAT INPUT/READ statement, this redimensioning must be accomplished in a REDIM statement prior to the new INPUT statement. Check to be sure that the new REDIM behaves the same as the original. If the INPUT list is exhausted prior to inputting the first element of any array, other than the first, on the HP 9845, the HP 9845 would not have redimensioned it.

```
    MAT<keyword> A,B$,C
→   <keyword> A(*),B$(*),C(*)

    MAT<keyword> A(<new dim>),B$,C(<new dim>),D$(<new dim>)
→   REDIM A(<new dim>),C(<new dim>),D$(<new dim>)
        <keyword>A(*),B$(*),C(*)D$(*)
    where <keyword> ::= INPUT|READ[#<file num>[,<rec num>];]
```

MAT PRINT,
MAT PRINT#:

These statements may be translated by dropping the MAT keyword and adding "(*)" to each array name.

```
    MAT PRINT[#<file num>[,<rec num>];]A,B$,C[,END]
--> PRINT [#<file num>[,<rec num>];]A(*),B$(*),C(*) [,END]
```

MAT SEARCH:

In general there is no equivalent for this statement on your new system.

If you are using MAT SEARCH just to determine the maximum or minimum element of an entire array, you can now use the MAX or MIN function.

If you are searching an entire array with the condition #LOC then a possible translation (which requires a temporary array) is:

```
MAT SEARCH A(*),#LOC(<relational op><expression>),<variable>
```

becomes:

```
MAT Temp=A<relational op>(<expression>)
<variable> =SUM(Temp)
```

Line commented out.

MAT SORT:

The MAT SORT statement now requires a key specifier-if sorting a one-dimensional array the form MAT SORT A must be MAT SORT A(*).

```
    MAT SORT<id> [TO <id>]
--> MAT SORT<id>(*) [TO <id>]
```

All other syntaxes remain unchanged.

MEMORY:

The internal CRT or one of its memory planes must be enabled as a PLOTTER.

One time diagnostic.

MOD:

The equivalent of MOD on the HP 9845 is MODULO on your new system. MOD on your new system has different semantics. Refer to the BASIC Language Reference or the "Numeric Computations" chapter for details.

```
    MOD
--> MODULO
```

MSCALE:

The metric scaling is established on the most recently initialized PLOTTER that is still enabled. Other PLOTTERS and GRAPHICS INPUT devices have metric scaling only if the hard clip areas on these devices is identical to the device for which metric scaling is established.

```
    One time diagnostic.
```

MSUS:

The MSUS codes have all changed. Affected statements include:

ASSIGN #
ASSIGN @...
COPY
DBOPEN and all other DBxxx statements
GET
LOAD
PROTECT
PURGE
RENAME
[RE-]SAVE
[RE-]STORE
LOAD KEY
STORE KEY
INITIALIZE
MASS STORAGE IS
VOLUME DEVICES ARE

NORMAL:

NORMAL becomes TRACE OFF.

```
    NORMAL
──► TRACE OFF
```

NOT:

See +, -, NOT.

OCTAL:

OCTAL(<exp>) is equivalent to DVAL(DVAL$(<exp>,8),10).

```
    OCTAL(<exp>)
──► DVAL(DVAL$(<exp>,8),10)
```

ON ERROR:

ON ERROR with CALL or GOSUB on your new system re-executes the statement causing the error on return from the error handling routine. The HP 9845 returns to the statement following the statement in error. Thus, if the error handling routine does not correct the source of the error or execute an OFF ERROR, your program could cycle in an infinite loop between an erroneous statement and its error handler.

Diagnostic given.

ON GKEY:

If priority is given, it must be preceded by a comma on your new system. Refer to CURSOR and DIGITIZE for comments about their use in an ON GKEY service routine.

```
   ON GKEY [<priority>] <action>
→  ON GKEY [,<priority>] <action>
```

ON/OFF INT #:

ON INT # becomes ON INTR. Update select code. Note: ENTER and OUTPUT no longer interrupt when they complete.

```
   ON INT #<isc>[,<prio>] <action>
→  ON INTR <new isc>[,<prio>] <action>
```

One time diagnostic is given about ENTER and OUTPUT.

OFF INT # becomes OFF INTR. Update select code.

```
   OFF INT #<isc>
→  OFF INTR <new isc>
```

ON KBD [ALL]:

For syntax changes see BASIC Language Reference.

The default set of untrapped keys on your new system is:

PAUSE
STOP
SHIFT RUN
SHIFT PAUSE
SHIFT CONT
SHIFT STOP
NEXT PARTITION
SPECIAL FUNCTION KEYS

The default set of untrapped keys on the HP 9835/45 is:

DOWN ARROW
ROLL UP
ROLL DOWN
TYPEWRITER
STOP
CONTROL-STOP (reset)
AUTO START
PRINT ALL

Specifying ALL on your new system traps all keys.

ALL on the HP 9835/45 traps all keys except:

CONTROL-STOP (reset)
AUTO START
PRINT ALL
UP ARROW

```
   ON KBD [<prio>] <action>[,ALL]
→  ON KBD [ALL][,<prio>] <action>
```

ON/OFF KEY #:

Change to ON/OFF KEY.

If ON KEY re-map has not been requested, these are translated:

```
   OFF KEY #<num>
→  OFF KEY <num>

   ON KEY #<num>[,<prio>]<action>
→  ON KEY <num>[,<prio>] <action>
```

If ON KEY remap has been requested, these are translated:

```
   OFF KEY #<num>
→  OFF KEY (<num>) MODULO 24

   ON KEY #<num>[,<prio>] <action>
→  ON KEY (<num>) MODULO 24 [,<prio>] <action>
```

OUTPUT:

Replace BINT, WINT, BFHS, WFHS, BDMA, WDMA, or WHS options with the BYTE or WORD attribute in the ASSIGN statement; handshake type is automatically selected by the driver.

There is no equivalent for the BYTE option when a string is the destination.

Your new system does not support a numeric array used as the destination.

Replace NOFORMAT with the FORMAT OFF attribute in the ASSIGN statement.

```
    OUTPUT <dest>[ <type>][USING <image> NOFORMAT];<list>
──► OUTPUT <new dest>[USING <new image>];<list>
```

where <type> ::= BINT | WINT | BFHS | WFHS | BDMA | WDMA | WHS | BYTE

If <dest> is an numeric array element, the line is commented out with a diagnostic.

If <dest> is <scs> then <new dest> is <new scs>.

If <dest> is a string variable, the the TRANSLATOR leaves it as is. If <type> is BYTE, the line is commented out with a diagnostic.

If a <type> other than BYTE is give, it is removed with a diagnostic.

If NOFORMAT is given, it is removed with a diagnostic.

OVERLAP:

OVERLAP has no effect on ENTER. All ENTERs are serial on your new system.

One time diagnostic.

PASS CONTROL:

Check the select code sequence.

```
    PASS CONTROL <scs>
──► PASS CONTROL <new scs>
```

PDIR:

On the HP 9845, the transformed image is rotated by PDIR. On your new system, the untransformed object is rotated. Thus, the rotation may introduce distortion not seen on the HP 9845.

The angle parameter is interpreted in current units on your new system, as opposed to always being GDUs on the HP 9845. The following sequence can be used to get the correct rotation angle if current units are UDU:

Change:

```
PDIR Angle
```

To:

```
ASKUU 0,0; Dx0,Dy0
ASKUU COS(Angle),SIN(Angle);Dxuu,Dyuu
PDIR Dxuu-Dx0,Dyuu-Dy0
```

One time diagnostic.

PEN:

PEN action is independent of background color and is globally maintained. See the GRAPHICS Modal Attributes section preceding this keyword list.

> One time diagnostic.

PLOTTER IS:

Use the new device selector format.

Device identifier string "INCREMENTAL" is not allowed. Device identifier string "9872A" becomes "HPGL".

This statement has no effect on values for primitive attributes (PEN, LINE TYPE, AREA COLOR, etc.) or viewing parameters (WINDOW, VIEWPORT, etc). The range of GDU coordinates in x and y directions is fixed by the aspect ratio specified in GINIT. Thus GDU space is totally device independent. The Logical View Surface is mapped to the P1-P2 area of the plotter in a non-distorting, best-fit manner. Add ";DISTORT" if full use of P1-P2 area is desired.

> `PLOTTER IS` <id>`(*)` translated as is.
>
> `PLOTTER IS` [<sc,ad>] <str exp> [<exp list>]
> ⟶ `PLOTTER IS` [<new sc,ad>] <new str exp> [<exp list>]`;DISTORT`
>
> One time diagnostic.
>
> If <str exp> is "9872A" it is changed to "HPGL". If <str exp> is "INCREMENTAL" the line is commented out.

PLOTTER...IS ON/OFF:

Use new device selector format.

> `PLOTTER` <id>`(*)` `IS ON/OFF` translated as is.
>
> `PLOTTER` <sc,ad> `IS ON/OFF`
> ⟶ `PLOTTER` <new sc,ad> `IS ON/OFF`
>
> One time diagnostic.

PLOTTER IS A(*):

Output to an array occurs in a different sequence, and additional output may occur. Programs which directly access these arrays by row index may require some reprogramming to establish new index values. The array must be in COM.

> One time diagnostic.

POINTER:

POINTER X,Y,Marker,Color should be replaced by SET LOCATOR X,Y followed by SET ECHO X,Y,Marker,Color.

Marker type 0 removes ECHO but does not disable movement. Marker types 4,5 and 6 do not result in rubberband lines.

```
   POINTER <x>,<y>[,<marker>[,<color>]]
──► SET LOCATOR <x>,<y>
   SET ECHO <x>,<y>[,<marker>[,<color>]]

   One time diagnostic.
```

POLYGON:

The POLYGON statement always results in a closed figure. On the HP 9845 the figure may not have been closed if not FILLed.

The HP 9845 POLYGON statment (without FILL) should be replaced by the POLYLINE statement. POLYGON with FILL requires no modification.

The HP 9845 POLYGON always outputs a regular polygon image, regardless of the UDU coordinate system. On your new system, the polygon is regular in UDU space, but may be transformed to an irregular image if the viewing operation is anisotropic.

```
   POLYGON <exp list>,FILL translated as is.

   POLYGON <exp list>
──► POLYLINE <exp list>

   One time diagnostic.
```

PPOLL,
PPOLL CONFIGURE,
PPOLL UNCONFIGURE:

Update select code sequence.

```
   PPOLL (<isc>)
──► PPOLL (<new isc>)

   PPOLL CONFIGURE <scs>;<exp>
──► PPOLL CONFIGURE <new scs>;<exp>

   PPOLL UNCONFIGURE <scs>
──► PPOLL UNCONFIGURE <new scs>
```

PRINT ALL IS:

The select codes have changed, as has the syntax for specifying an HP-IB address. For select code 0 use PRT, for 16 use CRT.

```
    PRINT ALL IS <sc,ad>
──► PRINTALL IS <new sc,ad>
```

PRINTER IS:

The select codes have changed, as has the syntax for specifying an HP-IB address. For select code 0 use PRT, for 16 use CRT. Also, the WIDTH parameter now requires the IO option and is separated from the device selector by a semicolon.

```
    PRINTER IS <sc,ad> [,WIDTH(<exp>)]
──► PRINTER IS <new sc,ad> [;WIDTH(<exp>)]
```

PRINT LABEL:

The PRINT LABEL <label>[ON <device spec>] on the HP 9835/45 has been changed to PRINT LABEL <label>[TO <device spec>] on your new system.

Don't forget msus changes.

```
    PRINT LABEL <label> [ON <msus>]
──► PRINT LABEL <label> [TO <new msus>]
```

PRINT PAGE:

See CRT PRINT PAGE.

Protect Codes:

Protect codes are now given with the file name in all statements that use them except PROTECT. These statements include ASSIGN, ASSIGN#, ASSIGN@..., COPY, GET, LOAD, MASS STORAGE IS, PURGE, RE-NAME, RE-SAVE and RE-STORE. Refer to "file specifier" in the glossary of the BASIC Language Reference.

PROUND:

See DROUND.

```
    One time diagnostic.
```

RANDOMIZE:

The HP 9835/45 randomizes on the fractional part of its argument. Your new system randomizes on the integer part. The seed may be in the positive range of DOUBLE on your new system.

```
    RANDOMIZE[(<expr>)]
──► RANDOMIZE[(FRACT(<expr>)*2.147E9)]
```

RATIO:

RATIO returns the aspect ratio of the Logical View surface and is device independent.

```
    One time diagnostic.
```

READ IO:

RNEQ

```
    Line commented out.
```

READ LABEL:

READ LABEL <label$>[ON <device spec>]
becomes READ LABEL<label$>[FROM <device spec>]

The <A$(*)> option has been deleted. Don't forget msus changes.

```
    READ LABEL <id>$[ON <msus>]
──► READ LABEL <id>$[FROM <new msus>]

    READ LABEL <id>$(*) commented out.
```

READBIN:

READBIN becomes ENTERBIN.

```
    READBIN(<dev>)
──► ENTERBIN(<new dev>)
```

REAL:

See DIM.

REMOTE:

Update device selector.

REMOTE does not drop ATN after raising REN. Refer to the BASIC Language Reference manual for further information.

```
    REMOTE <scs>
──► REMOTE <new scs>

    One time diagnostic.
```

REQUEST:

The new syntax does not allow string expressions. If a string expression is used, replace it with:

```
DVAL(<str exp>,2)
```

and update the device selector.

```
   REQUEST <isc>;<num exp>
→  REQUEST <new isc>;<num exp>

   REQUEST <isc>;<str exp>
→  REQUEST <new isc>;DVAL(<str exp>,2)
```

RESET:

Update the device selectors.

```
   RESET <scs>
→  RESET <new scs>
```

REWIND:

RNEQ

Line commented out.

RND:

One time diagnostic that the random number sequences are different.

ROW(A):

ROW(A) translates to SIZE(A,RANK(A) − 1) unless A is one-dimensional, in which case it is equivalent to SIZE(A,1).

```
   ROW(A)
→  SIZE(A,MAX(1,RANK(A)-1))
```

SCALE:

```
   SCALE <exp list>
→  WINDOW <exp list>
```

SECURE:

There is no equivalent for the SECURE statement on your new system; the COMPILE statement may be used for program security.

Line commented out with diagnostic.

SELECT CODE...ACTIVE/INACTIVE:
RNEQ

SENDBUS:
If SENDBUS address is the **interface select code**, SEND works with the syntax changes found in the BASIC Language Reference.

If addressing a device on the bus, with or without subcommands, use OUTPUT first to address the device followed by SEND.

```
    SENDBUS <isc>;<cmds>;[<data>[;<cmds>[;<data>]]]...
——► SEND <new isc>;MTL;UNL;CMD <cmds>;[DATA <data>
        [;CMD <cmds>[;DATA <data>]]]...

    SENDBUS <scs>;<cmds>;[<data>[;<cmds>[;]]]...
```

If a statement is detectably of the first form, it is translated.

If it is possible of the second form, it is commented out with a warning message.

SENTER:
Change SENTER to ENTER. ENTER is always serial on your new system.

SENTER is changed to ENTER and the statement is translated as ENTER (with a note that ENTER is always serial on your new system).

SET TIMEOUT:
Remove; use ON TIMEOUT and refer to Timeout Management following in this listing.

Line commented out.

SFK's
See CRT SKF's.

SHORT:
See DIM.

SOUTPUT:
Change SOUTPUT to OUTPUT and use the SERIAL attribute in the ASSIGN statement. (See also the note under ENTER in this keyword list.)

SOUTPUT is changed to OUTPUT and the statement is translated as OUTPUT with a note to use the SERIAL attribute in the ASSIGN statement.

STATUS:

If addressing a device on the bus, use SPOLL.

If doing a STATUS on the **interface**, check for changes in register definitions.

```
STATUS <sc>:<var1>[,<var list>]
```

If <sc> is a constant ≥ 100 and <var list> is not present then the line is translated as:

```
⟶ <var>=SPOLL(<new sc>)
```

If <var list> is present, the TRANSLATOR gives the diagnostic "Multiple vairables not supported.

If <sc> is a constant < 100, it is translated as:

```
⟶ STATUS <new sc>;<var1>[,<var list>]
```

If <sc> is not a constant, the user is cautioned to check the meaning of the statement.

STORE ALL:

Line commented out.

SYSTEM TIMEOUT ON/OFF:

RNEQ

Line commented out.

TBUF$,
TCLOSE,
TDISP:

RNEQ

Line commented out.

TIMEOUT:

RNEQ; refer to Timeout Management.

Timeout Management:

Timeouts no longer take the ON INT branch. Use ON TIMEOUT to trap timeouts.

TOPEN:

RNEQ

```
        Line commented out.
```

TRACE WAIT:

See WAIT.

```
    TRACE WAIT <num exp>
--> TRACE WAIT (<num exp>)/1000
```

TRACK...IS ON/OFF:

```
    TRACK <id>(*) IS ON/OFF
--> RNEQ

    TRACK <sc,ad> IS ON/OFF
--> TRACK <new sc,ad> IS ON/OFF

    Line commented out.
```

Trig Mode:

The caller's trig mode is saved on new context entry and current mode is reset to RAD. On context exit, the caller's mode is restored. On your new system, the called context inherits the caller's mode. The caller's mode is still saved, though, and is restored so any changes made by the called context do not affect the caller.

```
    Inserts the RAD statement at the beginning of every SUB and MLF.
```

TRIGGER

Update device selectors.

```
    TRIGGER <scs>
--> TRIGGER <new scs>
```

TRL:

See ENTER.

TYP:

```
    TYP(<exp>)
--> TYP(#ABS(<exp>),SGN(<exp>)>0)
```

TYPEWRITER ON/OFF:

To achieve the same effect on your new system, use:

`CONTROL KBD,0;1` for `TYPEWRITER OFF`
`CONTROL KBD,0;0` for `TYPEWRITER ON`

This change requires the IO option.

```
    TYPEWRITER OFF
──► CONTROL KBD,0;1

    TYPEWRITER ON
──► CONTROL KBD,0;0
```

The TRANSLATOR requires the IO option for this.

UNCLIP:

```
    UNCLIP
──► CLIP OFF
```

WAIT:

Units on the wait value have changed from milliseconds to seconds. Therefore, divide your old wait time by 1000.

```
    WAIT <num esp>
──► WAIT (<num exp>)/1000
```

WAIT READ,
WAIT WRITE:

RNEQ

Line commented out.

WRITE BIN:

WRITE BIN becomes OUTPUTBIN. Update device selectors.

```
    WRITE BIN <scs>;<list>
──► OUTPUTBIN <new scs>;<list>
```

WRITE IO:

RNEQ

Line commented out.

WHERE:

The current position is maintained in user units. The CP coordinate values are not changed when the viewing transformation changes. Thus the sequence:

```
SETGU
WHERE X,Y
```

returns values set by the most recent graphics output statement. These values are not the GDU coordinates of the plotter pen position.

To obtain the GDU coordinates of the CP execute the following sequence:

```
WHERE X,Y
ASKGU X,Y;Gdux,Gduy
```

One time diagnostic referring to this description.

Appendix C Compatibility with the Series 200

This appendix enumerates the general differences between HP 9000 Model 520 BASIC and Series 200 BASIC.)

It is a major accomplishment when two computer systems, with vastly different architectures, can run the same program and produce the same results. BASIC programs written on the Series 200 family of machines can normally be run on the Model 520 by making only a few minor modifications, such as changing select codes. In most cases, a single program can be written that runs on both machines and produces the same result. Of the over 500 features implemented by the Series 200 family, only the few mentioned here pose any problems when convering programs to the Model 520; most of these represent cases that do not occur in the vast majority of programs. Possible solutions or "work-arounds" for each difference are explicitly described in this appendix, so that you can not only create new programs but also modify your existing programs to run on both systems.

Model 520 BASIC is in many ways a superset of the BASIC supported by the Series 200 family. There are numerous features that are extended in the Model 520 and many which do not appear at all on the Series 200. Our goal was that programs which run without error on the Series 200 family will also run without error on the Model 520 and produce the same result. Only problems which may be encountered when transporting Series 200 programs to the Model 520 are discussed in this appendix; enhancements found in the Model 520 are not covered.

If you enter a line of your Series 200 program which has no equivalent on the Model 520, the system beeps and reports a syntax error. If encountered while doing a GET, the line is turned into a comment and listed as an error line.

To get a hard copy of the errors that occur during a GET, use PRINTER IS ... to direct the output to a line printer.

Your Series 200 program should be SAVEd on a medium which can be read by the Model 520. The LIF_DISC and MS options from the BASIC Binaries Options disc must be loaded. Your need for other binaries, such as the IO and ERRORS_ENG binaries, depends on the BASIC program you are transporting.

General Topics

The most important difference encountered is in the area of the select codes.

Select Codes			
Model 520		**Series 200**	
0	Graphics CRT		
1	CRT	1	Alpha CRT
2	Available to user	2	Keyboard
3	"	3	Graphics CRT
4	"	4	Built-in mass storage (not used on HP 9816)
5	"	5	Powerfail (optional)
6	Internal printer and keyboard	6	Reserved for future use
7	Built-in mass storage	7	Built-in HP-IB
8-23	Available to the user provided I/O expander is in place	8-31	External devices

An efficient way to handle this difference is to use variables or I/O path names instead of constants in your I/O statements. If you also cluster system configuration statements near the beginning of your program, they are easy to find. You need to change only one statement per device.

The built-in functions, CRT, KBD, and PRT can also be used instead of a select code number.

Since the Model 520 does not have a powerfail option, the ON INTR from select code 5 does not service power failures.

Arrays

The Series 200 determines the dimensionality of implicitly dimensioned arrays at prerun; the Model 520 at compile time. Since the Model 520's compiler does not know the dimensionality of Array(*) in statement 10 of the following program, an error occurs. The Series 200 can run a program like this without error.

```
10 PRINT Array(*)
20 Array(1)=1
30 END
```

An easy solution to this problem is to declare all local arrays in INTEGER, DOUBLE, SHORT, REAL or DIM statements.

COM

The time at which COM is deallocated and re-initialized differs in the following circumstances:

- The Series 200 requires a more rigorous matching of COM parameters than the Model 520. The difference is illustrated in the following two programs.

```
10 COM/Abc/ REAL Avogadro      !64 bits of storage
20 Avogadro=6.02E23
30 LOAD "NEWPROG"
40 END
```

```
10 !File NEWPROG
20 COM/Abc/ INTEGER X(1:4)      !64 bits of storage
30 PRINT X(*)
40 END
```

Series 200	Performing the LOAD operation in line 30 of the first program causes a new COM area named Abc to be allocated and intialized. Four zeros are printed.
Model 520	A new template named Abc is imposed on the old COM area when line 30 in program one is executed since the two definitions require the same amount of storage. The COM area is not initialized at this point. 17 631 -8 545 4 264 -11 423 are printed.

- The Series 200 requires that blank COM must always be created in the MAIN program. The Model 520 requires that the blank COM must always appear in the MAIN program. The following example executes without error on the Series 200 but not on the Model 520.

```
10 COM Buck
20 Buck=90
30 LOAD "X1"
40 END
```

```
10 !File X1
20 CALL S
30 END
40 SUB S
50 COM Buck
60 PRINT Buck
70 SUBEND
```

The Model 520 requires that labeled COM statements must appear in the MAIN program of file X1 when LOADed from the first program.

- The Series 200 labeled COM areas don't get reinitialized during a LOAD or a GET if the old labeled COM is present in the MAIN or a subprogram after the LOAD or GET. The Model 520 reinitializes the labeled COM unless it exists in the new MAIN program. The following programs illustrate this point.

```
10 COM/C1/ Value
20 Value=20
30 LOAD "X1"
40 END
```

```
10 !File X1
20 CALL S
30 END
40 SUB S
50 COM/C1/Value
60 PRINT Value
70 SUBEND
```

The output produced by the Series 200 is 20; the output produced by the Model 520 is 0.

- The Series 200 creates and initializes all COM areas at program prerun. The Model 520 creates them on a subprogram by subprogram basis as needed during program execution. The following program runs on the Series 200 but not on the Model 520.

```
10 CALL B
20 CALL A
30 END
40 SUB A
50 COM/Fred/X(100)
60 SUBEND
70 SUB B
80 COM/Fred/X(*)
90 SUBEND
```

  To avoid these problems, explicitly initialize your COM areas after you declare them in the MAIN program.

  To avoid problems encountered with LOAD or GET, use the LOADSUB statement as an overlay mechanism instead.

  If the MAIN programs are not replaced, the behavior is the same on both machines.

Compiled Subprograms

Compiled subprograms are incompatible because of different machine language on the two machines.

CRT

The screen size is 80 characters wide for the Models 216 and 236 and the Model 520 and 50 characters wide for the Model 226. Line-feeds occur at different places during ENTER from the CRT. System parameter defaults, such as CSIZE, are different because of the different CRT sizes and graphics resolution.

The Model 520 reads the character highlight information, as well as the data with an ENTER from the CRT. The Models 216 and 226 do not have character highlights; the Model 236 does, but they are ignored.

The visible representation for CHR$(255) on the CRT of the Series 200 is an inverse video "k". On the Model 520 it is a smudge.

The default WIDTH for the CRT on the Model 520 as the PRINTER/PRINTALL IS printer is infinite in order to support variable screen width in different partitions. On the Series 200 the default WIDTH is the physical screen size. If you specify the width you need, the variable default width problem does not occur.

The default TAB field is 20 characters on the Model 520 and 10 characters on the Series 200. Therefore, comma separators used in PRINT lists may cause different column spacing on the TAB displays. Use TAB(Column) and semicolon separators to have the output appear the same for both computers.

The Model 520 does not support the DELAY parameter of the ASSIGN statement's EOL to the CRT; the Series 200 does.

Error Conditions

System generated messages are not the same so programs which use the text contents of error messages with different wording may not port properly. Different error numbers are generated in a number of statements including DVAL, DVAL$, IVAL, IVAL$, ON CYCLE, ON DELAY, ON SIGNAL, ON TIME, RPT$ and when a bad mass storage unit specifier is given.

Expression Evaluation

The strategy used for expression evaluation is different as demonstrated in the following programs.

```
10 A=10
20 Value=FNVarbind(A)+A
30 PRINT Value
40 END
50 DEF FNVarbind(A)
60 A=5
70 RETURN 20
80 FNEND
```

```
10 A=10
20 Value=A+FNVarbind(A)
30 PRINT Value
40 END
50 DEF FNVarbind(A)
60 A=5
70 RETURN 20
80 FNEND
```

```
10 A=10
20 Value=A+0+FNVarbind(A)
30 PRINT Value
40 END
50 DEF FNVarbind(A)
60 A=5
70 RETURN 20
80 FNEND
```

Model 520	**Series 200**
25	25
30	25
30	30

The Model 520 binds the value of A when the system encounters A on line 20, while the Series 200 sets a pointer to the memory cell storing A and binds the value to A when an operator is encountered which uses A.

You can either avoid side effects altogether or use multiple statements instead of a combining them into one expression.

On the Model 520, 0^0 returns 1; the Series 200 gives an error.

Refer to "Numeric Data Types" in this appendix.

Files

Series 200 BIN files and PROG files are not transportable to the Model 520. ASCII files should be used to transport programs.

When you use the default numeric output formats, the Model 520 may output 7, 12, or 16 significant digits; the Series 200 always outputs 12. An ASCII or FORMAT ON BDAT file which was large enough to hold the data generated by the Series 200 may not be large enough on the Model 520.

Either avoid using the default output formats or create a larger file for the Model 520. You cannot avoid the default format on an ASCII file.

Refer to "Numeric Data Types" in this appendix.

Graphics

The Model 520 can support several plotters simultaneously and uses a logical view surface to facilitate this. The Series 200 can support only one plotter at a time.

The number of pixels on each machine is:

	horizontal	**vertical**
Model 520, with 9020A	512	390
Model 520, with 9020B/C	560	455
Model 216/226	400	300
Model 236	512	390

Human Interfaces

The human interfaces have many differences, from keyboard layout to where edit mode enters a program after an error. These do not affect program execution.

I/O Differences

Default Numeric Output

See Files above.

Error Mailbox

The Model 520 ignores the error mailbox during TRANSFER when I/O path names are cancelled during a LOAD or most GETs. However, if the GET caused program renumbering and the ON ERROR is in the portion of the old program which is not deleted by the GET, such errors are trappable by the new program.

Image Specifier

On the Model 520 a datum is described by the B, W or Y image specifier, however the handshake width is controlled by the WORD/BYTE IO attribute. The default is BYTE.

In the Series 200 the W image specifier causes a 16-bit quantity to be OUTPUT or ENTERed on the GPIO interface. The image specifier describes both the datum and the way it is sent to the interface. The default handshake width is BYTE.

You can use the W image specifier in the ENTER or OUTPUT statements and an I/O path with the WORD attribute to output 16-bit data.

Interface Cards
The interface cards are different; therefore the register assignments for STATUS and CONTROL to interface select codes are different. The ENABLE INTR masks also are not the same.

I/O Path Names
The Model 520 closes I/O path names in COM at STOP, END, LOAD, GET and SCRATCH. The Series 200 leaves them valid but for BDAT file I/O path names, it writes out an EOF to the disc. Thus for BDAT files the end-of-file indicator on the medium agrees with the end-of-file in the I/O path name's table in memory. Refer to the "Master Reset Tables" in the manuals for each machine.

The two systems behave identically with LOADSUB.

You can also reopen the file again after it has been closed.

Keyboard Processing
The Model 520 ignores the error mailbox if an I/O statement is executed from the live keyboard. Multiple errors are remembered.

The Series 200 flushes the I/O path name's error mailbox and can remember only one error.

The Model 520 does not allow subprograms to be invoked from the keyboard; the Series 200 does.

The Model 520 does not support LOADSUB FROM; the Series 200 does, but only from the keyboard.

Line Numbers
The Model 520 lister right justifies line numbers, while the Series 200 left justifies them. This is a problem only if you read an ASCII file with an ENTER and attempt to key in on the first few characters.

Mass Storage
The HP 9121D and the option 10 single volume versions of the HP 9134 and HP 9135 disc drives are supported by the Series 200 but not by the Model 520. Your program may contain some mass storage unit specifiers which will need to be changed.

OUTPUT To the Keyboard
In doing OUTPUT to the keyboard, the Model 520 discards all keys after the first closure key. The Series 200 does not discard them but recommends against using more than one closure key because of unpredictable behavior.

Replace your single OUTPUT statement with multiple OUTPUT statements.

Select Codes

The Model 520 uses the following select codes:

0	Graphics CRT
1	CRT (Same as HP 98x6.)
2-5	Available to the user
6	Internal printer and keyboard
7	Built-in mass storage
8-23	Available to the user provided the I/O expander is in place.

The Series 200 uses these:

1	Alpha CRT
2	Keyboard
3	Graphics CRT
4	Built-in mass storage (unused on the Model 16)
5	Powerfail (optional)
6	Reserved for future use
7	Built-in HP-IB
8-31	External devices

Use variable names or I/O path names instead of constants in your I/O statements so that you need to change only one statement per device. If you cluster these system configurations, they are easy to find.

KNOB Statements

Since there is no knob on the Model 520, there are no KNOBX and OFF/ON KNOB statements.

Matrix Evaluation

Differences in the order of scanning matrices can cause slightly different results due to rounding. This can happen when performing matrix operations which involve more than element-wise arithmetic, for example: RSUM, CSUM and matrix multiplication.

Number Conversions

The routines for converting between decimal and binary can produce slightly different results. The maximum discrepancy is approximately one part in 2^{48}.

This impacts numbers which are used with VAL, VAL$, ENTER and OUTPUT with FORMAT ON or to an ASCII file, READ and INPUT statements.

Numeric Data Types

The Model 520 uses the DOUBLE data type for all whole number constants in the range from -2 147 483 648 to 2 147 483 647 and REAL for legal numbers outside this range. The Series 200 uses the INTEGER data type for constants in the range from -32 768 to 32 767 and REAL for legal constants outside this range.

Since the range of numbers handled by the Series 200 REAL is significantly greater than the range of numbers handled by the DOUBLE data type on the Model 520 (2 147 483 647), constants legal on the Series 200 can cause overflow during expression evaluation on the Model 520. For example:

Series 200 program lines -

```
80 Value=2147483647+1
90 PRINT Value
```

Prints 2.147483648E+9

Model 520 program lines -

```
80 Value=2147483647+1
90 PRINT Value
```

Error 23: DOUBLE overflow

The Series 200 program used REAL arithmetic, while the Model 520 program used DOUBLE arithmetic to evaluate the expression.

Since the DOUBLE data type on the Model 520 uses four bytes of memory for its storage and the INTEGER data type on the Series 200 uses only two bytes, this also affects what is written to FORMAT OFF BDAT files. For example, when the statement:

```
OUTPUT @File;5,"HI" !Assume FORMAT OFF.
```

is executed, the Model 520 uses 4 bytes for the constant 5 (DOUBLE), 4 bytes to specify the string length, 2 bytes for the string.

The Series 200 uses 2 bytes for the constant 5 (INTEGER), 4 bytes to specify the string length, and 2 bytes for the string.

The programs needed to read this data would be different.

If you use a decimal or radix point for your constants they are treated as REAL numbers by both systems.

Softkeys

The Model 520 can assign softkey labels to keys 0 thru 7; the Series 200 can label keys 0 thru 9.

If labels are important, you should adjust your program to work with keys which are labeled 0-7.

Keywords

ASN	Values may differ in least significant one or two bits. Refer to "Number Conversions".
AXES	The Model 520 requires that parameters occur in pairs (0,2,4,6,7); the Series 200 allows them to be deleted singly (0-7).
BUFFER	The Model 520 sets the length of a string which is declared to be a BUFFER to its maximum declared length when the string's storage is created. The Series 200 sets the string length to zero at this time and to its maximum length when the I/O path is set up in an ASSIGN statement. You can work around this difference this way: If you explicitly initialize the string to have a null value, its length is set to zero. Just before doing an ASSIGN which sets up a buffer I/O path, give the string a value whose length is equal to its declared length.
COM	Refer to the discussion of COM in the previous section.
CONTROL	Since the interface cards are different, the register assignments for CONTROL to interface select codes and the register contents bit assignments also change. Multiple entries to address adjacent registers especially need to be rewritten.
CONVERT	The Model 520 copies tables associated with the CONVERT attribute in the ASSIGN statement into a special area of memory and then is able to change the value of the variable specified in the CONVERT without changing the conversion tables. In the Series 200 changes to the variable change the tables. If you must change the variable, use a subsequent ASSIGN statement to update the conversion tables.
COPY	The Model 520 does not support: `COPY<msus> TO <msus>` The DISC_BACKUP and DISC_COPY Utility programs described in the "Utilities " appendix provide this capability. The Series 200 supports this COPY.
CSIZE	Model 520 defaults are 3.3 for the height, 0.6 for the aspect ratio and 0 for the slant angle. The Series 200 defaults are 5 for the height, 0.6 for the aspect ratio and the slant angle is not supported.
DELAY	The Model 520 ignores the DELAY parameter on the ASSIGN statement's EOL attribute to certain devices, e.g. the CRT. The Series 200 never ignores this DELAY parameter.
DUMP DEVICE IS	There is no DUMP DEVICE IS statement on the Model 520.

DUMP GRAPHICS	There is no DUMP GRAPHICS key on the Model 520, but the statement is supported in a program or from the keyboard.
ENABLE INTR	Both systems use zero as the power-up default bit mask and use the previous bit mask after that. The non-zero bits may have different meanings because of different interface cards.
ENTER	The Model 520 always reads up to 256 bytes of data when searching for a terminator. If a parity error occurs and the next operation on the interface is a RESET, an OUTPUT or anything which causes ASSIGN TO * to be invoked, some data between the byte in error and the next terminator may be lost. If ENTER is the next operation no data is lost. The Series 200 quits when the error is detected and consequently loses no data.
ENTER CRT	Line-feeds occur at different places since the CRT sizes are different: 50 characters wide for the Model 226 and 80 characters wide for the Model 520 and the Models 216 and 236. The Model 520 reads character highlight information as well as data. The Models 216 and 226 do not have character highlights, and the Model 236 ignores them.
ENTER USING	On the Model 520 ENTER USING without a % or # loses data up to the next record separator if a formatting or IMAGE processing error occurs. The Series 200 loses no data under these conditions. ENTER USING "nD" can give different results if there are non-numeric characters in the input stream. Use symmetric IMAGE statements for ENTER and OUTPUT to avoid this problem.
FOR ... NEXT	The initial, limit and increment values for the loop are determined at different times. The Model 520 evaluates in this order: limit, increment and initial. The Series 200 evaluates in this order: initial, limit and increment. On the Series 200 a check is made to avoid the problem in statements such as: `FOR I=1 TO I` The difference in evaluation order is a problem only if you use side effects.
GCLEAR	GCLEAR clears only the memory inside the current soft clip limits on the Model 520. On the Series 200 GCLEAR clears the entire graphics memory. See GINIT.

GINIT	After a GINIT, which causes deselection of all plotters, the Model 520 clears the CRT graphics image when the CRT is selected as a default plotter. The PLOTTER IS statement provides a NO GCLEAR option.
	On the Series 200 all graphics statements except GSTORE or PLOTTER IS to an external device cause the GCLEAR after a GINIT.
GRAPHICS INPUT IS	Since the Model 520 has a logical view surface, nondistorted as well as distorted mapping is allowed.
	The Series 200 maps the input device's P1 and P2 to the output device's P1 and P2, resulting in a distorted transformation if the ratios are different.
	Refer to the Series 200 BASIC Programming Techniques manual for a method of getting nondistorted mapping.
GRID	The Model 520 requires that parameters occur in pairs (0,2,4,6,7). Tick marks associated with GRID which are outside the window are visible.
	The Series 200 allows parameters to be deleted singly (0-7). Tick marks associated with GRID which are outside the window are not visible.
KNOBX	Since there is no knob on the Model 520, KNOBX is not supported.
LABEL	The Model 520 clips the output of LABEL at the hard clip limits. No function calls are allowed in the argument list. Block justification of multiple line LABELs is done. Nothing is output for characters in the range 0-31 and 128-159, but the Model 520 does count them when determining the LORG position.
	The Models 226 and 236 clip the output of LABEL at the soft clip limits. Function calls may be in the argument lists. No block justification of multiple line LABELs is done. LABEL outputs blanks for characters in the range 0-31 and 128-159, except for characters 8, 10 and 13 which produce blanks.
LINE TYPE	The default repeat length is 4 on the Model 520. and 5 on the Series 200.
	Refer to the "Useful Tables" in the BASIC Language Reference manual for the Model 520 and under the keyword LINE TYPE in the BASIC Language Reference manual for the Series 200 for the available CRT line types.
LOAD, LOADSUB	See "Files" in the previous section.
LOCK	The LOCK statement's CONDITIONAL parameter is set to 1 on the Model 520 and to 0 on the Series 200 if the specified file is locked.
LOG	Values may differ in the least significant one or two bits. Refer to "Numeric Conversions".

OFF/ON KNOB	Since there is no knob on the Model 520, these statements are not supported.
ON	The order of servicing ON events with the same priority may be different. ONs generated as a result of waiting for overlapped I/O to complete during SUBEXIT are serviced before SUBEXIT on the Model 520, but not on the Series 200. Use different priority levels to force the desired order of interrupt servicing or do not rely on interactions between service routines. The Model 520 requires that the X in : `ON <event> CALL X` be a parameterless subprogram. The Series 200 allows X to be a subprogram all of whose parameters are optional.
ON END	On the Model 520 an ON END is triggered only when an EOF error is reported. On the Series 200 the ON END is triggered whenever an EOF is encountered, even if an EOF error would not be reported.
ON INTR	The Model 520 does not support a powerfail option and hence the ON INTR from the powerfail select code does not work.
ON KEY	The keys which may have soft key labels are different. The Model 520 can assign soft key labels to keys 0 thru 7; the Series 200 can label keys 0 thru 9.
OUTPUT	In doing OUTPUT to the keyboard, the Model 520 discards all keys after the first closure key. The Series 200 does not discard them but recommends against using more than one closure key because of unpredictable behavior. Change your single OUTPUT statement to multiple OUTPUT statements if you encounter this.
OUTPUT USING	If an OUTPUT USING statement contains only the END keyword in the output list, the Model 520 sends both the EOL and EOI on an HP-IB interface; the Series 200 suppresses the EOL sequence and does not generate an EOI.
PEN	The Model 520 interprets PEN 0 as a no-op while the Series 200 interprets it as "complement each pixel". The Model 520 maps a positive but nonexistent pen number for an HPGL plotter into a subset of allowable pen numbers using the MOD function. No range check is performed on the Series 200 resulting in unpredictable states for the plotter. Negative pen numbers use the current linetype on the Model 520 and a linetype of 1 on the Series 200.

PIVOT	The RPLOT local origin is affected by PIVOT on the Model 520 but not on the Series 200.
PLOTTER IS	On the Model 520 PLOTTER IS adds the specified plotter to the list of active plotters. You must use the TERMINATED option to cancel an active plotter. PLOTTER IS does do a GCLEAR unless the NO GCLEAR option is specified.
	PLOTTER IS cancels the old plotter definition and replaces it with the new one on the Series 200. You may insert a GINIT before every PLOTTER IS to work around this. PLOTTER IS does not do a GCLEAR.
PRINT	The CRT tab field width is 20 on the Model 520 and 10 on the Series 200.
RATIO	On the Model 520 RATIO returns the ratio of the logical view surface. The default value for RATIO is approximately 1.23.
	On the Series 200 RATIO returns the ratio of the hard clip limits. The default value for RATIO is approximately 1.33 on the Models 216 and 226 and 1.31 on the Model 236.
READIO	The Model 520 does not have this statement; the Models 226 and 236 do. Consider using the ENTERBIN statement.
RPLOT	The RPLOT local origin is affected by PIVOT on the Model 520; it is not on the Series 200.
SAVE	SAVE creates a DATA file on the Model 520; SAVE ASCII is used to make ASCII files.
	SAVE creates an ASCII file on the Series 200.
SET ECHO	The Model 520 uses a full screen white crosshair The Series 200 uses a small complementing cursor.
	If you specify a marker type of 2 in the SET ECHO statement on the Model 520, you obtain the small cursor.
STATUS	See CONTROL.
SUSPEND INTERACTIVE	The Model 520 does not support the RESET option on this statement since it does not have a RESET key.
TRANSFER	The Model 520 checks for timeouts on a per segment basis and does this in all cases.
	The Series 200 checks for timeouts on a per byte basis and only checks them during an interrupt mode or DMA TRANSFERs if another I/O statement is waiting to use the interface or I/O path name involved in the TRANSFER operation.
	Modify your timeout specifications if necessary when transporting a program.

VIEWPORT	On the Model 520 VIEWPORT immediately rescales the viewport and clips all subsequent graphics output. The Series 200 immediately causes all subsequent graphics output to be clipped at the viewport limits, but the viewport is not rescaled until the next SHOW or WINDOW is executed. If you insert a SHOW or WINDOW statement immediately after the VIEWPORT, the viewport is rescaled.
WRITEIO	The Model 520 does not have this statement; the Series 200 does.

Glossary

access method - the method by which data entries are retrieved from a data base. In IMAGE there are seven: re-read, serial forward, serial backward, directed, chained forward, chained backward, and calculated.

active controller - the HP-IB device which currently controls the bus. Only the active controller is allowed to reconfigure the bus by asserting the ATN bus line. See *system controller.* The active controller is also referred to as the controller in charge (CIC).

alphanumeric display - a mode of the internal CRT which displays the contents of SCREENs independently of the contents of the *graphics display.*

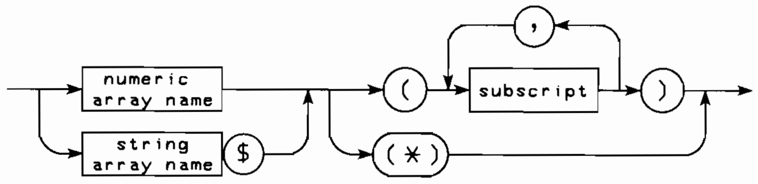
See the ALPHA statement.

angle mode - the current unit of measure for angles. Angles may be expressed in degrees, radians or grads. A circle contains 360 degrees, 2π ($\approx$2*PI) radians or 400 grads.

See the RAD, GRAD and DEG statements.

array - a structured data type that allows more than one data element to be associated with a single name. Each element is accessed by specifying a set of subscripts which denote its position in the array. 1 to 6 subscripts uniquely describe the position of an element. Each subscript must be in the range of DOUBLE and the total number of elements cannot exceed 2 147 483 647. Each array can be of type INTEGER, DOUBLE, SHORT, REAL or string.

To specify an entire array, the characters (*) are placed after the array name. To specify a single element of an array, subscripts are placed in parentheses after the array name.



array identifier - an array name followed by (*), used for accessing all elements in an array collectively.

array plotter - an array which has been initialized by a PLOTTER IS statement to receive graphics plotting data. The plotting data is stored in a form which is useable with AIPLOT, APLOT, ARPLOT, IPLOT, PLOT, RPLOT and SYMBOL.

The array plotter is enabled/disabled with PLOTTER...IS ON/OFF and terminated with PLOTTER...IS TERMINATED.

ASCII - American Standard Code for Information Interchange. A 128-character set represented by 7-bit binary values. (ASCII does not define the value of the eighth bit.)

ASCII file - one of the four types of data files supported by the BASIC Language System. It is generally used to store text or to transfer data to other HP machines. Each character (byte) in the string is specified by an 8-bit code. A string may not be longer than 32 766 characters. INTEGER, DOUBLE, SHORT or REAL values are stored as strings of digits in an ASCII file.

ASCII files cannot be accessed randomly since they do not have fixed length records.

See the CREATE ASCII and SAVE ASCII statements.

aspect ratio - the ratio of the width to height of an area; for example, a character cell or the display area of a plotter.

attribute - **1**: one or more terms or expressions used in the ASSIGN, PRINTER IS, PRINTALL IS and TRANSFER statements which affect the method of Input/Output (I/O). **2**: a characteristic which affects the appearance of graphics primitives (fill color, line type, pen color, and character orientation are examples of attributes).

automatic master data set - a data set which contains only one data item, the key item. It is related to at least one detail data set. When a new key item value is added to a related detail set, a new entry is automatically added to the master. When the last entry containing that key item value is deleted from all related detail sets, the master entry is automatically deleted.

back clipping plane - the 3D soft clipping boundary which is parallel to the UV plane and farthest from the view plane. If the soft clip boundaries are enabled, all subsequent graphics primitives behind this plane are not displayed.

See the CLIP statement.

background partition - a partition without its private SCREENs and the keyboard (KBD) attached. See *partition* and the ATTACH statement.

BASIC name - used to name variables, subprograms, line labels or I/O path names. It is a literal 1 thru 15 characters long. The first character is an upper case ASCII letter (A thru Z) or an upper case national character from the range CHR$(161) thru CHR$(254). The remaining characters can be ASCII or national lowercase letters, digits or the underscore character (CHR$(95)).

BCD - see ''binary-coded decimal''.

BCD file - one of four data file types supported by the BASIC Language System. In BCD files, numeric data is represented as a sequence of values in Binary Coded Decimal. It is provided primarily for data interchange with the HP 9835/45 family of desktop computers. Its format is exactly the same as an HP 9835/45 DATA file.

BCD files can be serially or randomly accessed.

Unless data interchange is a concern, it is recommended that DATA files be used instead of BCD files. DATA files support the DOUBLE data type, and do not require conversion of numeric data between the IEEE internal binary form and the BCD file form.

See the CREATE BCD statement.

BDAT file - one of four types of data files supported by the BASIC Language System. In BDAT files, data is represented as a sequence of values in internal form. BDAT files are good for general-purpose data storage. They are significantly quicker to load and store and more compact than BCD, DATA or ASCII files.

Numeric quantities are stored in binary form, using forms which depend on the type of value supplied. No type information is stored with the data. This means that data written as a REAL number could be incorrectly read back as four INTEGERs, two DOUBLEs, part of a string, or a REAL value. For this reason, BDAT files require more care on the part of the user than other data file types.

BDAT file can be serially or randomly accessed.

BDAT files can also be created with a defined-record length of one, allowing random-byte accessing.

If the directory format is LIF or 9845, a sector is created at the beginning of the file for system use. This sector does not affect the created length of the file and cannot be accessed by BASIC programs.

See the CREATE BDAT statements.

best fit non-distorting mapping - when the logical view surface is mapped to the largest possible portion of a physical plotter's display area while maintaining the aspect ratio of the logical view surface.

binary-coded decimal - a system of representing numbers in which each digit (and sign) of a decimal number is represented by a group of four binary digits. Abbreviated BCD.

BIN file - a file containing a set of BASIC features written in the machine's internal code which adds enhancements to the existing operating system.

See the LOAD BIN, RESTORE BIN and STORE BIN statements.

bit - a contraction of BInary digiT. A bit can have a value of 0 or 1.

bit mask - a numeric expression which is evaluated to the decimal representation of a binary number, or a string expression identifying a literal pattern of bits in a binary number.

blank - see *space.*

buffer - a segment of contiguous random-access memory locations used for temporary storage of input and output data. A numeric array or string variable can be used as a buffer by appending the term BUFFER to the array or string identifier when the variable is dimensioned. Buffers cannot be dimensioned with the ALLOCATE statement.

bus command - a value transferred on the HP-IB DIO lines with the ATN line true (1). Bus commands provide bus and device control.

bus reconfiguration - I/O statements which specify one or more HP-IB primary device addresses, or, for which such addresses are implicitly specified, *reconfigure* the bus prior to output or input. Reconfiguration consists of changing which device is the current *talker* and which device or devices are the current *listeners*. The HP-IB interface card involved must be the *active controller* to reconfigure the bus. It need not be the *system controller*.

For output, the messages sent are:

ATN, MTA, UNL, LAG, ATN, DAB

For input, the messages sent are:

ATN, TAG, UNL, MLA, (LAG), ATN, Read DAB

On output, the output data is sent to all specified devices on the same bus. The Listen Address Group (LAG) consists of one or more device addresses. On input, the data is read from the first specified device (TAG). If multiple devices are specified, the second and succeeding devices are addressed as listeners (LAG). The input LAG message is omitted if only a single device is specified.

See the *HP-IB Mnemonics* table.

byte - a unit of data storage consisting of 8 bits. A byte can represent any character, an unsigned numeric value in the range 0 thru 255 or a signed numeric value in the range −128 thru 127.

calculated access - a method for locating an entry in a master data set. The value of the key item is transformed via a hashing algorithm to produce a primary data entry record number.

calling context - the context from which the currently executing context gained control. The MAIN context has no calling context.

capacity - the defined capacity of a data base data set is the number of data entries specified for the set in SCHEMA or DEFINE. The actual capacity differs if the set has been extended with DBEXTEND. The capacity is an integer value in the range of a DOUBLE. The defined capacity of a master data set should be a prime number for optimum performance.

carriage-return - or CR: ASCII character code 13. See the *Character Codes* and *Control Character Effects* tables.

chain - a series of paired record numbers (pointers) stored in the header of a data entry's media record. Each pair of pointers designates the previous and next entry on the chain. Entries on chains are related by one of three characteristics: free records (empty entry), synonym (duplicate hash value), and same key value (detail chain).

chained access - a method of locating detail data entries which retrieves the next or previous entry having the same synonym (in a master data set) or search item value (in a detail data set).

character cell - **1:** on the internal printer (PRT) and CRT, a rectangular dot matrix within which characters are printed or displayed. The PRT cell is 7 x 12 dots and the CRT cell is 9 x 15 dots. **2:** in graphics, a parallelogram which establishes mapping from font coordinates to current units (for GTEXT) or GDU's (for CSIZE).

clip - the elimination from view of all graphics primitives or parts of primitives which lie outside the clipping boundaries.

See the CLIP statement.

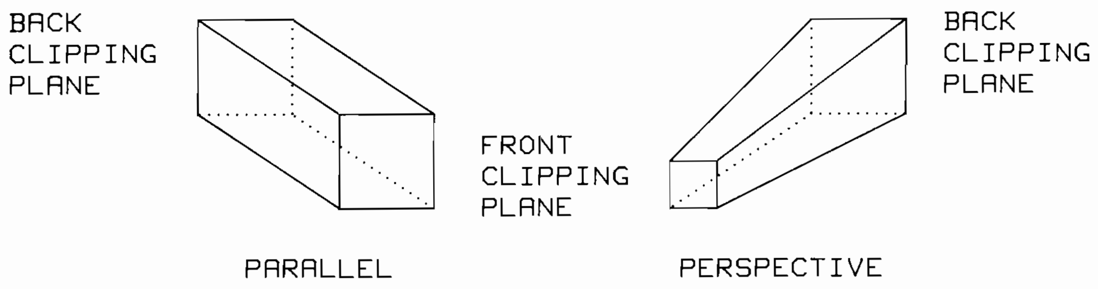
clip area - that plotting area within the clipping boundaries.

clipping boundaries - boundaries on the logical view surface or volume beyond which graphics primitives cannot extend.

There are two types of clipping boundaries: hard clip boundaries and soft clip boundaries. The hard clip boundary is the edge of the logical view surface mapped to a specfic device. The soft clip boundary is the limit of the window as defined by the WINDOW statement or the clip area as defined by the CLIP statement.

See the CLIP, LIMIT, PLOTTER IS, VIEWPORT and WINDOW statements.

clipping volume - a three-dimensional subset of the XYZ coordinate system outside of which output primitives are not displayed. The clipping volume's sides are determined by the sides of the window, and its ends are the front and back clipping planes. If the projection type is parallel, opposite sides of the clipping volume are parallel to each other, forming a parallelepiped. If the projection type is perspective, the clipping volume has the shape of a truncated pyramid (only the top and bottom are parallel to each other). Here are examples of these clipping volumes.



code data type - an enumerated data item type with a *code value* equal to one of 32 user-defined 16-character strings. Each defined value corresponds to a *code number* in the range 1 to 32. The strings are specified in SCHEMA and DEFINE, and are stored in a code table in the root file. The code table can be accessed via DBINFO mode 108. Only the code number is physically stored in a data entry.

color map - the location in a CRT's memory where the definitions of that CRT's colors is stored. Not all CRTs have color maps.

color map location - one definition of a color in a color map. It is referenced by a pen number.

color map mode - a display mode of the CRT in which pen numbers refer to entries in the color map to determine the color displayed. You can specifically define the colors with which displays are created as opposed to using system defaults.

command - a BASIC language statement which can only be executed from the keyboard. See also *bus command.*

comparison operator - a dyadic operator which returns a boolean value based on the result of a relational test of the preceding and succeeding expressions. A boolean value is an integer; zero if false, one if true. The relational operators are:

<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to
<>	not equal to (less than or greater than)
=	equal to

See LEXICAL ORDER for a discussion of string comparision.

COM variables - declared in COM statements. COM variables are accessible in all contexts of a given program and will retain their values during LOAD and GET operations if the new MAIN program contains matching COM statements. COM variables may be referenced by different names in different contexts.

context - a term often used as a synonym for "subprogram" to emphasize that the MAIN program is a context as well as SUBs and multi-line DEF FNs. This definition is adequate when talking about a program that is not running; but when describing a running program, particularly one that is recursive, a more precise definition is necessary.

A context represents a particular invocation of a subprogram. It includes the execution state of the subprogram itself and the binding between memory locations and variable names resulting from a particular call to that subprogram. Different subprograms represent different contexts. Furthermore, a given subprogram may be represented in several contexts at a given time because of recursion. A local variable in that subprogram may have a different value for each level of recursion. In a given context, the variable has only one value.

control character - a member of a character set which may produce some action in a device other than a printed or displayed character. In ASCII, control characters are those in the code range 0 thru 31, and 127. Control characters are generated by simultaneously pressing a displayable character key and (CTRL).

See the *Control Character Effects* table.

control code - the numeric value of the binary pattern of a control character.

current clip boundaries - see *clip boundaries.*

current partition - the partition in which a program or statement is running.

current position - the point in the XYZ coordinate system relative to which subsequent graphics primitives are positioned.

current units - units with which graphics output primitives are measured. They may be Graphic Display Units (GDUs) or User Defined Units (UDUs).

data base - a collection of logically related files. A *root* file contains structural information. Master and detail *data set* files contain the data and relational information.

data entry - that portion of a data set media record which contains the values for the data items which are stored in that data set.

DATA file - one of four types of data files supported by the BASIC Language System. DATA files are used for general data storage. Although a little bulkier and slower to use than BDAT files, they do not require the particular care during reading or writing that BDAT files demand. Since each data item is preceded by a type field, the system does type checking and numeric conversion is performed automatically.

DATA files can be accessed serially or randomly.

See the CREATE DATA statement.

data files - files of type ASCII, BCD, BDAT and DATA can store data from, or assign data to, program variables. DSET files also store variable data, but are only accessed with IMAGE/Data Base statements.

BIN, DIR, PROG and ROOT files are not data files.

data item - the smallest named element in a data base. It is stored in a field of a data entry and can contain a value which corresponds to a program variable. An item can be an entire array, but is always accessed as a single unit.

data set - a collection of data entries. All data entries in the same set have identical structure, but not necessarily identical data. There are two types of data sets, master and detail. See *data set file.*

data set file - a file of type DSET containing all of the media records for a single data set. Each media record can contain one data entry.

default mode - a display mode of the CRT in which additive primaries are used to create colors on the CRT.

defined record - the smallest unit of a file which you can access randomly. The actual length in bytes of the record is determined when you create the file. Defined records are an even number of bytes in size for file types BCD and BDAT. If you select an odd number for the record size, it rounds up to the next even number. Defined record can be an odd number of bytes for BDAT files. All defined records in a file are the same size. ASCII files do not have *defined records.*

detail data set - a data set containing entries for one or more search and/or detail items. If the data set is linked to one or more master data sets, the detail entries containing the same key value are chained. Detail data sets support only re-read, serial, chained and directed access methods.

detail data item - in a master data set, a data item other than the key item. In a detail data set, a data item other than a search item, having no pointers linking its entry to a chain.

device coordinates - the coordinate system used to define the smallest addressable units of a plotter or a graphics input device. Each device has a unique range of device coordinates which depend on the resolution of the device hardware.

device selector - a numeric expression specifying the source or destination device of an I/O operation. A device selector can be either an interface select code or a combination of an interface select code, a primary address and zero or more secondary addresses. To construct a device selector with a primary address, multiply the interface select code by 100 and add the primary address.

Secondary addresses are appended by multiplying the device selector by 100 and adding a secondary address. Each additional secondary is added to the current value of the device selector in the same fashion. A device selector, rounded to an integer, can contain a maximum of 16 digits, or six secondary addresses.

When a device selector contains an odd number of digits, the left-most digit is the interface select code. For an even number of digits, the left-most two digits are the interface select code. For example, 30217 denotes interface 3, primary address 02, and secondary address 17. Device selector 1516 denotes interface 15 and primary address 16.

Although the primary address is normally used for HP-IB addressing, the internal I/O devices listed in the following tables use the same notation.

With statements requiring the GRAPHICS Option:

device selector	refers to
1	LIGHT PEN, GRAPHICS or INTERNAL device drivers
100	the graphics display of the internal CRT
101	memory plane #1 of the internal CRT
102	memory plane #2 of the internal CRT
103	memory plane #3 of the internal CRT
602	ARROW KEYS device driver

With all other I/O statements:

device selector	refers to
100	the DISPLAYS SCREEN of the internal CRT
101	SCREEN 1
1xx	SCREEN xx
199	SCREEN 99
601	internal printer (PRT)
602	internal keyboard (KBD)

digitize - the process of obtaining an X,Y,Z coordinate set based on the position of the locator of an input device. The system waits for a digitize button press on a graphics input device and then samples the locator position.

digitize button - a button, key or switch on a graphics input device which causes the input device to sample its locator's position and assign the coordinates to program variables if a DIGITIZE statement is in progress.

For the internal keyboard, the digitize key is RETURN. For the light pen, the digitize switch is on the barrel of the pen.

dimension - an element in the vector which defined an array element's location in the array; for example, if OPTION BASE is 0 -

```
Sample(0,6,5)
```

specifies the element in the first position of the first dimension, the seventh position of the second dimension and the sixth position of the third dimension of the three-dimensional array `Sample`.

direct bus I/O - I/O statements which specify only an HP-IB interface select code, or for which statements an interface select code is implicitly specified, do not reconfigure the bus prior to output or input. See *bus reconfiguration*.

For output, the interface must be the current talker (have received an MTA command), and it sends data to the device or devices which are the current listeners. The bus messages sent by the I/O statement are summarized in the following table.

	Interface is Active Controller	**Interface is not Active Controller**
MTA already received	Send DAB	Send DAB
MTA not yet received	Error	Wait for MTA Send DAB

For input, the interface must be a listener (have received an MLA command), and data is read from the device which is the current talker. The bus message received by the interface are summarized in the following table.

	Interface is Active Controller	**Interface is not Active Controller**
MLA already received	Read DAB	Read DAB
MLA not yet received	Error	Wait for MLA Read DAB

See the *HP-IB Mnemonics* table.

directed access - a method of retrieving a data entry in which the entry's record number is specified.

DIR file - see *directory.*

directory - a file used to catalog other files on a mass storage medium. Each directory contains entries for its own unique files. The directory's information includes each file's name, type, length, location and protection.

The size of a directory is determined by the number of file entries permitted. See the CREATE DIR statement. SDF directories can extend automatically as necessary to add file entries.

A directory can be protected like any other file. You must have WRITE capability to create or purge a file in a directory, READ capability to catalog a directory and MANAGER capability to purge a directory.

directory format - the organization of files on a mass storage medium, determining such things as length of file names, protection capabilities and file extension capabilities. The BASIC Language System supports Structured Directory Format (SDF), Logical Interchange Format (LIF) and HP 9835/45 format (9845).

LIF and 9845 support require the loading of BIN option files.

dithering - the mixing of pixels to approximate colors other than those available within each pixel.

DOUBLE - a numeric data type represented internally as a 32 bit 2's-complement integer. The range of a DOUBLE is −2 147 483 648 thru 2 147 483 647.

driver - an HP-supplied system program which converts BASIC I/O requests to the form required by a device or interface. Drivers are supplied in two varieties: Device drivers and Interface drivers.

Device drivers convert BASIC I/O requests to a form compatible with the intended peripheral device. The request is then passed to the Interface driver supporting the interface to which the device is connected. The Device driver is specified by a DRIVER I/O attribute or a media specifier.

Interface drivers convert both BASIC and Device driver requests to a form compatible with the HP-IO interface card specified or implied by the I/O statement. Interface drivers do not generally support any device features unless only a single device type can be connected to the interface.

dyadic operator - an operator which performs an operation using a preceding and succeeding argument. The dyadic operators are all of the *comparision operators* and the following:

+	Addition
−	Subtraction
*	Multiplication
/	Division
^	Exponentiation
	& String concatenation
DIV	Truncated quotient of a division
MOD	Remainder of a rounded division
MODULO	Remainder of a truncated division
AND	Logical AND
OR	Logical inclusive OR
EXOR	Logical exclusive OR

echoing - the process of indicating the position of the locator on a graphics input device. Echoing can take a number of forms depending upon the devices being used and the type of input function.

Typically the locator position of an input device such as a light pen is echoed using a marker such as a crosshair on a CRT or the physical pen on a hard-copy plotter.

enabled device - a graphics device for which a GRAPHICS INPUT...IS ON or PLOTTER...IS ON statement is in effect. You must initialize graphics devices before enabling them.

end of file - abbreviated **EOF**; a condition signalled by mass storage device drivers when the end of data is detected in a file. The physical implementation varies with the file type.

An ASCII file has a single EOF. Each string output to an ASCII file causes an EOF to be written after that string. The EOF is stored as a string having a length of −1.

A BDAT file has a single EOF pointer designating the next record in the file after the end of data. This pointer is advanced by one record when data is output to the current EOF record. The pointer can be arbitrarily reset to any record in the file with a CONTROL statement or an output statement specifying an END.

In BCD and DATA files an EOF is a special data type header. When these files are created, an EOF is written at the start of each record. When data is written to a record the type header is changed from an EOF to the data type written. It can be reset to an EOF by an output statement specifying an END.

EVENT - used by cooperating programs in separate partitions for synchronizing processes or use of resources. It is an internal integer variable which has the range of a DOUBLE. It is a signalling device sometimes called a semaphore. It is best defined in terms of the operations CREATE EVENT, CAUSE EVENT, EVENT LEVEL, ON/OFF EVENT and WAIT FOR EVENT.

expression - **1**: a combination of operators, constants, variables and expressions which evaluates to a numeric value (see *numeric expression*). **2**: a combination of operators, literals, variables and expressions which evaluates to a character string (see *string expression*). **3**: a legitimate combination of one or more secondary keywords and a numeric or string expression.

file - a collection of related records treated as a unit. The BASIC Language System has seven types: ASCII, BCD, BDAT, BIN, DATA, DIR and PROG. IMAGE Data Base Management has two types: DSET and ROOT.

file name - a literal which uniquely identifies a file in a directory.

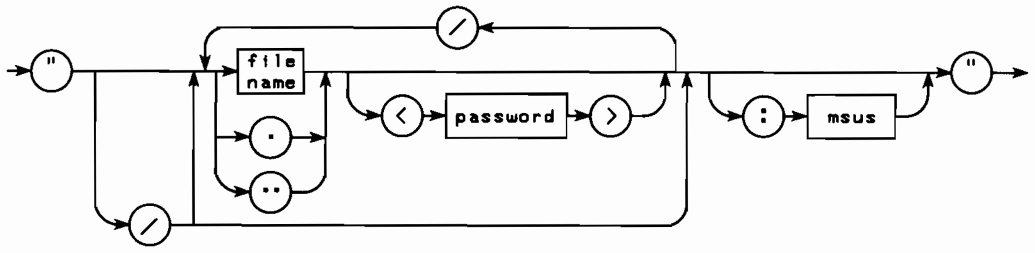
The name consists of characters excluding the slash (/), less-than (<) and colon (:). A single period (.) or double period (..) cannot be used as a file name. Leading and embedded blanks are deleted. Names with a length less than the maximum for the directory format can be considered to have trailing blanks.

In a Structured Directory (**SDF**), file names are 1 to 16 characters long. If compatability with the HP-UX file system is desired, you should limit the length of SDF file names to 14 characters.

In a Logical Interchange Format (**LIF**) directory, file names are 1 to 10 characters long. LIF is supported by many HP computers. If strict LIF compatability is desired, you should also restrict LIF file names to the upper case ASCII letters (A thru Z), the ASCII numerals (0, 1 thru 9), the underscore (_), and always begin the file name with a letter.

In an HP9835/45 directory (**9845**), file names are 1 to 6 characters long. If strict compatability with the HP 9835/45 is desired, you should also restrict 9845 file names to exclude the ASCII NUL (CHR$(0)), quote (") and the smudge (CHR$(255)) characters.

file specifier - a string expression which designates the file affected by a statement. The literal form of file specifier is:



The entire syntax diagram applies only to files in structured (SDF) directories. A maximum of 6 file names, ".''s and "..''s are allowed.

If the file's directory format is Logical Interchange Format (LIF) or 9845, only one file name and no "/''s, ".''s or "..''s can be given in the file specifier.

Item	Description	Range Restrictions
file name	literal	(see file name in Glossary)
password	literal	(see password in Glossary)

msus	literal;
	Default = MASS STORAGE IS device
		(see media specifier in Glossary)
		Note that msus may be REMOTE.

Example *file specifiers*:

```
"FNAME"
"/FNAME"
"."
"../.."
"FILE:INTERNAL"
"DIR1/DIR2/FNAME:HP9885,501"
"DIR1<PASS1>/DIR2/FNAME<PASS2>:INTERNAL"
"DIR1/DIR2/FNAME:REMOTE"
```

Notes

. signifies the current directory.

.. signifies the directory pointing to the current directory.

If the file specifier begins with a "/", the path to the file is started from the mass storage medium's root directory. Each succeeding "/" denotes that the previous file is a directory which contains the next file. If a file name is given without a preceding "/", the path to that file is started at the current directory, not the root directory.

A maximum of 6 levels of files may be included in a file specifier at one time.

When accessing a file, the file name's password permits access to capabilities protected by that password; such as reading from or writing to the file. See the PROTECT statement.

When creating a file or directory or copying a file, the password associated with the file or directory is used to protect the manager and write capabilities of that file or directory.

Unless creating a directory, a password associated with a directory is used to gain access through that directory.

Here is a typical file specifier.

```
"/FILE1<PASS1>/FILE2<PASS2>/FILE3<PASS3>"
```

The first "/" denotes that FILE1 is found through the root directory of the current MASS STORAGE IS device. The second "/" denotes that FILE1 is a directory which contains FILE2. The third "/" denotes that FILE2 is a directory which contains FILE3. Since there are no succeeding file names, FILE3 is the file to be operated on by the statement.

Specifying PASS1 enables access through directory, FILE1. Specifying PASS2 enables access through directory, FILE2. The meaning of PASS3 depends on the operations being performed on FILE3. If FILE3 is being created, PASS3 protects the manager and write capabilites of FILE3. If FILE3 has already been created, PASS3 provides a previously defined set of access capabilities.

fill color - the color used to fill polygonal areas. The fill color is specified by an AREA COLOR or AREA INTENSITY statement, or by data values in IPLOT, PLOT, RPLOT, or SYMBOL arrays.

font - a set of characters which are sent to the graphics display when the corresponding character codes are output in a GTEXT or LABEL statement. Fonts are system-defined or user-defined.

See the Useful Tables section for a listing of system-defined fonts.

See FONT...IS, FONT IS and FONT...IS TERMINATED.

font coordinate system - the coordinate system used to define individual characters in a font or symbol.

See the FONT...IS and SYMBOL statements.

font window - determines how much of the first quadrant of the font coordinate system is mapped to a character cell. For system-defined fonts the window is 9 by 15. For user-defined fonts the window is specified in the descriptor array of the FONT...IS statement.

See the FONT...IS and SYMBOL statements.

foreground partition - the partition which has its private SCREENs and the system keyboard (KBD) attached. See *partition* and the ATTACH statement.

formal parameter - variables that are specified in SUB or DEF FN statements. Formal parameter specifications amount to a special form of variable declaration since they include type information as well as a name for each parameter and may be used as variables in the subprogram or single-line functions. When the subprogram or single-line function is invoked, values for these variables are supplied by the invoking statement.

See the CALL, SUB, DEF FN and FN statements

front clipping plane - the 3D soft clipping boundary which is parallel to the UV plane and closest to the view plane. If the soft clip boundaries are enabled, all subsequent graphics output in front of this plane is not displayed.

See the CLIP statement.

GDU - see Graphic Display Unit.

Graphic Display Unit (GDU) - a unit-of-measure for graphics primitives which is not affected by the current viewing transformation. A single GDU is equal to 1/100 of the shortest side of the logical view surface.

Although a GDU along the X axis is equal to a GDU along the Y axis on the logical view surface, this may not be true on a physical plotter. See "BASIC Graphics Programming Techniques for the Model 520" for more information.

graphics display - a graphics output device which displays or plots visible graphics primitives such as lines, polygons, filled areas, graphics text, and labels.

See the GRAPHICS and PLOTTER IS statements.

graphics input device - any device which provides a locator position, or can supply GKEY or button device input.

Graphics input devices include data tablets and the computer's arrow keys and light pen.

See GRAPHICS INPUT IS.

graphics primitive - the basic element of a graphics output. The primitives which BASIC Graphics supports are: MOVEs, DRAWs, multi-line PLOTs, POLYLINEs, POLYGONs, LABELs and GTEXT.

All BASIC graphics output statements use these primitives to create their output.

gtext - a string of alphanumeric characters which are converted to vectors defined by the current font and are output to a gtext plane.

See the GTEXT statement.

gtext plane - a plane in the XYZ coordinate system on which gtext is output. It is defined by the GTEXT ORIENT statement and the current position.

See the GTEXT and GTEXT ORIENT statements.

hard clip boundaries - the boundaries of the logical view surface beyond which graphics primitives cannot extend.

See LIMIT and PLOTTER IS.

half-word - a unit of stored or input/output data consisting of 16 bits (two bytes).

See INTEGER.

hashing - see calculated access.

HPGL (Hewlett-Packard Graphics Language) - a command set for use with graphics devices consisting of two-letter ASCII mnemonics followed by parameters. HP graphics software statements automatically generate HPGL commands appropriate to the device and operation.

HP-IB - the Hewlett-Packard Interface Bus. Hewlett-Packard's implementation of the I.E.E.E Standard Digital Interface for Programmable Instrumentation, as defined in the IEEE document IEEE Std 488-1978, IEC Publication 625-1 and ANSI MC1.1. In the "Useful Tables" section, see the *HP-IB Mnemonics* table for a discussion of HP-IB messages and mnemonics. See the *Character Codes* table for a list of HP-IB command character codes.

HP-IO - Hewlett-Packard Input/Output. A computer I/O architecture standard which defines I/O processor control, physical channel configuration, and interface cards which connect computers which support HP-IO to other computers, peripherals, terminals, instruments and data communications equipment.

HSL - Hue, Saturation and Luminosity, a system for defining a color.

hue - the wavelength of light associated with a color.

IMAGE name - used to name data bases, data sets, data items, and data item pseudonyms. It is a literal, 1 to 16 characters long, consisting of characters from the following table. The name is terminated by a blank, semicolon, or end of string. If fewer than 16 characters are specified, trailing blanks are presumed.

Characters	Character Codes	Leading	Embedded	Trailing
space	32	NO	NO	YES
0 thru 9	48 thru 57	NO	YES	YES
A thru Z	65 thru 90	YES	YES	YES
_	95	NO	YES	YES
a thru z	97 thru 122	NO	YES	YES
À thru Ï	161 thru 167	YES	YES	YES
Ù thru Û	173 thru 174	YES	YES	YES
Ý thru ý	177 thru 178	YES	YES	YES
Ç thru ñ	180 thru 183	YES	YES	YES
â thru ß	192 thru 222	YES	YES	YES

initialize - the process of preparing a mass-storage medium for data storage. This consists of the following steps:

1. Sector patterns (pre-amble, data area, post-amble) are written to every sector of the medium. Some media are factory-formatted and do not require this step.
2. A logical sector number is written to each sector pre-amble. If the *interleave* factor is not 1, the logical number is not the same as the sector's physical location.
3. A test data pattern is written to the data area of each sector.
4. The test patterns are validated, and if a sector contains bad data, the sector is marked defective. Defective sectors are not used for data storage. A spare sector may be substituted, or, the entire track may be marked defective and be replaced by a spare track, or made inaccessible by re-numbering subsequent tracks.
5. The specified or defaulted volume label is written.
6. A root directory of the specified or defaulted format and size is written.

See the INITIALIZE statement. Steps 1 through 4 do not apply to ":MEMORY".

inner product - the sum of the products of corresponding elements in two vectors.

See the DOT function.

interface select code - a numeric expression that selects an an interface for an I/O operation. An interface select code is a subset (the first one or two digits) of a *device selector*. The range of an interface select code is 1 thru n. The value of n, never greater than 23, depends on your system's I/O configuration. Your computer can have 1 thru 3 I/O Processors (IOP). Each IOP controls 8 select codes. The maximum value of n is therefore (8 × <number of IOP's>) − 1. Select code zero is reserved. Specifying a select code for which no interface is installed is an error.

INTEGER - a numeric data type stored in 16-bit two's-complement binary form. Each INTEGER variable requires four bytes of memory. Elements of INTEGER arrays requires two bytes of memory. The range of an INTEGER is −32 768 thru 32 767.

interleave - a step in disc media initialization in which a logical sector number is written into each physical sector preamble. During sequential access to logical sectors, the physical record number is computed by the disc controller by adding the interleave factor to the physical sector number of the current logical sector.

In disc operations which require computation between sequential logical sector accesses, an optimum interleave factor minimizes latency. Latency is the time consumed when the disc controller has to wait for the desired sector to rotate under the read/write head.

internal CRT - the Cathode Ray Tube display mounted on the top of the HP 9000 Model 520.

internal mass storage device - that device which is addressable with the media specifier ":INTERNAL". It specifies the 5¼ inch (9130K) flexible disc drive installed in the HP 9000 Model 520. The 9130K is also addressable with ":CS80,7,1".

internal printer - the thermal line printer installed in the HP 9000 Model 520.

interrupt - a hardware logic condition which occurs within an interface card, or is signalled to an interface card by a device connected to it. An interrupt may use special circuitry on the interface or may be an otherwise ordinary, but unexpected, I/O transaction. If an executing program has an active ENABLE INTR/ON INTR or ENABLE INTR/WAIT FOR INTR statement pair specifying the same interface, the BASIC Language System logs the interrupt request from the card, and enabled the program branch. Simulated ON INTR branches can be forced with the CAUSE INTR statement.

I/O - abbreviation for Input/Output.

I/O attribute - one of several expressions which you can specify in the ASSIGN, PRINTALL IS and PRINTER IS statements which control the I/O characteristics of the interface, device, file, or BUFFER specified by the statement.

I/O path name - a data type associated with a BUFFER, interface, device, devices or file. An I/O path name is a BASIC name preceded by an @ (commercial at) symbol. The @ is shown separately in all statements which allow the use of I/O path names. You cannot dimension (DIM) or ALLOCATE an I/O path name, but you can declare it in a COM statement.

An I/O path name consumes approximately 160 bytes of memory when it is created and opened by the ASSIGN statement. This memory contains information about the I/O path name, all I/O attributes specified or defaulted in the ASSIGN statement and up to 100 registers. The registers associated with an I/O path name are numbered in the range 0 thru 99, and are accessible with the STATUS and CONTROL statements.

Julian date - internal representation of the date. It is the number of days since the base point, 24 Nov −4713.

Julian time - internal representation of the time. It is the number of seconds in a day (86 400) times the Julian date, plus the number of seconds since midnight of the current day. It is also the number of seconds since midnight on the morning of the base point, 24 Nov −4713.

key item - in a master data set, a data item which can be located by calculated access.

keyword - a literal composed of uppercase ASCII letters that has a predefined meaning to the computer.

label - **1:** a unique BASIC name, followed by a colon, which precedes a program line. **2**: a string of alphanumeric characters output to the graphics display using the LABEL or LABEL USING statements. It is justified about the current position specified by LORG. Its size and orientation is specified in GDUs and is not affected by viewing operation. Labels are not clipped at the soft clip boundaries. See CSIZE, LABEL, LABEL USING, LORG and LDIR. **3**: a string which names a mass storage volume. See *volume label.*

left-handed coordinate system - any three-dimensional coordinate system that can be rotated such that the positive X-axis is directed to the right, the positive Y-axis is directed up, and the positive Z-axis is directed away.

LIF - Logical Interchange Format; the HP standard disc media directory format for mass storage files. Although intended for use on small capacity portable media such as flexible discs, you can initalize any media type as LIF in the BASIC Language System. You can also create any BASIC file type in a LIF directory; however, only ASCII files are defined by the standard to be compatible with other HP computers which support LIF.

line label - a BASIC name which identifies a program segment. It is followed by a colon (:) and precedes the BASIC statement in the line it identifies. The following are examples of line labels in various BASIC statements:

```
Label: FOR Index=Start TO Stop
Report_error: !This GOSUB routine handles errors.
Format_37: IMAGE 5(3D.ESZZ,4X," converted."),%
Integer: PACKFMT Integer
Label_only: !
```

line number - an integer constant which identifies a program line. The range of a line number is 1 thru 999 999. See the REN statement.

listener - an HP-IB device which is configured to receive data bytes (DAB) sent on the bus. A device is normally made a listener by sending a Listen Address (LA*) or Listen Address Group (LAG) message specifying the device's listen address. An HP-IB interface is made a listener when it sends or receives its own listen address (MLA). Some devices can be locally configured by their operator to "listen always".

literal - a string constant. Literals are typically composed of characters typed on a keyboard or ASCII data. A syntax element of a BASIC statement which requires a literal must appear as characters in the displayed or listed program and cannot be a string expression. When quote marks (") delimit a literal, the quotes are not part of the literal. If the contents of the literal must include the quote character, use two quotes for each literal quote.

live keyboard - a mode of the BASIC Language System which allows you to evaluate numeric computations and execute most BASIC commands from the keyboard while a program is running in the foreground partition.

local origin - a point which is temporarily considered the origin of plotting. All points to be plotted to are taken as displacements from the local origin. Local origins are used with ARPLOT, POLYGON, POLYLINE, RECTANGLE and RPLOT.

The local origin can be updated by executing any graphics output statement other than one using a local origin.

local variable - a variable in a subprogram or function that is not listed in a COM statement or in the formal parameter list of the caller. It cannot be accessed from any other context. Storage allocated for a local variable is temporary and is deallocated upon return to the calling program.

locator - a physical device which indicates position on a graphics input device.

Examples are a light pen or a graphics tablet stylus.

logical view surface - a conceptual model used in graphics plotting. In this model, the logical view surface is a generalized two-dimensional plotter to which all graphics are output. All physical plotters then derive their output from this central image.

luminosity - the amount of light emitted, transmitted or reflected by a color.

maintenance word - a data base protection code defined by DBCREATE and required for all subsequent DBCREATE, DBERASE, DBEXTEND and DBPURGE utility operations on the same data base. The maintenance word is 16 characters long and is composed of any characters in the set except the semicolon. Characters after a blank or semicolon are ignored. If less than 16 are specified, trailing blanks are assumed.

manual master data set - a master data set which can contain detail items in addition to the key item. A manual master need not be linked to a detail data set. Entries in a master set must be explicitly added or deleted.

marker - an output primitive that is a center oriented symbol used to identify a point in the UVW coordinate system.

master data set - a data set whose entries contain one key item and zero or more detail data items. A master set is generally used as an index to one or more detail data sets. Master data sets support all data base access methods.

media - the physical material on which data is actually stored (as distinct from the device, which does the actual reading and writing). Media fall into two categories: removable and non-removeable. Typical removeable media supported by the Series 500 are:

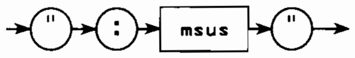
- p/n 9164-0105 8 inch 0.5Mb flexible discs
- HP 92190A 5¼ inch .25Mb flexible discs
- HP 92195A 8 inch 1.2Mb flexible discs

Typical non-removeable, or fixed media are:

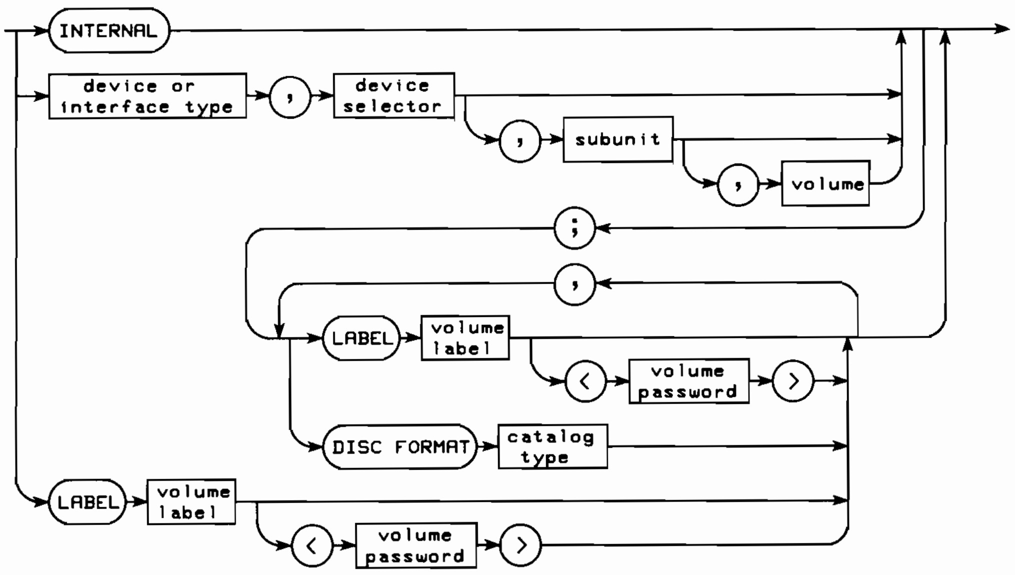
- ":MEMORY" volumes
- the disc platters of the internal HP 97093A (system option 610) Winchester disc
- the disc platters of the peripheral HP 7908/11/12 Winchester and 7933 discs

media record - a record in a data set file (type DSET) which contains one data entry. Each media record contains a header (with flag and pointer fields), and the data entry which contains fields for data item values.

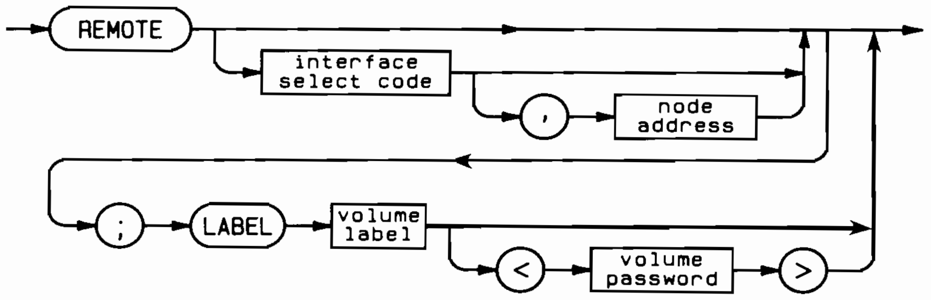
media specifier - a string expression which designates the mass storage medium affected by the statement.



literal form of **media specifier:**



literal form of local **msus**:



literal form of remote **msus:**

Item	Description/Default	Range Restrictions
msus	literal, mass storage unit specifier; Default = MASS STORAGE IS device	(see syntax diagram)
device or interface type	literal specifying the type of mass storage device or interface	CS80 HP82901 HP82902 HP8290X HP9885 HP9895 MEMORY
device selector	integer constant	(see *device selector* in Glossary)
subunit	integer constant; Default = 0	(device dependent)
volume	integer constant; Default = 0	(device dependent)
volume label	literal	(see volume label)
volume password	literal providing access to the volume (SDF only); Default = no password	(see *password* in Glossary)
catalog type	literal specifying the type of catalog organization on the medium; Default = SDF	9845 LIF SDF
interface select code	integer	(See interface select code in Glossary)
node address	integer; Default = 0	Range of Double

Example *media specifiers*:

```
    ":INTERNAL"
    ":CS80,7"
    ":MEMORY,0,27"
    ":LABEL VNAME"
    ":HP9895,705,1;LABEL VNAME<PASS1>"
    ":REMOTE"
```

Notes

INTERNAL specifies one of two possible disc drives installed in the HP 9000 Model 520. See *internal mass storage device* in this Glossary.

REMOTE specifies a remote device on the Shared Resource Management system.

The media type **MEMORY** specifies a mass storage device created from the system's random access memory. This pseudo-device is initialized like physical discs with the INITIALIZE statement. There can be up to 32 memory volumes at one time. They are addressed by the **unit numbers** 0 thru 31; the **device selector** is zero.

The **catalog type** is also referred to as the directory format.

Volume passwords are only available with the SDF directory format. When accessing a volume, the volume name's password permits access to all capabilities for all files in that volume. This is not the same as a password on the root directory. To set a password on the volume or root directory, you must use the PROTECT statement.

The **DISC FORMAT** expression of the INITIALIZE statement specifes the catalog type when the medium is initialized. When the DISC FORMAT expression is specified in any other statement, the system checks to make sure the volume specified has that directory format. If it does not, an error occurs.

memory plane - a single color plane of a plotter. All graphics consisting of that color are displayed by that plane.

A monochrome raster CRT has a frame buffer consisting of 1 bit per pixel. A color raster CRT has multiple bits per pixel, each bit corresponding to a different color (the 98770B has 3 bits per pixel), to store color information. Each set of bits in the frame buffer is called a memory plane. Thus, the 98780B has one memory plane and the 98770B has three memory planes.

memory word - a portion of a raster plotter's memory which defines the state of one pixel.

modelling transformation - the user-definable transformation which is concatenated with the viewing operation to alter the view of graphics output. It is typically used to generate a number of similar objects from a single source.

Common modelling transformations include rotations, translations and scaling.

modification counter - the number of DBPUT, DBDELETE, and DBUPDATE operations which have been performed on a data base since the counter was last reset. The counter is returned by DBINFO mode 501 as in integer in the range of a DOUBLE. Data base operations which reset the counter include: create, erase, purge of the entire data base, and backup of the data base with an HP-supplied utility.

monadic operator - an operator which performs an operation using a single succeeding expression as an argument. The monadic operators are:

−	unary minus
+	unary plus
NOT	boolean complement

msus - an abbreviation of "mass storage unit specifer". It is a literal that specifies a device to be used for mass storage operations.

See media specifier.

null string - any string expression which has an evaluated length of zero. The literal form of a null string is " ". The length of a string can be determined with the LEN function.

number builder - the algorithm within the BASIC Language System which converts strings representing numbers to numeric values. The number builder evaluates strings for three BASIC language operations:

1. Strings input to numeric variables in an INPUT statement and constants entered in program lines (except DATA).
2. Strings evaluated by the VAL function, or input to numeric variables in a READ/DATA or READ# statement.
3. Strings input to numeric variables in an ENTER statement.

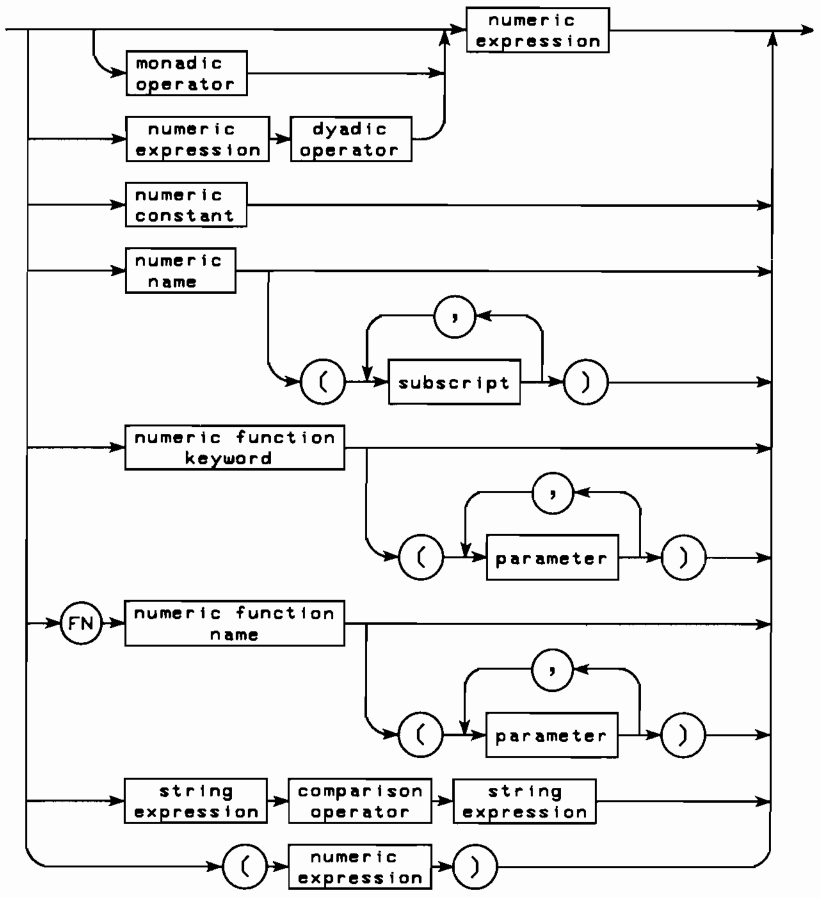
In general the number builder expects well formed numbers. The number may begin with a sign (+, −), followed by a mantissa containing at least one digit (0,1 thru 9) and zero or one decimal point (.). If more digits are received for the mantissa than a REAL data type can represent, the extra digits are included in the calculation of the magnitude of the exponent, but are otherwise ignored.

An exponent field can follow the mantissa. It must begin with a exponent letter (e,E,D,L) and is followed by at least one digit of the exponent magnitude. The magnitude field can begin with a sign and must consist of one, two or three digits. A decimal point in the exponent terminates (delimits) the entire number.

The following table summarizes the minor differences in the three types of number builder operations. *Evaluated* means that BASIC evaluates the non-numerics as program or BASIC names before they are sent to the number builder. *Delim* means that the conversion of the substring stops at that character.

	INPUT	VAL, READ	ENTER
Leading non-numeric characters	Evaluated	Error	Skipped
Extra decimal point, sign, exponent char.	Error	Delim.	Delim.
D and L in exponent	Error	OK	OK

numeric expression - an expression which evaluates to a numeric value in the range of an INTEGER, DOUBLE, SHORT or REAL.



A **numeric function keyword** is a literal naming a BASIC function which returns a numeric value, such as ABS or SIN.

A **numeric function name** is a BASIC name identifying a user-defined function which returns a numeric value

The other terms and expressions in the diagram are defined elsewhere in this Glossary.

numeric variable - a variable which is of type INTEGER, DOUBLE, REAL or SHORT. A **simple numeric variable** is a BASIC name of a scalar numeric variable. A **numeric array variable** is a BASIC name of an array variable followed by a (*) **term. A numeric array element** is a BASIC name of a numeric array followed by a one or more subscripts enclosed within parentheses.

parallel projection - a viewing projection in which the projectors are parallel. A parallel projection is the special case of a perspective projection in which the center of projection is moved infinitely far from the view reference point.

See the PROJECT statement.

parameter - see *formal parameter.*

parity - the process of setting (on output) or checking (on input) the state of one or more reserved bits in each binary data element. Memory is organized in 39-bit elements, each consisting of 32 data bits and 7 parity bits. The parity bits contain an error correcting code. No BASIC statement provides direct control of memory parity. I/O data can have parity computed for 8-bit (byte) elements. HP-IB commands are sent with odd parity. I/O path name, PRINTALL IS and PRINTER IS data can have none, zero, one, odd or even parity if a PARITY I/O attribute is specified.

In a data byte, the parity bit is the high-order bit. It is normally referred to as bit 7. If parity is not *off*, this bit is set on output and checked and removed on input. A parity error (152) occurs if the input parity check fails. If the binary value of the data requires the use of bit 7, such as extension characters, parity should be off or bit 7 may be changed on output and is always zero after input.

Parity **off** signifies that data is transferred as 8-bit bytes. Parity is neither generated nor checked. Bit 7 is treated as a data bit.

Parity **zero** signifies that the data is transferred as 7 data bits plus parity. The parity bit is zero.

Parity **one** signifies that the data is transferred as 7 data bits plus parity. The parity bit is one.

Parity **odd** signifies that the data is transferred as 7 data bits plus parity. There is an odd number (1,3,5,7) of bits on in the data byte, including the parity bit. If there are 1,3,5 or 7 data bits on, the parity bit is zero. If there are 0,2,4, or 6 bits on in the data, the parity bit is one.

Parity **even** signifies that the data is transferred as 7 data bits plus parity. There is an even number (0,2,4,6,8) of bits on in the data byte, including the parity bit. If there are 0,2,4, or 6 data bits on, the parity bit is zero. If there are 1,3,5, or 7 bits on in the data, the parity bit is one.

partition - a logical computer within the physical computer which is independent of other partitions. The number of partitions which can exist simultaneously is limited by their memory requirements. A partition can be in either the foreground or the background. See the ATTACH and CREATE PARTITION statements.

password - provides access to a protected file, volume or IMAGE data set. It is a non-listable string kept with the file or data set descripton in the file's directory or data base root file.

	Maximum number of characters	**Characters not allowed**
SDF	16	>
LIF	2	>
9845	2	>
Data set	16	; and imbedded blanks

See the PROTECT and DBOPEN statements.

path - in a data base, the links between master data set entries and the heads of chains of detail set entries which contain the same key item values.

path count - a number specified in a data base schema which specifies the number of paths a key item has from a master data set to linked detail data sets.

pen - the device on a plotter which is used to make graphics output primitives visible.

perspective projection - a viewing projection in which the center of projection is at a finite distance from the view reference point. The projectors converge to a center of projection in a perspective projection.

See the PROJECT statement.

pixel - picture element. The smallest unit of resolution on a raster CRT (a single dot).

power-up - (noun and adj.), **power up** (verb) - supplying the system with electrical power, loading and initializing the operating or language system.

pre-run - an initialization of a partition which occurs after the execution of a RUN or programed LOAD statement but before the program begins executing. A subset of these operations, ''subprogram pre-run'' is performed at subprogram entry. The checks performed and the partition attributes reset during pre-run are summarized in the Master Reset Table in the Useful Tables section.

primary address - in a *device selector*, the two digits following the *interface select code* which identify a specific device connected to that interface. The range of a primary address is 00 thru 99. For HP-IB devices, the range is 00 thru 31. Devices must be set to an address in the range 00 thru 30 because address 31 is reserved for use as the unlisten (UNL) command. HP-IB mass-storage devices are usually restricted to addresses in the range 00 thru 07. The HP-IB primary address is normally configured by switches on the device.

primary entry - in a data base, the record number produced by the calculated access method hashing algorithm is the *primary location*. If the key item value stored in that entry hashed to that location, it is the primary entry. A different key item value which also hashes to the same record is called a *synonym* and is stored in a *secondary entry*.

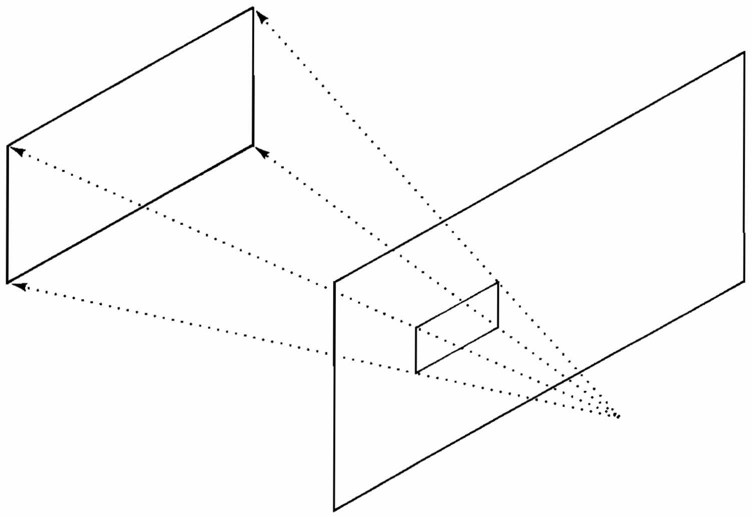
primary path - in a data base, the path represented by the first defined master data set which is linked to the first search item in the specified detail data set. If no DBINFO is performed, a chained DBGET in a detail data set uses the primary path.

PROG file - a file which contains a BASIC program in either intermediate or compiled code. A PROG file supports STORE SUB and COMPILE TO by allowing space used by a context to be re-used if the context is replaced in the file.

The number of records in the file must be at least large enough to hold the file's overhead information. The overhead is 16 bytes plus 42 bytes for each context.

See the CREATE PROG, LOAD, LOADSUB, RE-STORE, STORE, and STORESUB statements.

projector - a 3D line that passes through a point to define its projection onto a plane. The following picture shows some projectors.



pseudonym - one of zero to four alternate names for a data base data item. The list of pseudonyms for an item is returned by DBINFO mode 107.

raster - a display composed of a raster (multiple lines) of pixels (dots). Normally, the visible state of each pixel is represented by the value of one or more bits in an associated raster memory. This is also referred to as a "bit-mapped" display.

REAL - a numeric data type that is stored internally in eight bytes using IEEE 64-bit floating point binary representation. One bit is used for the number's sign, 11 bits are for a biased exponent (bias = 1023) and 52 bits for a mantissa. There is an implied 1 preceding the mantissa. The range of REAL is:

±1.797 693 134 862 315 E+308 thru ±2.225 073 858 507 202 E−308 and 0

If a variable is not explicitly declared another type, it is implicitly declared REAL.

record - a unit of data storage. For BCD, BDAT and DATA files, see *defined record.* For a data base, see *media record.*

RGB - Red, Green and Blue, a system for defining a color by its primary constituents.

register - a numeric expression corresponding to a storage location associated with an interface, device, or I/O path name. Each location may CONTROL, or contain the STATUS of, the interface, device, or I/O path name. Registers do not necessarily correspond to physical registers in an interface or device. Each interface, device, or I/O path name has one or more registers, numbered in the range 0 thru 999.

root directory - the highest level directory in a volume. All other directories and files on the device are accessed through the root directory.

root file - in a data base, a file of type ROOT which contains all of the structural information about a data base.

saturation - the attribute of color denoting its degree of departure from the achromatic color of the same brightness, that is, black, grey or white.

scalar - a simple numeric variable or single numeric array element.

schema - a description in IMAGE schema language which defines the structure of a data base. The schema is normally created in a file. The SCHEMA processor program reads the schema text file as data to create the root file.

SCREEN - a subdivision of the CRT. SCREENs are numbered from 0 thru 99. There are two types of SCREENs: PRIVATE (0 thru 50) and PUBLIC (51 thru 99). PRIVATE SCREENs can be accessed only from the partition which created them. PUBLIC SCREENs can be accessed from any partition. If a SCREEN has a system function assigned (see ASSIGN...TO SCREEN), that function can only be seen in the foreground partition.

SDF - an abbreviation for Structured Directory Format. SDF provides tree-structured access to files through the *root directory* of the volume. Unlike LIF and 9845, SDF supports file type DIR (directory). Each DIR file contains its own catalog of files, which can include additional DIR files.

search item - an item in a detail data set having associated pointers which link its entry to a chain of detail entries containing the same value for that item. The chain is linked to one or more master data sets in which that item is *key item.*

secondary addresses - in an HP-IB *device selector*, the pairs of digits following the *primary address* which identify a feature within the bus device. Up to six secondary addresses can be specified in a device selector. The range of a secondary address is 00 thru 31. Sending secondary talk or listen addresses to a device has a device-specific effect. Refer to the appropriate device documentation.

secondary entry - a master data set record occupied by a data entry which cannot be stored (DBPUT) at its primary location because that record is already occupied by an entry having a different key item value. A secondary entry is linked to its primary location by a synonym chain.

sector - the smallest unit of data written or read by mass storage device drivers. A sector is typically 256 bytes long and is sometimes referred to as a ''physical record''.

semaphore - a variable within the BASIC Language System which records the occurrence of a condition. Boolean semaphores record only a single occurrence of the condition, such as an interrupt (INTR). Counting semaphores can record more than one occurrence, such as an EOR. Each partition has one semaphore for each ON EOR, ON EOT and ON INTR condition. There is only one semaphore for each named EVENT in the entire system. See EVENT Semaphore.

serial access - the reading or writing of records in a file sequentially. In a data base, the reading (DBGET) or writing (DBUPDATE) of *occupied* entries sequentially.

SHORT - a numeric data type that is stored internally in four bytes using IEEE 32-bit floating point binary representation. One bit is used for the number's sign, 8 bits are for a biased exponent (bias = 1023) and 23 bits for a mantissa. There is an implied 1 preceding the mantissa. The range of SHORT is approximately:

$\pm 3.402\,823$ E$+38$ thru $\pm 1.175\,494$ E-38

simple variable - a single numeric or string variable which is identified by a BASIC name. A simple variable cannot be an array element or a substring, although a simple string variable can contain a substring.

single variable - a variable which identifies a single location in memory. It can be a simple numeric or string variable, a single numeric or string array element or a substring.

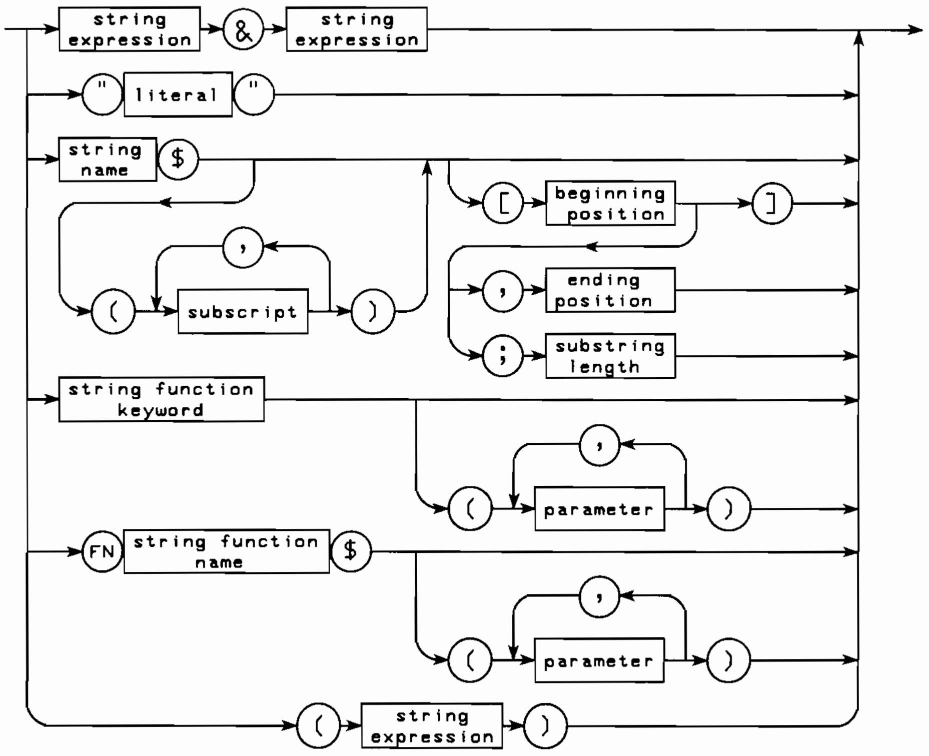
soft clip boundaries - user-definable boundaries around the current clip area. The soft clip boundaries can be around the viewport or an area defined by the CLIP statement. Graphics primitives, other than LABELs, are clipped at these boundaries.

See the CLIP and VIEWPORT statements.

sort item - a data item in a detail data set which is used by IMAGE to sort a chain in DBPUT. New entries are inserted in the current chain in ascending order according to the value of the sort item. Duplicate sort item values are inserted in chronological order.

space - the ASCII blank character represented by character code 32 (decimal). Code 32 is used for all "blank-filled" operations. Code 160 is also a blank if the output device supports the HP 8-bit Roman extension character set. Code 223 is a blank on the internal printer and display of the HP 9000 Model 520, but is not recommended. Codes 32, 160 and 223 are not lexically identical.

string expression - an expression which evaluates to a string of characters or bytes.



string name is a BASIC name of a string variable.

beginning position is a numeric expression which, when evaluated and rounded to an integer, is in the range of DOUBLE>0 and within the length of the string or substring.

ending position is a numeric expression which, when evaluated and rounded to an integer, is in the range of DOUBLE⩾0 and within the length of the string or substring.

substring length is a numeric expression which, when evaluated and rounded to an integer, is in the range of DOUBLE⩾0 and less than or equal to the remaining length of the string or substring.

string function keyword is a literal naming a BASIC function which returns a string value.

string function name is a BASIC name identifying a user-defined function which returns a string value.

string variable - a variable of type string. A **simple string variable** is a BASIC name followed by a $. A **string array variable** is a BASIC name followed by a **$(*)** term. A **string array element** is a string array name followed by a one or more subscripts enclosed within parentheses. A **substring** is a simple string variable or string array element followed by substring specifiers enclosed within brackets. See *substring.*

subprogram - the smallest portion of a program that can be separately compiled. SUB statements and multi-line DEF FN statements establish boundaries between subprograms. Variables and line labels with a given name in one subprogram have no relationship to variables and line labels with that name in other subprograms. Control can only be passed from outside a subprogram to the first line of that subprogram and can only return to the next statement in the calling program after the CALL or FN. Statements such as GOTO and GOSUB may only reference lines defined within their subprogram. Subprograms may communicate with the main program and each other via formal parameters, COM variables, files or EVENTs. Each invocation of a subprogram is a separate context.

The MAIN program is a special case of subprogram since it can be separately compiled, but does not have parameters and does not begin with SUB or DEF FN.

subscript - a numeric value that specifies a position in a dimension of an array. An element of the array is uniquely identified if a subscript is specified for each dimension of the array.

substring - a contiguous series of characters that comprises all or part of a string. Substrings are accessed by specifying a *beginning position*, or a beginning position and an *ending position*, or a beginning position and a maximum *substring length*. See *string expression*.

The beginning position must be at least one and no greater than the current string length plus one. When only the beginning position is specified, the substring includes all characters from that position to the current end of the string.

The ending position must be no less than the beginning position minus one and no greater than the dimensioned length of the string. When both beginning and ending positions are specified, the substring includes all characters from the beginning position to the ending position or current end of the string, whichever is less.

The maximum substring length must be at least zero and no greater than one plus the dimensioned length of the string minus the beginning position. When a beginning position and substring length are specified, the substring starts at the beginning position and includes the number of characters specified by the substring length. If there are not enough characters available, the substring includes only the characters from the beginning position to the current end of the string.

synonym - a key item value which hashes to a record already occupied by a different key item value. A new synonym is stored in the next available free record, which is then linked to the primary location by a synonym chain. This is called a secondary entry.

system controller - the HP-IB controller that can assert the REN and IFC lines. IFC takes active control of the bus. REN controls the remote/local state of all the other bus devices. Only one device is permitted to be system controller, but many devices capable of being *active controller* are allowed.

SYSTEM name - used to name EVENTs and PARTITIONs. It is a literal consisting of 0 thru 16 characters. There are no restrictions on the characters. They can be any codes from 0 thru 255. If the name is less than 16 characters long, it is considered to have trailing blanks. A null string is considered to be all blanks.

talker - an HP-IB device which is configured to send data bytes (DAB) on the bus. Only one device can be the talker on the bus at one time. A device is normally made the talker by sending a Talk Address (TA*) command specifying the device's talk address. An HP-IB interface is made a talker when it sends or receives its own talk address (MTA).

tracking - also called "echoing"; the process of indicating on a plotter the position of the locator of a graphics input device.

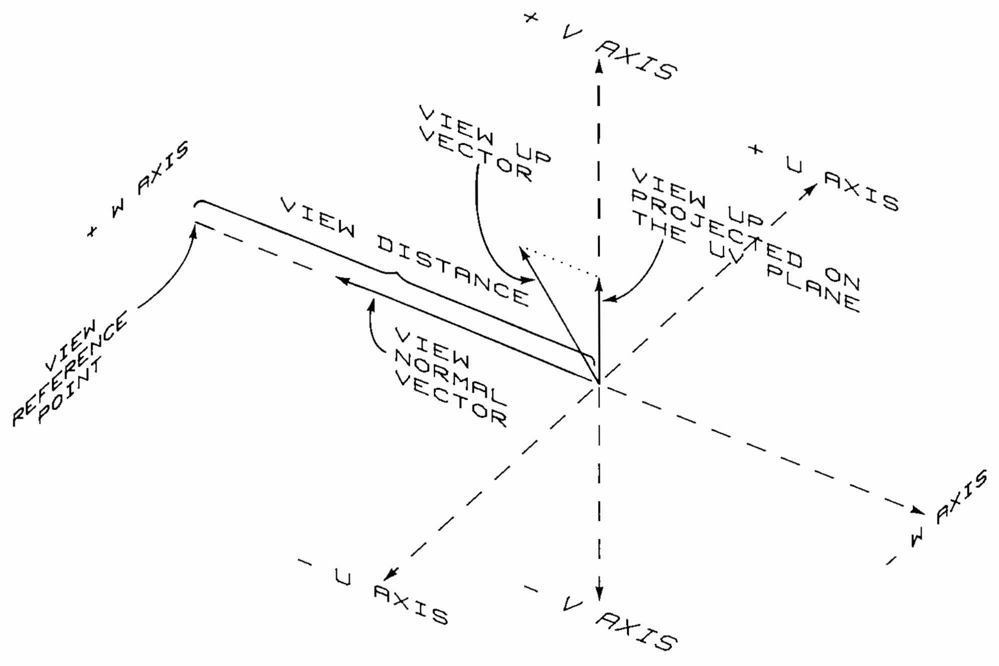
Typically, the locator position of an input device such as a light pen is tracked using a marker such as a crosshair on a CRT or the physical pen on a hard-copy plotter.

See TRACK...IS ON/OFF and SET ECHO.

UDU - See *User Defined Units.*

User Defined Units - a unit-of-measure in the XYZ coordinate system. The length of each unit along the X, Y or Z axis is determined with the MSCALE, SHOW and WINDOW statements.

UVW coordinate system - the view plane coordinate system. The W axis is parallel with the view normal vector. The V axis is determined by projecting the view up vector onto the view plane with a projection parallel to the view normal vector. The U axis is in the view plane and is perpendicular to the V axis. The origin of the UVW coordinate system is the projection of the view reference point onto the view plane using the current projection (parallel or perspective). The following picture is a model of this coordinate system.



UVW units - the unit-of-measure in the UVW coordinate system. These units are the same length as UDUs.

vector - a one dimensional array.

view distance - the distance from the view plane to the view reference point along the view normal. The view distance is always positive.

viewing operation - the process by which the graphics system interprets the coordinates of graphics output statements to create an image for display. The process is defined by CLIP, GINIT, MSCALE, PROJECT, SETGU, SETUU, SHOW, VIEW DEPTH, VIEW DISTANCE, VIEW NORMAL, VIEWPORT, VIEW POINT, VIEW UP and WINDOW.

view normal vector - a vector in the UVW coordinate system specified relative to the view reference point and is perpendicular to the view plane. The view normal orients the view plane in the UVW coordinate system. See UVW coordinate system for the view normal vector's relationship to the UVW coordinate system.

view plane - a plane in the XYZ and UVW coordinate systems onto which all graphics primitives are projected. This plane is also used to specify the window for clipping. The view plane is perpendicular to the view normal vector at the distance specified by VIEW DISTANCE from the view reference point. See UVW coordinate system for the view plane's relationship to the UVW coordinate system.

viewport - a rectangular region of the logical view surface onto which the window can be mapped.

view reference point - a point in the UVW coordinate system, typically on or near the object being viewed, that is used to position the view plane.

view surface - the plotting surface of a plotter.

volume - a complete and separate unit of mass storage which represents an independent file system. It is often a unique mountable medium such as a floppy disc. It is the largest organizational unit of mass storage on a medium.

volume label - a literal which uniquely identifies a volume.

The name consists of characters excluding the slash (/), less-than (<), colon (:) and semicolon (;). A single period (.) or double period (..) cannot be used as a label. Leading and embedded blanks are deleted. Names with a length less than the maximum for the directory format are considered to have trailing blanks.

In a Structured Directory (**SDF**), labels are 1 to 16 characters long. Note: The HP-UX file system ignores volume labels.

In a Logical Interchange Format (**LIF**) directory, labels are 1 to 6 characters long. LIF is supported by many HP computers. If strict LIF compatability is desired, you should also restrict LIF volume labels to the uppercase ASCII letters (A thru Z), the ASCII numerals (0, 1 thru 9), the underscore (_), and always begin the label with a letter.

In an HP9835/45 directory (**9845**), labels are 1 to 6 characters long. If strict compatability with the HP9835/45 is desired, you should also restrict 9845 labels to exclude the ASCII NUL (CHR$(0)), comma (,) space ( ), and the DEL (CHR$(127)) characters.

window - in **3D**, it is the portion of the view plane which contains projected images that are output. The edges of the window are specified by minimum and maximum U and V coordinates.

See the CLIP statement.

working directory - the directory from which the path of the file specifier starts unless another directory is specified. The working directory can be defined using the MASS STORAGE IS statement.

See the MASS STORAGE IS statement.

working size of array - a subset of the declared (originally dimensioned) size of an array. Storage is allocated for an array based on its declared size. Some array operations change the size of the array. This new size is its working size. It does not change the amount of storage allocated. The working size of an array may not exceed its declared size.

See the REDIM statement.

XYZ coordinate system - a coordinate system used for defining graphics output. It is a left-handed coordinate system where the X axis is horizontal, the Y axis is vertical and the Z axis travels from front to back.

The unit-of-measure can be GDUs or UDUs.

Errors

Error Messages

1	Missing OPTION or configuration error
2	Memory overflow
3	Line not found or not in current program context
4	Improper return
5	Improper context terminator
6	Improper FOR/NEXT matching
7	Undefined function or subroutine
8	Improper parameter matching
9	Improper number of parameters
10	String value required
11	Numeric value required
12	Attempt to redeclare variable
13	Array dimensions not specified
14	Multiple OPTION BASE statements or OPTION BASE after declaration
15	Invalid string or array bounds
16	Dimensions are improper or inconsistent
17	Subscript out of range
18	Substring out of range or improper string length
19	Improper value
20	INTEGER overflow
21	SHORT overflow
22	REAL overflow
23	DOUBLE overflow
24	SIN,COS,TAN argument too large for accurate evaluation
25	Magnitude of ASN or ACS argument is >1
26	Zero to non-positive power
27	Negative base to non-integral power
28	LOG or LGT of non-positive number
29	Illegal floating point number
30	Negative argument to SQR
31	Division by zero, X MOD Y, or X MODULO Y with Y=0
32	String does not represent a valid number
33	Improper argument for NUM or RPT$
34	Referenced line is not IMAGE
36	Out of DATA items
37	EDIT string longer than 160 characters
39	Multi-line function not allowed here
40	Improper COPYLINES, MOVELINES, DEL or REN
41	First line number > second
42	Attempt to replace, modify or delete a busy line or subprogram
43	Matrix not square
44	Illegal operand in matrix transpose or matrix multiply
46	No binary to STORE BIN or no program to STORE or SAVE
47	COM declarations are inconsistent or incorrect
48	Direct recursion not allowed in a single line function
50	File number <1 or >10
51	File not currently assigned
52	Improper mass storage unit specifier, bad subaddress specified, or bad driver name
53	Improper file name
54	Duplicate file name
55	Directory overflow
56	File name is undefined
57	SDF support missing
58	Improper file type
59	Physical or logical End Of File/BUFFER found
60	Physical or logical End of Record found in random mode
61	Defined record size is too small for data item
62	File is protected, wrong PROTECT code specified or PROTECT not allowed
63	Invalid record size
64	Medium overflow – out of user storage space, possibly due to fragmentation
65	Incorrect data type
66	INITIALIZE failed – excessive bad tracks, or can't spare dynamically
67	Mass storage parameter is incorrect
68	Invalid line number or line did not parse during GET
69	Format switch on drive is off
73	Incorrect device type in mass storage unit specifier
77	File open on PURGE
78	Invalid volume label
79	File open on target device
80	Door open, medium not in drive, medium changed, or printer out of paper
81	Device/Interface hardware failure
82	Device/Interface not present
83	Write protected
84	Record not found; medium possibly uninitialized
85	Mass storage medium is not initialized
86	Incorrect mass storage medium
88	Read data error
89	Checkread error

90	Mass storage system error
91	Negative length field in BDAT unformatted string
92	TYP not defined for this file type or device
100	Item in USING list is a string but the corresponding IMAGE is numeric
101	Item in USING list is numeric but the corresponding IMAGE is string
102	Numeric field specifier is too large
103	Item in USING list has no corresponding IMAGE
108	Image is too long and/or complex
117	Too many nested structured statements
128	Line exceeds maximum line length in GET
133	DELSUB of missing or busy subprogram
136	REAL underflow
137	SHORT underflow
141	Variable already ALLOCATED or not allocatable
142	Variable not ALLOCATED
143	Attempt to reference a missing OPTIONAL parameter
145	Too many COM blocks, or COM blocks are interleaved
150	Improper device specifier or select code
152	Parity error
153	Insufficient data to satisfy ENTER
155	Invalid interface register number or value
157	No ENTER terminator found within 256 characters of satisfying input
158	Improper IMAGE specifier
159	Numeric data not received for numeric item
163	Driver or interface not present
164	Illegal BYTE/WORD option
165	IMAGE specifier has count > size of variable
166	Improper TRANSFER length
167	Interface status error
168	Device timeout occurred and ON TIMEOUT branch could not be taken
170	I/O operation not allowed, or HP-IB improperly addressed
171	I/O error – illegal addressing sequence
172	I/O device or peripheral error
173	I/O operation requires active or system control of the HP-IB
174	Concurrent I/O operation not allowed on object – nested I/O
175	Unreported overlapped I/O error(s) pending
177	Undefined I/O path name
208	Volume not mounted
209	SDF directory format required
210	Bad status array
211	Improper data base specified
212	Data set not found
213	Data base directory not found
214	Data base not created
215	Operation left at least one data set corrupt
216	Maximum number of data bases already open
217	Data base definition incomplete
220	Improper or illegal use of maintenance word
221	Data set not created
222	Data base directory not created
225	Root file not compatible with current version of IMAGE
226	Corrupt root file – must purge and redefine it
227	Corrupt data base – some sets require erasure
229	Data base in use
230	Improper set list or duplicate sets in the set list
231	Improper record count specified
232	Root file cannot be purged until all data sets have been purged
233	Root file not found
234	Referenced line not a 'PACKFMT' statement
236	String buffer too short for required information
244	Device was busy and could not handle request
247	Tape runaway – no data found on medium
248	Beginning/End of tape
306	Interface card failed self test
315	Missing clock from multiplexor pod
316	Link is down, clear to send false too long
326	Register address error
327	Register value error
330	Lexical table size exceeds array size
331	Improper pointer array
332	Non-existent dimension specified
333	Pointer array contains out-of-range subscript value
334	Pointer array length does not equal the number of records in the reorder dimension
335	Pointer array is not one-dimensional
337	Substring specifier extends beyond dimensioned maximum length
338	Subscript out-of-range in key specifier
340	Mode table too long or case table indicator is improper
341	Improper mode indicator
342	Lexical table is not one dimensional or is not of type INTEGER
343	Lexical mode section pointer is out of range
344	1 for 2 replacement list is either empty or too long

345	Data type of expression in CASE does not match type of expression in SELECT
347	Improper matching of structured programming construct
353	Remote node does not respond, data link failure
401	Improper argument passed to system function or statement
402	MOVELINES could not completely delete source lines after copying them
403	Line failed to copy; program modification may be incomplete
404	Specified SCREEN does not exist
405	Attempt to delete a SCREEN with system function(s) attached
406	File specifier or BUFFER parameter not allowed as a single line function parameter
407	SAVE failed; program contains unlistable line(s)
408	Out of line numbers during LOAD or LOADSUB
409	Attempt to load a non-BASIC intermediate code context
410	STORE or STORESUB failed; all contexts reference missing options
411	Record length must be 256 for RE-SAVE and RE-STORE KEY files
413	Variables must be explicitly declared in FORCE DECLARE ON mode
415	Line too complex
416	Cannot COMPILE a TRACE or GET statement
417	SCREEN number is outside the allowed range of 1 to 99
418	Illegal screen size in CREATE SCREEN
419	Illegal screen position in CREATE or MOVE SCREEN
420	Cannot ASSIGN ROLL KEYS TO SCREEN with no scrolling buffer
421	Bad key number in stored SFK definition
422	Attempt to ASSIGN KEYBOARD to a public SCREEN
423	HP-IB EOI assertion requires data
424	Insufficient I/O bandwidth to honor request
425	Too many chained SFK definitions
426	I/O resource in use by another partition or subsystem
450	Specified volume not found
451	Volume labels do not match
452	Duplicate volume labels
453	File in use
454	Directory formats do not match
455	Possible corrupt directory
456	Directory format does not support this operation
457	Passwords not supported for this directory format
458	Unsupported directory format
459	Specified file is not a directory
460	Directory not empty
461	Duplicate passwords not allowed
462	Invalid password
465	RENAME cannot specify different volumes
466	Duplicate volume entries
467	Medium has been improperly inserted
468	Disc capacity exceeds 32 bit record address range
469	HP-IB TCT byte must be at end of ATN sequence
470	Device does not support CHECKREAD
471	Device does not support TRANSFER
472	Interface cannot be HP-IB active controller
473	Synchronous data rate could not be met to complete this operation
474	Device failed its self test or diagnostic
475	HP-IB interface too slow for this device
476	Termination mode not supported by this driver
477	Only one driver may be attached to this device
478	Media failure
479	Operation incomplete due to user programmed holdoff
480	Data operation aborted by an interface or device clear operation
481	File already locked, or unlocked in exclusive mode
482	Cannot move a directory via a RENAME
483	Shared Resource Management controller is down
484	Password not found
485	HP-IB secondary command seen
486	Write blocked due to unread inbound data
487	Request incompatible with previous requests or current state
500	Partition already exists
501	Partition not present
502	Cannot delete the foreground partition
503	Partition must be foreground to ATTACH
504	Partition(s) must be in the STOP state
505	EVENT not present
506	EVENT already exists
507	EVENT LEVEL negative – cannot delete
508	Unsupported partition type in CREATE PARTITION
509	Attempt to COMPILE an empty program

510	BUFFER parameters not allowed in a default CALL
511	Result array for INV is not SHORT or REAL
512	ON EVENT active – cannot delete
515	PROG file or PROG file directory too small
600	Attribute cannot be modified once established
602	BUFFER variable has insufficient longevity
603	Variable not declared BUFFER
604	Illegal source and destination combination for TRANSFER
605	TRANSFER must be to a BDAT file
606	TRANSFER termination delimiters not supported
607	Inconsistent attributes
608	Zero count specified
609	IVAL result too large
610	Maximum buffer size of .5 Mbytes exceeded
612	BUFFER pointer(s) in use – CONTROL/RESET not allowed
650	Illegal remote operation
700	Graphics driver specifier not recognized
701	Incompatible GRAPHICS INPUT and PLOTTER drivers
702	Internal CRT has no graphics hardware, or malfunctioning hardware, or no graphics driver
704	Upper bound <= lower bound
705	Specified area is outside GDU limits
706	Incompatible hardware and driver
707	Graphics device limits out of range
708	Graphics device not initialized
709	Graphics device not selected
710	Service request interrupt on HP-IB from unknown origin
711	Maximum number of graphics devices already initialized
712	Memory overflow while attempting to report an overlapped I/O error
713	Request not supported by the device or driver
714	An attribute value (PEN or LINE TYPE) is out of range for a graphics device
730	Array not in COM for array PLOTTER
731	The specified TRACK operation has not been initiated
733	GESCAPE opcode not recognized
734	FONT identifier out of range or not found
735	FONT identifier already in use
736	FONT specification data invalid
737	FONT stroke data invalid
740	Parameters specify the zero length vector
741	Perspective image for the point is undefined – zero distance from center of projection along the view normal
742	Front plane is not between the center of projection and the back plane
743	Parameters specify a point that is on the wrong side of the view plane as determined by the view normal
744	Viewing matrix is not invertible
745	Two vectors specifying the viewing transformation are parallel
746	GDU z value too large – no point has a perspective image with that value
749	Graphics system error

IMAGE status errors

0	Successful execution – no error
−1	Miscellaneous host system error, see status element 10 for error number
−10	Maximum number of data bases already open
−11	Bad data base reference or preceding blanks missing
−12	Not all necessary data sets are locked
−14	DBPUT, DBDELETE, and DBUPDATE not allowed with DBOPEN mode 8
−21	Improper or nonexistent data set, data item, password, or volume specified
−22	Detail data set required
−23	Write access to data set required
−24	DBPUT, DBDELETE, or DBUPDATE not allowed on an Automatic master data set
−31	Improper mode specified
−32	DBOPEN mode conflicts with another user
−52	Item specified is not a key item in the specified set or bad List parameter
−91	Root file not compatible with current version of IMAGE
−92	Data base not created
−93	Corrupt root file, must purge and redefine it
−94	Corrupt data base, some sets require erasure
−135	Wait lock not allowed while a lock is already in effect
10	Beginning of file encountered
11	End of file encountered
12	Negative record number specified
13	Record number greater than capacity specified

14	Beginning of chain encountered
15	End of chain encountered
16	The data set is full
17	No current record, no chain head, or the selected record is empty
18	Broken chain encountered
20	At least one requested data set is already locked
41	DBUPDATE will not alter a key or sort item
43	Key value already exists in Master set
44	Can't delete a Master entry with non-empty Detail chains
50	String buffer is too small for requested data
51	Variable size or type does not match the item size or type
52	Number of variables specified does not match the item list
53	Argument parameter type or size incompatible with key item type or size
94	Corrupt data base opened in read-only mode
1xx	There is no chain head for path xx
3xx	The Automatic Master for path xx is full

System Loader Messages

The **system loader** is a program (which permanently resides in the computer) that causes the computer to search for and load an operating system. If the computer is unable to locate and load the BASIC System, (or any operating system), a message is displayed. Each of these messages is explained below. Possible causes for many of the messages are provided. If the message begins with `ERROR:`, the system halts after issuing the message. If the message begins with `NOTE:`, the message provides information and the computer continues operating.

If the message you receive indicates a hardware failure, run the Module Self Test or the System Integrity Test before calling your HP Customer Engineer for service.

Often, the computer attempts to identify the device to which it was "talking" when the message was generated. A value (represented as NNNNN in the messages below) is displayed with the message. You may identify the device with the following procedure.

1. Divide NNNNN by 8; drop the fractional portion to yield an integer result. This value identifies the I/O Processor (IOP) which controls the device to which the computer was "talking" when the message was generated: 2 identifies the first IOP, 3 identifies the second IOP, and 4 identifies the third IOP.
2. Find the I/O card cage controlled by the IOP identified in step 1 above. The first IOP always controls the I/O card cage located on the right side of the computer. The second and third IOPs controll I/O expanders.
3. Find the result of NNNNN modulo 8. This value specifies a slot number of I/O card cage identified in step 2 above. Find the interface card occupying this slot. (The top slot in the computer's built-in I/O card cage is slot 2; the first slot in an I/O expander is slot 0.) The computer was talking to this interface card or to a device connected via this interface card when the message was generated.

For example, if NNNNN is 29 then the second IOP is the controlling I/O processor (since the integer portion of 29/8 is 3 and since 3 identifies the second IOP). The slot number of the I/O card cage is 5 (since 29/8 is 3 with remainder 5; the result of a modulo operation is defined to be the integer remainder of the division).

Messages

`Loader XXX` - informational message identifying the revision of the system loader. This message is usually followed by a single line message identifying the operating system the computer is attempting to load.

`Please mount next volume.` - informational message. The loader is ready to load another portion of the operating system. Mount the volume containing an unloaded portion of the operating system. Volumes may be mounted in any order without affecting the loading process.

`SYSTEM NOT FOUND; WILL RETRY: XXX` - unable to find an operating system on any mass storage device. The loader will attempt to find an operating system again in XXX seconds. Possible causes: mass storage device not powered up, no media in mass storage device, wrong disc in disc drive, computer or mass storage device hardware failure, media failure, incompatible loader/system revision numbers, etc.

`BAD SYSTEM FILE: NNNNN` - operating system loaded; however, an error was detected in the operating system code while loading. Possible causes: corrupt system, media failure, mass storage hardware failure or computer hardware failure.

`INSUFFICIENT USABLE MEMORY: XXXX` - computer has only `XXXX` bytes of usable memory available. The amount of usable memory is too small to contain the operating system. The amount of memory required to load an operating system is: 32 kbytes + operating system size (rounded up to a multiple of 16 kbytes). Possible causes: corrupt system or hardware (memory) failure.

`BAD CARD OR DEVICE: NNNNN` - informational message. A hardware failure has been detected (interface card or mass storage device did not pass Module Self Test). The loader continues searching for an operating system.

`DEVICE NOT READY: NNNNN` - while loading, the device containing the operating system went "off line".

`VOLUME NOT MOUNTED: NNNNN` - while loading, the volume containing the operating system was removed from its mass storage device.

`DMA FAILED: NNNNN` - data did not transfer properly from the mass storage device to the computer. Possible causes: mass storage device hardware failure or computer hardware failure.

`UNRECOVERABLE DATA: NNNNN` - part of the operating system is not readable. Possible causes: media failure or mass storage hardware failure.

`END OF VOLUME: NNNNN` - attempt to address or read past the end of a volume. Possible causes: corrupt system, media failure or mass storage device hardware failure.

`CTRLR/UNIT FAULT: NNNNN` - hardware passed initial self test; however, it failed while being used to load operating system. Possible causes: computer (interface card) hardware failure or mass storage device hardware failure.

`IO TIMEOUT: NNNNN` - mass storage device failed to respond fast enough while attempting to load from it. Possible causes: computer hardware failure or mass storage device hardware failure.

`CS80 DEVICE: NNNNN` - indicates a mass storage device hardware failure.

`KBD/SCM NOT FOUND.` - indicates a computer hardware failure (keyboard) on a Model 520 Computer; computer hardware failure (system control module) on a Model 530 or Model 540 Computer.

`BAD IO BUS: NNNNN` - indicates a computer hardware failure on the computer's first I/O Processor.

`BAD NVM1: NNNNN` - indicates that the 64 byte NVM (Non-Volatile Memory) failed self test. Possible causes: computer hardware error.

`BAD NVM2: NNNNN` - indicates that the 2 kbyte NVM (Non-Volatile Memory) failed self test (applicable only to HP 9000 Model 530 and Model 540 Computers). Possible causes: computer hardware error.

`BAD RTC: NNNNN` - indicates that the computer's built-in real time clock is not functioning.

`BAD SP: NNNNN` - indicates that the Model 530 or Model 540 Computer's service processor failed self test.

`HPIB CARD: NNNNN` - indicates a failure of an HP-IB interface. Try loading the system again. If the same failure occurs, call your HP Customer Engineer.

Missing Option Message

You can GET a file containing a program which includes segments from options even if those options are currently not loaded in your system. When you LIST the program, the message

```
    *MISSING OPTION nn
```

appears at each line where the statement belongs. nn is the option number. The numbers signify an option as shown in the following table.

Option	Option Number
XREF	8
IO	16
GRAPHICS	48
IMAGE_DBM	64
MS	128

Subject Index

Manual Comment Sheet Instruction

If you have any comments or questions regarding this manual, write them on the enclosed comment sheets and place them in the mail. Include page numbers with your comments wherever possible.

If there is a revision number, (found on the Printing History page), include it on the comment sheet. Also include a return address so that we can respond as soon as possible.

The sheets are designed to be folded into thirds along the dotted lines and taped closed. Do not use staples.

Thank you for your time and interest.