BASIC 5.0/5.1 Programming Techniques

Vol. 1: General Topics

HP 9000 Series 200/300 Computers

HP Part Number 98613-90012

HEWLETT
PACKARD

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

NOTICE

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

HEWLETT-PACKARD MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS MANUAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Hewlett-Packard shall not be liable for errors contained herein or direct, indirect, special, incidental or consequential damages in connection with the furnishing, performance, or use of this material.

WARRANTY

A copy of the specific warranty terms applicable to your Hewlett-Packard product and replacement parts can be obtained from your local Sales and Service Office.

Restricted Rights Legend

Use, duplication or disclosure by the U.S. Government Department of Defense is subject to restrictions as set forth in paragraph (b)(3)(ii) of the Rights in Technical Data and Software clause in FAR 52.227-7013.

Use of this manual and flexible disc(s) or tape cartridge(s) supplied for this pack is restricted to this product only. Additional copies of the programs can be made for security and back-up purposes only. Resale of the programs in their present form or with alterations, is expressly prohibited.

This software and documentation is based in part on the Fourth Berkeley Software Distribution under license from the Regents of the University of California

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.

January 1987...Edition 1

November 1987...Edition 2. This edition reflects 5.0 corrections and 5.1 additions.

Table of Contents

Chapter 1: Manual Organization

Chapter 2: Program Structure and Flow

Chapter 3: Numeric Computation

Chapter 4: Numeric Arrays

Chapter 6: Subprograms and User-Defined Functions

Chapter 7: Data Storage and Retrieval

Chapter 8: Using a Printer

Chapter 9: The Real-Time Clock

Chapter 10: Communicating with the Operator

Chapter 11: Handling Errors

Chapter 12: Debugging Programs

Chapter 13: Efficient Use of the Computer's Resources

Chapter 14: Porting to 3.0

Chapter 15: Porting to Series 300 and 4.0

Chapter 16: Porting to 5.0

Chapter 17: Porting and Sharing Files

Chapter 18: 5.1 Enhancements

Index

Manual Organization 1

Manual Organization 1

Welcome

This manual is intended to introduce you to the Series 200/300 BASIC programming language and to provide some helpful hints on getting the most utility from it. Although this manual assumes that you have had some previous programming experience, you need not have a high skill level, nor does your previous experience need to be in BASIC. If you have never programmed a computer before, it will probably be more comfortable for you to start with one of the many beginner's text books available from various publishing companies. However, some beginners may find that they are able to start in this manual by concentrating on the fundamentals presented in the first few chapters. If you are a programming expert or are already familiar with the BASIC language of other HP computers, you may start faster by going directly to the *BASIC Language Reference* and checking the keywords you normally use. You can always come back to this manual when you have extra time to explore the computer's capabilities, or if you bump into an unfamiliar concept.

After reading each section and trying the examples shown, try your own examples. Experiment. You cannot damage the computer by pressing the wrong keys. The worst thing that can happen is that an error message will appear. All errors are listed in the "Error Messages" appendix of the *BASIC Language Reference.*

What's In This Manual?

No matter what your skill level, it is helpful to understand the contents and organization of this manual. First of all, there are some things that it is **not**. Because it is organized by topics and concepts, it is not a good place to find an individual keyword in a hurry. Keywords can be found using the index, but even so, they are often imbedded in discussions, contained in more than one place, or only partially explained. Also, this is not a good place to find complete syntactical details. Program statements are often presented only in the form that applies to the specific concept being discussed, even though there may be other forms of the statement that accomplish different purposes. If you want to quickly find the complete formal syntax of a keyword, use the *BASIC Language Reference.* It is specifically intended for that purpose.

This manual contains explanations and programming hints organized topically. A program performs various sub-tasks as it completes its overall job. Many of these tasks should be viewed separately to be understood more easily and used more effectively. For example, perhaps you have experience in another programming language. You know exactly what a "loop" does, but you didn't find the statement you were looking for in the *BASIC Language Reference.* In the chapter on "Program Structure and Flow," there is a section called "Repetition" which explains the kinds of loops available and all the statements needed to create them. The following is an overview of the chapters in this manual.

Programming Techniques Volume 1

Chapter 1: Manual Organization

Chapter 2: Program Structure and Flow

This chapter tells how the computer finds its way around your program and offers ideas on getting it to follow the proper path efficiently.

Chapter 3: Numeric Computation

This chapter covers mathematical operations and the use of numeric variables. It includes discussions on types of variables, expression evaluation, arrays, and methods of managing data memory.

Chapter 4: Numeric Arrays

This chapter covers numeric array operations.

Chapter 5: String Manipulation

Although string data can be used for any purpose the programmer desires, it is most often used for the processing of characters, words, and text. Since words are more pleasant than numbers to humans, skillful use of strings can make the input and output of programs much more natural to those using the programs. This chapter explains the programming tools available for processing string data.

Chapter 6: Subprograms and User-Defined Functions

An outstanding feature of this language is its ability to change program contexts and the speed with which it can do so. Alternate contexts (or environments) are available as user-defined functions or subprograms. These are discussed in this chapter.

Chapter 7: Data Storage and Retrieval

This chapter shows many of the alternatives available for storing the data that is intended as program input or created as program output. Topics range from convenient ways to define constants within a program to a discussion of file I/O and the computer's unified mass storage system.

Chapter 8: Using a Printer

This chapter tells how to use an external printer. Also covered are the formatting techniques (useful on both printer and CRT) to create organized, highly-readable printouts.

Chapter 9: Using the Real-Time Clock

An accurate real-time clock is available with timing resolution to the hundredth of a second and a range of years. Its capabilities are covered in this chapter.

Chapter 10: Communicating with the Operator

It is very frustrating for operator and programmer alike when the operator cannot figure out what is expected next, or when the program crashes every time a wrong key is pressed. This chapter presents some programming techniques that help ease the interaction between the computer and a human operator.

Chapter 11: Handling Errors

This chapter discusses techniques for intercepting (or trapping) errors that might occur while a program is running. Many errors can be dealt with easily by a programmer. Error trapping keeps the program running and provides valuable assistance to the computer operator.

Chapter 12: Debugging Programs

We all wish that every program would run perfectly the first time and every time. Unfortunately, there is little evidence in real life to support that fantasy. The next best thing is to have debugging tools. This chapter explains the powerful and convenient debugging features available on the computer.

Chapter 13: Efficient Use of the Computer's Resources

Which takes longer, calculating a square root or raising a number to the .5 power? Does a program run faster if the variable names are shorter? If you have a time-critical or memory-critical application, you will be interested in these answers and others provided in this chapter.

Programming Techniques Volume 2

Chapter 14: Porting to 3.0

This chapter helps the user who is porting programs from previous versions of BASIC to the 3.0 system. It discusses changes and enhancements.

Chapter 15: Porting to Series 300 and 4.0

This chapter describes Series 300 computer hardware from the standpoint of how it is different from Series 200 hardware. Then, it presents the methods of porting existing Series 200 software to Series 300 computers. It also describes the new software features of the BASIC 4.0 system (which are few, except that it supports the Series 300 computers).

Chapter 16: Porting to 5.0

This chapter lists the new features available with the 5.0 revision of the BASIC system. It also describes considerations you may need to make in porting 4.0 programs to the 5.0 system.

Chapter 17: Porting and Sharing Files

This chapter describes the file types available with BASIC systems. It discusses how to transport files to and from, and share files with, the Series 200/300 Pascal and HP-UX systems. It will be especially useful if you are sharing files between BASIC, Pascal, and HP-UX via the Hierarchical File System (HFS) disc format.

Chapter 18: 5.1 Enhancements

This chapter consist of BASIC 5.1 functionality additions and manual enhancements.

What's Not in this Manual

This is a manual of programming techniques, helpful hints, and explanations of capabilities. It is not a rigorous derivation of the BASIC language. Any statements appropriate to the topic being discussed are included in each chapter, whether they have been previously introduced or not. Since most users will not read this manual from cover to cover anyway, the approach chosen should not present any significant problems. In those cases when you have difficulty getting the meaning of certain items from context, consult the Index to find additional information.

Program Structure and Flow 2

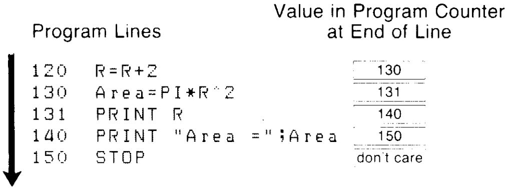
Program Structure and Flow 2

Two of the most significant characteristics of a computer are its ability to perform computations and its ability to make decisions. If the execution sequence could never be changed within a program, the computer could do little more than plug numbers into a formula. Computers have powerful computational features, but the heart of a computer's intelligence is its ability to make decisions.

The computational power of your computer is exercised as it evaluates the expressions contained in the program lines. The "Numeric Computation" and "String Manipulation" chapters present the various data manipulation tools available. This decision-making power is used to determine the order in which program lines will be executed. This chapter discusses the ways that decisions are used in controlling the "flow" of program execution.

The Program Counter

The key to the concept of decision making in a computer is an understanding of the program counter. The **program counter** is the part of the computer's internal system that tells it which line to execute. Unless otherwise specified, the program counter automatically updates at the end of each line so that it points to the next program line. This is illustrated in the following drawing.



```
120   R=R+2                      130
130   Area=PI*R^2                131
131   PRINT R                    140
140   PRINT "Area =";Area        150
150   STOP                       don't care
```

Figure 2-1. Program Counter Changes with Linear Program Flow

This fundamental type of program flow is called "linear flow." As shown by the arrow, you can visualize the flow of statement execution as being a straight line through the program listing. Although linear flow seems very elementary, always remember that this is the computer's normal mode of operation. Even experienced programmers are sometimes embarrassed to discover that a "bug" in their program was caused by the simple incrementing of the program counter into the wrong portion of the program.

As stated in the introduction of this chapter, a computer would be little more than a glorified adding machine if it were limited to linear flow. There are three general categories of program flow. These are:

- Sequence
- Selection (conditional execution)
- Repetition

In addition to capabilities in all three of these categories, your computer also has a powerful special case of selection, called **event-initiated branching**. The rest of this chapter shows how to use all of these types of program flow and gives suggestions for choosing the type of flow that is best for your application.

Sequence

This section describes the types of sequences of program execution:

- Linear flow—the BASIC system executes lines in sequential fashion.
- Halting program execution—stopping the flow of a program.
- Branching—the BASIC program redirects the normally sequential flow.

Linear Flow

The simplest form of sequence is linear flow. The preceding section showed an example of this type of flow. Although linear flow is not at all 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. Although this form of flow requires little explanation, keep these characteristics in mind:

- Linear flow involves **no** decision making. Unless there is an error condition, the program lines involved in this type of flow will always be executed in exactly the same order, regardless of the results of, or arguments to, any expression.
- Linear flow is the default mode of program execution. Unless your include a statement that stops or alters program flow, the computer will always "fall through" to the next higher-numbered line after finishing the line it is on.

Halting Program Execution

One of the obvious alternatives to executing the next line in sequence is not to execute anything. There are three statements that can be used to block the execution of the next line and halt program flow. Each of these statements has a specific purpose, as explained in the following paragraphs.

A main program is a list of program lines with an END statement on the last line. Marking the end of the main program is the primary purpose of the END statement. Therefore, a program can contain only one END statement. The secondary purpose of the END statement is stopping program execution. When an END statement is executed, program flow stops and the program moves into the stopped (non-continuable) state.

It is often necessary to stop the program flow at some point other than the end of the main program. This is the purpose of the STOP statement. A program can contain any number of STOP statements in any program context. When a STOP statement is executed, program flow stops and the program moves into the stopped (non-continuable) state. Also, if the STOP statement is executed in a subprogram context, the main program context is restored. (Subprograms and context switching are explained in the "User-Defined Functions and Subprograms" chapter.)

As an example of the use of STOP and END, consider the following program.

```
100  Radius=5
110  Circum=PI*2*Radius
120  PRINT INT(Circum)
130  STOP
140  Area=PI*Radius^2
150  PRINT INT(Area)
160  END
```

When the RUN key (f3 in the System menu of ITF keyboards) is pressed, the computer prints `31` on the CRT and the Run Indicator (lower right corner of CRT) goes off. This first press of the RUN key caused linear execution of lines 100 thru 130, with line 130 stopping that execution. If the RUN key is pressed again, the same thing will happen; the program does **not** resume execution from its stopping point in response to a RUN command. However, RUN can specify a starting point. So, execute `RUN 140`. The computer prints `0` and stops. This command caused linear execution of lines 140 thru 160, with line 160 stopping that execution. However, a RUN command also causes a prerun initialization which zeroed the value of the variable `Radius`.

You could try pressing CONTINUE or CONT (f2 in the System menu of ITF keyboards) in the preceding example, but you will get an error. A stopped program is not continuable. This leads up to the third statement for halting program flow. Replace the STOP statement on line 130 with a PAUSE statement, yielding the following program.

```
100  Radius=5
110  Circum=PI*2*Radius
120  PRINT INT(Circum)
130  PAUSE
140  Area=PI*Radius^2
150  PRINT INT(Area)
160  END
```

Now when the program is run, and the computer prints 31 on the CRT. Then when CONTINUE is pressed, the computer prints 78 on the CRT. The purpose of the PAUSE statement is to **temporarily** halt program execution, leaving the program counter intact and the program in a continuable state. One common use for the PAUSE statement is in program troubleshooting and debugging. This is covered in the "Program Debugging" chapter. Another use for PAUSE is to allow time for the computer user to read messages or follow instructions. Interfacing with a human is covered in greater depth in the "Communicating with the Operator" chapter, but here is one example of using the PAUSE statement in this way.

```
100  PRINT "This program generates a cross-reference"
110  PRINT "printout. The file to be cross-referenced"
120  PRINT "must be an ASCII file containing a BASIC"
130  PRINT "program."
140  PRINT
150  PRINT "Insert the disc with your files on it and"
160  PRINT "press CONTINUE."
170  PAUSE
180  !     Program execution resumes here after CONTINUE
```

Lines 100 thru 160 are instructions to the program user. Since a user will often just load a program and run it, the programmer cannot assume that the user's disc is in place at the start of the program. The instructions on the CRT remind the user of the program's purpose and review the initial actions needed. The PAUSE statement on line 170 gives the user all the time he needs to read the instructions, remove the program disc, and insert the "data disc." It would be ridiculous to use a WAIT statement to try to anticipate the number of seconds required for these actions. The PAUSE statement gives freedom to the user to take as little or as much time as necessary.

When CONTINUE (f2) is pressed, the program resumes with any necessary input of file names and assignments. Questions such as "Have you inserted the proper disc?" are unnecessary now. The user has already indicated compliance with the instructions by pressing CONTINUE.

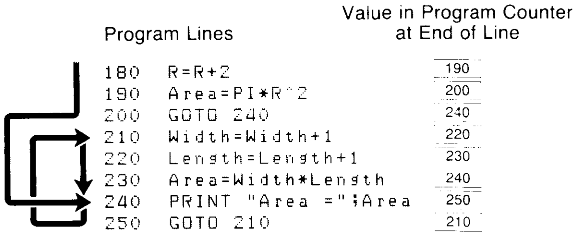
Simple Branching

An alternative to linear flow is branching. Although conditional branching is one of the building blocks for selection structures, the unconditional branch is simply a redirection of sequential flow. The keywords which provide unconditional branching are GOTO, GOSUB, CALL, and FN. The CALL and FN keywords invoke new contexts, in addition to their branching action. This is a complex action that is the topic of the "Subprograms and User-Defined Functions" chapter. This section discusses the use of GOSUB and GOTO.

Using GOTO

First, you should be aware that the structuring capabilities available in BASIC make it possible to avoid the use of the unconditional GOTO in most applications. You should also be aware that this is a highly-desirable goal. The problem is not anything inherent in the GOTO statement. The problem lies in the programmer's tendency to "glue together" pieces of an algorithm, using more and more GOTOs with each revision. Then comes that inevitable day when a fatal bug reveals that it is impossible to "GET BACK FROM" the last "GO TO." The excessive use of GOTO has been appropriately named **spaghetti coding**. Keep this very descriptive term in mind when you are deciding whether to "just throw something together" or "do it right the first time." (See the section on "Top-Down Design" in the "User-Defined Functions and Subprograms" chapter.)

The only difference between linear flow and a GOTO is that the GOTO loads the program counter with a value that is (usually) different from the next-higher line number. The GOTO statement can specify either the line number or the line label of the destination. The following drawing shows the program flow and contents of the program counter in a program segment containing a GOTO.



```
180   R=R+2                         190
190   Area=PI*R^2                   200
200   GOTO 240                      240
210   Width=Width+1                 220
220   Length=Length+1               230
230   Area=Width*Length             240
240   PRINT "Area =";Area           250
250   GOTO 210                      210
```

Figure 2-2. Program Counter Changes with GOTO Statements

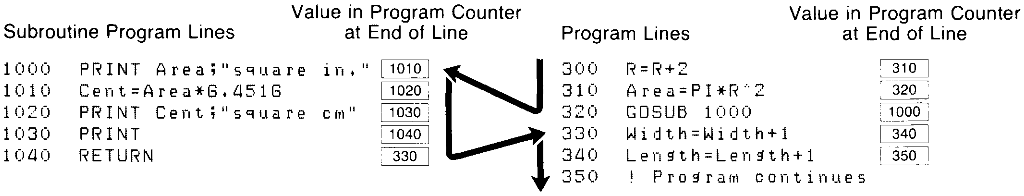
As you can see, the execution is still sequential and no decision-making is involved. The first GOTO (line 200) produces a forward jump, and the second GOTO (line 250) produces a backward jump. A forward jump is used to skip over a section of the program. An unconditional backward jump can produce an **infinite loop**. This is the endless repetition of a section of the program. In this example, the infinite loop is line 210 thru 250.

An infinite loop by itself is not usually a desirable program structure. However, it does have its place when mixed with conditional branching or event-initiated branching. Examples of these structures are given later in this chapter.

Using GOSUB

The GOSUB statement is used to transfer program execution to a subroutine. Note that a subroutine and a subprogram are very different in HP BASIC. Calling a **subprogram** invokes a new context. 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 and no local variables. If you are a newcomer to HP's BASIC, be careful to distinguish between these two terms. They have been used differently in some other programming languages.

The GOSUB is very useful in structuring and controlling programs. The similarity it has to a procedure call is that program flow can automatically return to the proper line when the subroutine is finished. The GOSUB statement can specify either the line label or the line number of the desired subroutine entry point. The following drawing shows the program flow and contents of the program counter in a program segment containing a GOSUB.


Subroutine Program Lines
Value in Program Counter at End of Line
Program Lines
Value in Program Counter at End of Line

```
1000  PRINT Area;"square in."   1010        300  R=R+2                 310
1010  Cent=Area*6.4516          1020        310  Area=PI*R^2           320
1020  PRINT Cent;"square cm"    1030        320  GOSUB 1000           1000
1030  PRINT                     1040        330  Width=Width+1         340
1040  RETURN                     330        340  Length=Length+1       350
                                            350  ! Program continues
```

Figure 2-3. Program Counter Changes with GOSUB Statement

Program execution is sequential and no decision-making is involved. The main reason that a GOSUB is a more desirable action than a GOTO is the effect of the RETURN statement. The RETURN statement always returns program execution to the line that would have been executed if the GOSUB had not occurred. This is especially useful when using an event-initiated GOSUB. Since it is usually impossible to predict when a user might press a softkey (for example), it is usually impossible to predict which program line should be returned to at the end of a service routine. By using GOSUB and RETURN, the computer does the work for you.

Another common advantage gained from the use of GOSUB is program economy resulting from the consolidation of common tasks. For example, assume that you are writing a page formatter program to neatly print letters, reports, etc. The actions taken at the end of each page might be such things as:

1. Skip two blank lines
2. Print the page number
3. Update the page counter
4. Print a form-feed
5. Zero the line counter

These end-of-page actions might be necessary at many places in the program. For example: in the new-page segment, in the conditional-page algorithm, in the normal line-printing segment, and in the end-of-file process. It would be wasteful duplication to repeat all those end-of-page steps every place they are needed.

That kind of duplication also opens the door to updating problems. Suppose that you wanted to modify the end-of-page action to make it print line-feeds instead of a form-feed for the benefit of a printer that doesn't use form-feeds. If you had duplicated the end-of-page routine in five different places in the program (or was that six?), you will be doing five times as much typing to make the change, and you will probably miss a spot.

The solution is a subroutine. For the sake of completeness in this example, the hypothetical end-of-page subroutine is shown below.

```
540 End_page:  !
550   PRINT USING "2/,K";Pagenumber
560   Pagenumber=Pagenumber+1
570   PRINT CHR$(12);
580   Lines=0
590   RETURN
```

There are no "rules" to say when a program action should be made into a subroutine and when it should be left in linear flow. The following suggestions may help you decide.

- There is no significant speed penalty for using a subroutine. The time required to process the GOSUB and RETURN is extremely small. If you are having trouble getting your application to run fast enough, it is doubtful that your problems will be solved by removing a couple of GOSUBs. In fact, the resulting loss of "readability" may actually make it more difficult to identify and correct the real problem in timing.
- The "cross-over point" in line overhead is a subroutine that is only three lines long and is called from only two places in the program. In other words, it takes the same number of program lines to duplicate three lines as it does to stick a RETURN on the end of them and add two GOSUB statements. However, there is nothing "magical" about this observation. It does not mean that you shouldn't have a subroutine shorter than three lines, or that you should go around making a subroutine out of every three-line sequence you see repeated. It should simply make you aware of possible improvements that could be made if you see the same sequence repeated in several places in your program.
- Decisions about subroutines are best made on a conceptual level. Although there is nothing wrong with accidentally discovering that you repeated ten lines which would make a good subroutine, it is better to identify the appropriateness of subroutines during planning. One question to ask yourself is, "Does it make sense to handle this task in a subroutine?" If it takes a dozen flags and status variables to select all the variations that are needed from one call to the next, a subprogram is probably a cleaner solution. Lines of code that "just happen" to be repeated in several places are not good candidates for a subroutine. A subroutine should have some identifiable task, like opening a file, normalizing a variable, processing an end-of-page, decoding a key press, parsing a string, and so forth.

Selection

The heart of a computer's decision-making power is the category of program flow called **selection**, or **conditional execution**. As the name implies, a certain segment of the program either is or is not executed according to the results of a test or condition. This is the basic action which gives the computer an appearance of possessing intelligence. Actually, it is the intelligence of the programmer which is remembered by the program and reflected in the pattern of conditional execution.

Consider a chemistry lab application as an example. There would be little use for a computer whose only function was to turn on a valve when a technician pressed the "START" button. The technician might just as well turn the valve himself. However, if the computer turned on a valve when the "START" was pressed and turned off the valve when a specified pH level occurred, then it is performing a much more useful task. If the example is extended to include state-of-the-art remote-control valves and electronic pH measuring devices, the computer is now significantly out-performing the technician. In this example, (in spite of any fancy instrumentation) the quality that moved the computer from "useless" to "useful" was its ability to *decide* when to turn off the valve. It was the programmer (you) who actually specified the criteria for the decision. Those criteria were then communicated to the computer using conditional-execution program structures. As a result, the computer was able to repeat the programmer's intention with much greater speed and accuracy than a human.

This section presents the conditional-execution statements according to various applications. The following is a summary of these groupings.

1. Conditional execution of one segment.
2. Conditionally choosing one of two segments.
3. Conditionally choosing one of many segments.

Conditional Execution of One Segment

The basic decision to execute or not execute a program segment is made by the IF...THEN statement. This statement includes a numeric 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. Although the expression contained in an IF...THEN is treated as a Boolean expression, note that there is no "BOOLEAN" data type. Any valid numeric expression is allowed.

The conditional segment can be either a single BASIC statement or a program segment containing any number of statements. The first example shows conditional execution of a single BASIC statement.

```
100  IF Ph>7.7 THEN OUTPUT Valve USING "#,B";0
```

Notice the test (`Ph>7.7`) and the conditional statement (`OUTPUT Valve`...) which appear on either side of the keyword THEN. When the computer executes this program 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 OUTPUT statement is executed. If you don't already understand logical and relational operators, refer to the "Numeric Computation" and "String Manipulation" chapters.

By the way, the image specifier `#,B` causes the output of a single byte. In the example, the value for that byte is specified as zero (all bits cleared). Presumably, this turns off all devices connected to a GPIO interface. That interface is specified by the value contained in the device selector `Valve`. It is beyond the scope of this manual to explain the details of controlling valves and instruments. If you want to do this kind of control, refer to the *BASIC Interfacing Techniques* manual and study the manual that came with the interface.

The same variable is allowed on both sides of an IF...THEN statement. For example, the following statement could be used to keep a user-supplied value within bounds.

```
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 of `Number` is greater than nine, the conditional assignment is performed, replacing the original value in `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 used for various purposes, but the "common denominator" is that the computer needs to find them during prerun as the first keyword on a line. (A possible exception to this reasoning is REM, which is not allowed because it makes no sense to allow it. Comments certainly aren't executed conditionally. If comments are necessary on an IF...THEN line, the exclamation point can be used.) The following statements are not allowed in a single-line IF...THEN.

Keywords used in the declaration of variables:

COM
DIM
INTEGER
OPTION BASE
REAL

Keywords that define context boundaries:

DEF FN
SUB
END
FNEND
SUBEND

Keywords that define program structures:

CASE
CASE ELSE
ELSE
END IF
END LOOP
END SELECT
END WHILE
EXIT IF
FOR
IF
LOOP
NEXT
REPEAT
SELECT
UNTIL
WHILE

Keywords used to identify lines that are literals:

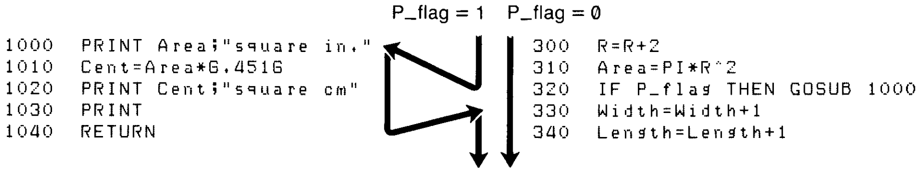
DATA
REM

Conditional Branching

Powerful control structures can be developed by using branching statements in an IF...THEN. Here are some examples.

```
110  IF Free_space<100 THEN GOSUB Expand_file
120  !  The line after is always executed
```

This statement checks the value of a variable called `Free_space`, and executes a file-expansion subroutine if the value tested is not large enough. The same technique can be used with a CALL statement to invoke a subprogram conditionally. 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. This is illustrated in the following drawing.



```
1000   PRINT Area;"square in."
1010   Cent=Area*6.4516
1020   PRINT Cent;"square cm"
1030   PRINT
1040   RETURN
```

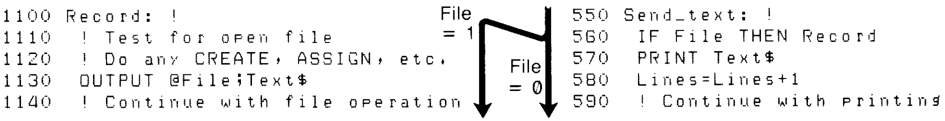
```
300   R=R+2
310   Area=PI*R^2
320   IF P_flag THEN GOSUB 1000
330   Width=Width+1
340   Length=Length+1
```

Figure 2-4. Program Flow with Conditional Subroutine

The conditional GOTO is such a commonly used technique that the computer allows a special case of syntax to specify it. Assuming that line number 200 is labeled "Start", the following statements will 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 improves the readability of programs, because phrases like "then start" sound more like English and less like computer jargon.) If execution is redirected by a conditional GOTO (implied or expressed), the program flow does not automatically return to the line following the IF...THEN. Thus, a conditional GOTO acts like a switch on a railroad track. This is illustrated in the following drawing.



```
1100 Record: !
1110   ! Test for open file
1120   ! Do any CREATE, ASSIGN, etc.
1130   OUTPUT @File;Text$
1140   ! Continue with file operation
```

```
550 Send_text: !
560   IF File THEN Record
570   PRINT Text$
580   Lines=Lines+1
590   ! Continue with printing
```

Figure 2-5. Program Flow with Conditional GOTO

Multiple-Line Conditional Segments

If the conditional program segment requires more than one statement, a slightly different structure is used. Let's expand the valve-control example.

```
100  IF Ph>7.7 THEN
110    OUTPUT Valve USING "#,B";0
120    PRINT "Final Ph =";Ph
130    GOSUB Next_tube
140  END IF
150  !  Program continues here
```

Any number of program lines can be placed between a THEN and an END IF statement. In executing this example, the computer evaluates the expression `Ph>7.7` in the IF...THEN statement. If the result is false, the program counter is set to 150, and execution resumes with the line following the END IF statement. If the condition is true, the program counter is set to 110, and the three conditional statements (lines 110, 120, 130) are executed. Program flow then picks up at line 150, because the END IF is only used during prerun.

When using multiple-line IF...THEN structures, remember to mark the end of the structure with an END IF statement and don't put any of the statements on the same line as the IF...THEN. If the beginning and end of the structure are not properly marked, the computer reports error 347 during prerun.

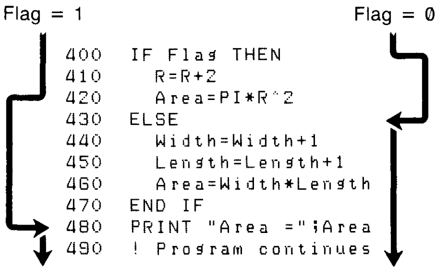
The conditional segment can contain any statement except one that is used to set context boundaries (such as END or DEF FN). In the previous example, the `GOSUB Next_tube` could have been a `GOTO Next_tube`. In that case, program execution does not pass through 150 when the condition is true. A false condition would cause a branch to line 150, while a true condition would send execution from line 100, to 110, to 120, to 130, and then to the line labeled "Next_tube."

If structuring statements are 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
1010    IF End_of_page THEN
1020      FOR I=1 TO Skip_length
1030        PRINT
1040        Lines=Lines+1
1050      NEXT I
1060    END IF
1070  END IF
```

Choosing One of Two Segments

Often you want a program flow that passes through only one of two paths depending upon a condition. This type of decision is represented pictorially by the following diagram. If you have ever been forced to program this type of structure using only the conditional GOTO, you know that the result is much more confusing than it needs to be.



```
400  IF Flag THEN
410    R=R+2
420    Area=PI*R^2
430  ELSE
440    Width=Width+1
450    Length=Length+1
460    Area=Width*Length
470  END IF
480  PRINT "Area =";Area
490  ! Program continues
```

Figure 2-6. Choosing One of Two Segments

This language has an IF...THEN...ELSE structure which makes the one-of-two choice easy and readable. The following example looks at a device selector which may or may not contain a primary address. The variable Isc is needed later in the program and must be only an interface select code. If the operator-supplied device selector is greater than 31, the interface select code is extracted from it. If it is equal to or less than 31, it already is an interface select code. (This example assumes that no secondary addressing is used.)

```
500  IF Select>31 THEN
510    Isc=Select DIV 100
520  ELSE
530    Isc=Select
540  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. Like the previous example, the structure is terminated by END IF, and the proper nesting of other structures is allowed. The next example shows a program segment that removes certain "escape sequences" from a string. The number of bytes in the escape sequence varies, but can be determined by inspecting the characters following the escape code. Notice the nesting of structures and the conditional branch. When no more escape sequences remain in the string, program execution continues at Next_seq.

```
3800 Escape:  !
3810  Point=POS(A$,Esc$)
3820  IF NOT Point THEN Next_seq
3830  IF A$[Point+1;1]<>"&" THEN
3840    A$[Point]=A$[Point+2]       ! 2-byte sequence
3850  ELSE
3860    IF A$[Point+2;1]="d" THEN
3870      A$[Point]=A$[Point+4]     ! 4-byte sequence
3880    ELSE
3890      A$[Point]=A$[Point+5]     ! 5-byte sequence
3900    END IF
3910  END IF
3920  GOTO Escape                   ! Look for more
3930  !
3940 Next_seq:               ! Program continues here
```

Choosing One of Many Segments

Using SELECT Constructs

Consider as an example the processing of readings from a voltmeter. In this example, we assume that the reading has already been entered, and it contained a function code. These hypothetical function codes identify the type of reading and are shown in the following table.

Table 2-1. Function Codes

Function Code	Type of Reading
DV	DC Volts
AV	AC Volts
DI	DC Current
AI	AC Current
OM	Ohms

The first example shows the use of the SELECT construct. The function code is contained in the variable `Funct$`. For the sake of simplicity, the example does not show any actual processing. Comments are used to identify the location of the processing segments. The rules about illegal statements and proper nesting are the same as those discussed previously in the IF...THEN section.

```
2000  SELECT Funct$
2010  CASE "DV"
2020    !
2030    ! Processing for DC Volts
2040    !
2050  CASE "AV"
2060    !
2070    ! Processing for AC Volts
2080    !
2090  CASE "DI"
2100    !
2110    ! Processing for DC Amps
2120    !
2130  CASE "AI"
2140    !
2150    ! Processing for AC Amps
2160    !
2170  CASE "OM"
2180    !
2190    ! Processing for Ohms
2200    !
2210  CASE ELSE
2220    BEEP
2230    PRINT "INVALID READING"
2240  END SELECT
2250  ! Program execution continues here
```

Notice that the SELECT construct starts with a SELECT statement specifying the variable to be tested and ends with an END SELECT statement. The anticipated values are placed in CASE statements. Although this example shows a string tested against simple literals, the SELECT statement works for 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 variable.

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 numeric variable tested with comparison operators and a range specifier.

```
1500  SELECT Ds
1510  CASE <1
1520    ! Processing for invalid device selector
1530  CASE 1 TO 31
1540    ! Processing for interface select code
1550  CASE >31
1560    ! Contains primary address
1570  END SELECT
```

A CASE statement can also specify multiple matches by separating them with commas, as shown below.

```
CASE -1,1,3 TO 7,>15
```

The following CASE statement shows the use of a string expression, rather than a simple constant.

```
CASE CHR$(27)&")@"&Eol$
```

You should be aware that if an error occurs when the computer tries to evaluate an expression in a CASE statement, the error is reported for the line containing the SELECT statement. This is a result of the nature of SELECT constructs and is not a bug. However, it can make things a bit confusing if you aren't aware of it. An error message pointing to a SELECT statement actually means that there was an error in that line *or* in one of the CASE statements. It requires more "detective work" on your part to locate the line which actually contains the erroneous expression.

Using the ON Statement

This type of program flow can also be generated with the ON statement and some additional processing. Let's do a string example first, using the previous voltmeter example. All the anticipated values are placed in a simple string. This string is then searched using the POS function. The results of the POS function are adjusted to become consecutive integers beginning with one. This result can then be used in the ON statement.

```
100  Match$="DVAVDIAIOM"
 .
 .
 .
500  Pointer=POS(Match$,Funct$)
510  Pointer=INT((Pointer-1)/2+1)
520  ON Pointer+1 GOSUB Case_else,Case_dv,Case_av,
Case_di,Case_ai,Case_om
```

Notice that a match can only cause values of 1, 3, 5, 7, or 9 from the POS function. A "match not found" gives a value of 0. Line 510 converts these to consecutive integers from 0 thru 5. The `Pointer+1` expression in line 520 shifts the values to a range 1 thru 6, which is acceptable to the ON statement.

The values of the match characters will determine the "pre-processing" necessary. If you are trying to match single bytes, simply adding one to the results of the POS is all that is necessary. Finding 3-letter sequences requires a line like 510, only with a division by 3. Note also that, except for single bytes, this method may not always work. For example, if the current ranges had been indicated by DA and AA (instead of DI and AI), `Match$` would be "DVAVDAAAOM." A subsequent search for "AA" would return 6 instead of 7—not good. In a case like that, there are two choices. One approach is to rearrange the string being searched; "DVAVDAOMAA" would work. Perhaps the items in the string could be separated with a "pad" character and the calculation adjusted accordingly. The other approach is to make each match value a separate element of a string array. The array could then be searched with a FOR...NEXT loop. This approach works well to resolve conflicts, especially with long match strings. However, the extra code lines and array accesses slow the process down significantly.

The ON statement can also be used for numeric values. If the numeric values you are trying to match just happen to be consecutive integers starting with one, the variable to be tested can be used in the ON statement. However, programmers don't usually get that lucky. To match arbitrary values, the following trick can be used. This example tests the three cases: <0, 1, and >1.

```
700  Pointer=1*(X<0)+2*(X=1)+3*(X>1)
710  ON Pointer GOSUB Negative,One,Greater
```

Assuming that you use non-overlapping comparison tests, only one of the values in parentheses will be true. The system returns a value of "1" for true. This is multiplied times the corresponding factor to give the final value to `Pointer`. All the other factors drop out because their comparison result is zero. Programmers who like strong type-checking may raise an eyebrow at this technique, but it works.

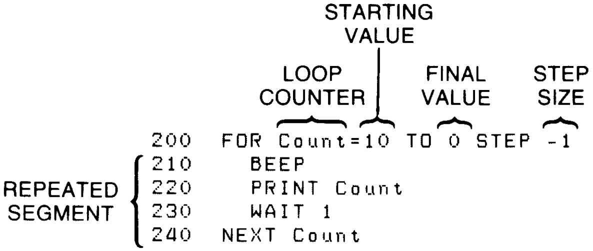
Another way of testing for numbers that are integers between 0 and 255 is to use the CHR$ function to create string bytes and apply the POS function as explained previously.

Repetition

Humans usually prefer tasks with variety that avoid tedious repetition. A computer does not have this shortcoming. You have four structures available for creating repetition. The FOR...NEXT structure is used for repeating a program segment a predetermined number of times. Two other structures (REPEAT...UNTIL and WHILE...END WHILE) are used for repeating a program segment indefinitely, waiting for a specified condition to occur. The LOOP...EXIT IF structure is used to create an iterative structure that allows multiple exit points at arbitrary locations.

Fixed Number of Iterations

The general concept of repetitive program flow can be shown with the FOR...NEXT structure. With this structure, 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 structure 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=10 TO 0 STEP -1
210    BEEP
220    PRINT Count
230    WAIT 1
240  NEXT Count
```

Figure 2-7. FOR...NEXT Loop Structure

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 will be used to take the loop counter from the starting value to the final value. When the loop counter is an INTEGER, the number of iterations can be predicted using the following formula:

$$\text{INT}\left(\frac{\text{Step Size} + \text{Final Value} - \text{Starting Value}}{\text{Step Size}}\right)$$

Note that the formula applies to the values in the variables, not necessarily the numbers in the program source. For example, if you use an INTEGER loop counter and specify a step size of 0.7, the value will be rounded to one. Therefore, 1 should be used in the formula, not 0.7.

The loop counter can be a REAL number, with REAL quantities for the step size, starting, or final values. In some cases, using REAL numbers will cause the number of iterations to be off by one from the preceding formula. This is because the NEXT statement performs an "increment and compare," and there is a slight inaccuracy in the comparison of REAL numbers. If you are interested, this is discussed in the next chapter. However, there is no clean way around it with FOR...NEXT loops. Here is an example:

```
200  Count=0
210  FOR X=10 TO 20
220    Count=Count+1
230    PRINT Count
240  NEXT X
```

According to the formula, this loop should execute 11 times: INT((1+20−10)/1=11). The result on the CRT confirms this when the loop is executed. If line 210 is changed to:

```
210 FOR X=1 TO 2 STEP .1
```

the formula still yields 11 as the number of iterations. However, executing the loop produces only 10 repetitions. This is because of a very, very small accumulated error that results from the successive addition of one-tenth. The error is less significant than the 15th digit, but discernable to the computer. In this case, rounding cannot be performed at a time that would help. When you find yourself in this situation, one way out is to add a slight adjustment factor to the final value. The following line does give the 11 iterations predicted by the formula.

```
210 FOR X=1 TO 2.05 STEP .1
```

Remembering the "increment and compare" operation at the bottom of the loop is helpful. After the loop counter is updated, it is compared to the final value established by the FOR statement. If the loop counter has **passed** the specified final value, the loop is exited. If it has **not passed** the specified final value, the loop is repeated. The loop counter retains its exit value after the loop is finished. This is not necessarily one full step past the final value. For example:

```
FOR I=1 TO 9.9
```

This statement establishes a loop that executes nine times (the default step size is one). The variable `I` has the value 10 when the loop is exited.

```
FOR Count=12 TO 1 STEP -0.3
```

This statement establishes a loop that executes 37 times. The variable `Count` has the value .9 when the loop is exited. Notice that negative step sizes are allowed using the same keywords as positive step sizes.

The final points to mention concern the execution of the FOR statement. If any variables are present to the right of the equal sign, the value used is the value they have when the FOR statement is executed. Remember that the FOR statement is only executed once before the loop begins. Also, if the number of iterations evaluates to zero or less, the loop is not executed and program execution goes immediately to the line following the NEXT statement. Here are some examples.

```
400  FOR Item=First TO Last
410    GOSUB Process
420    Last=Last+1
430  NEXT Item
440  ! Execution continues here
```

This loop would not be executed if `Last` were less than `First`. This is almost always desirable, since it prevents the subroutine `Process` from being invoked with a null item. Also notice that the number of iterations is fixed at loop entry when line 400 is executed. That number of iterations does not change when the value of `Last` is changed.

```
FOR Item=Item+1 TO Last
```

The variable `Item` is used as the loop counter. It receives a starting value that is one greater than the value it had when this line is executed.

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. Consider a very simple example. The following segment asks the operator to input a positive number. Presumably, negative numbers are not acceptable. 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 takes once, that is just fine. If the poor operator takes ten tries before he realizes what the computer is asking for, so be it. 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
```

A typical use of this is an iterative problem involving non-linear increments. One example is musical notes. Performing the same operation on all the notes in a 3-octave band is a repetitive process, but not a linear one. Musical notes are related geometrically by the 12th root of two. The following example simply prints the frequencies involved, but your application could involve any number of operations.

```
1200  Note=110      ! Start at low A
1210  REPEAT
1220    PRINT Note;
1230    Note=Note*2^(1/12)
1240  UNTIL Note>880  ! End at high A
```

For this example, a FOR...NEXT loop might have been used, with the loop counter appearing in an exponent. That would work because it is relatively easy to know how many notes there are in three octaves of the musical scale. However, the REPEAT...UNTIL structure is more flexible than FOR...NEXT when working with exponential data in general. Examples often occur in the area of graphics. The following segment could be used to plot audio frequency data, where the x-axis is logarithmic.

```
1500  Freq=20
1510  MOVE LOG(Freq),FNFunction(Freq)
1520  REPEAT
1530    DRAW LOG(Freq),FNFunction(Freq)
1540    Freq=Freq*1.2
1550  UNTIL Freq>20000
```

The flexibility of this structure is in line 1540. By increasing the frequency with a factor of 1.2, a very fast but rough graph is generated. This lets you place axes, labels, markers, etc. where you want them without waiting for a time-consuming plot for each cosmetic change. Once you have the desired appearance, you could change line 1540 to `Freq=Freq*1.01`. This would greatly increase the resolution of the plot (and reduce its speed). To take it one step further, you could make the "resolution factor" a variable and input its value at the start of the program. That would make it easy to try many different increments to achieve the best compromise between resolution and smoothness. Attempting a similar technique with FOR...NEXT loops would involve many extra (and unnecessary) calculations.

The WHILE loop is used for the same purpose as the REPEAT loop. The only difference between the two is the location of the test for exiting the loop. The REPEAT loop has its test at the bottom. This means that the loop is always executed at least once, regardless of the value of the condition. The WHILE loop has its test at the top. Therefore, it is possible for the loop to be skipped entirely (if the conditions so dictate). The following segment shows the same plotting example using a WHILE loop.

```
1500  Freq=20
1510  MOVE LOG(Freq),FNFunction(Freq)
1520  WHILE Freq<=20000
1530    DRAW LOG(Freq),FNFunction(Freq)
1540    Freq=Freq*1.2
1550  END WHILE
```

The next segment shows the use of conditional branching to simulate a REPEAT...UNTIL structure.

```
1500  Freq=20
1510  MOVE LOG(Freq),FNFunction(Freq)
1520 Loop_top:  !
1530    DRAW LOG(Freq),FNFunction(Freq)
1540    Freq=Freq*1.2
1550  IF Freq<=20000 THEN Loop_top
```

The WHILE structure can also be simulated using GOTO statements. The following segment shows this technique.

```
1500  Freq=20
1510  MOVE LOG(Freq),FNFunction(Freq)
1520 Loop_top:  !
1530  IF Freq>20000 THEN Loop_exit
1540    DRAW LOG(Freq),FNFunction(Freq)
1550    Freq=Freq*1.2
1560  GOTO Loop_top
1570 Loop_exit:  !
```

The REPEAT...UNTIL and WHILE structures are especially useful for tasks that are impossible with a FOR...NEXT loop. One such situation is a loop where both the loop counter and the final value are changing. Consider the example of stripping all control characters from a string. This can't be done in a loop that starts `FOR I=1 TO LEN(A$)`, because the length of `A$` changes each time a character is deleted. Therefore, the loop counter used as a subscript will eventually exceed the length of the string by more than one, generating an error. The WHILE loop does not have this problem. Note that the test at the top of the loop prevents the subscripting from being attempted on a null string. This is necessary to avoid an error.

```
600  I=1
610  WHILE I<=LEN(A$)
620    IF A$[I;1]<CHR$(32) THEN
630      A$[I]=A$[I+1]
640    ELSE
650      I=I+1
660    END IF
670  END WHILE
```

Arbitrary Exit Points

A pass through any of the loop structures discussed so far included the entire program segment between the top and the bottom of the loop. There are times when this is not the desired program flow. The LOOP structure defines the repeated program segment and allows any number of conditional exits points in that segment.

For the first example, consider a search-and-replace operation on string data. In this example, the "shift out" control character is being used to initiate underlining on a printer that understands standard escape sequences. The "shift in" control character is used to turn off the underline mode. (There is nothing significant about this choice of characters. any combination of characters could serve the same purpose.)

One approach is to use a loop to search every character in every string to see if it is one of the special characters. There are two problems with this method. First, it is a little cumbersome when the replacement string is a different length than the target string. Second, it is slow. Admittedly, speed it not a significant consideration when driving common mechanical printers. But the destination might eventually be a laser printer or mass storage file, making the program's speed more visible.

A better approach is to use the POS function to locate the target string. Since this function locates only the first occurrence of a pattern, it must be placed in a loop to insure that multiple occurrences will be found. The LOOP structure is well suited to this task, as shown in the following example.

```
2000  LOOP
2010    Position=POS(A$,CHR$(14))
2020  EXIT IF NOT Position
2030    A$[Position]=CHR$(27)&"&dD"&A$[Position+1]
2040  END LOOP
2050  !
2060  LOOP
2070    Position=POS(A$,CHR$(15))
2080  EXIT IF NOT Position
2090    A$[Position]=CHR$(27)&"&d@"&A$[Position+1]
2100  END LOOP
2110  ! Last EXIT goes to here
```

In this segment, all occurrences of "shift out" are replaced by "escape &dD" to enable underline mode. All occurrences of "shift in" are replaced by "escape &d@" to disable underlining. Notice that there is no problem replacing one character with four (assuming that **A$** is large enough). Lines containing no special characters are processed by only two POS functions, which is much faster and cleaner than performing two comparisons for every character in every line.

Another common use for this structure is the processing of operator input. Recall the REPEAT...UNTIL example that tested for the input of a positive number. Although this structure kept the computer happy, it left the operator in the dark. The LOOP structure provides for the additional processing needed, as shown in the following example.

```
200 LOOP
210   INPUT "Enter a positive number.",Number
220 EXIT IF Number>=0
230   BEEP
240   PRINT
250   PRINT "Negative numbers are not allowed."
260   PRINT "Repeat entry with a positive number.
270 END LOOP
```

Another point to remember is that the LOOP structure permits more than one exit point. This allows loops that are exited on a "whichever comes first" basis. Also, the EXIT IF statement can be at the top or bottom of the loop. This means that the LOOP structure can serve the same purposes as REPEAT...UNTIL and WHILE...END WHILE, if that suits your programming style.

The EXIT IF statement must appear at the same nesting level as the LOOP statement for a given loop. This requirement is best shown with an example. In the "WRONG" example, the EXIT IF statement has been nested one level deeper than the LOOP statement because it was placed in an IF...THEN structure.

WRONG:

```
600 LOOP
610   Test=RND-.5
620   IF Test<0 THEN
630     GOSUB Negative
640   ELSE
650     EXIT IF Test>.4
660     GOSUB Positive
670   END IF
680 END LOOP
```

RIGHT:

```
600 LOOP
610   Test=RND-.5
620 EXIT IF Test>.4
630   IF Test<0 THEN
640     GOSUB Negative
650   ELSE
660     GOSUB Positive
670   END IF
680 END LOOP
```

Event-Initiated Branching

Your computer has a special kind of program flow that provides some very powerful tools. This tool, called *event-initiated branching*, uses interrupts to redirect program flow. The process can be visualized as a special case of selection. Every time program flow leaves a line, the computer executes an "event-checking" routine. This is a set of machine-language "if...then" statements concerning interrupts. If an event is:

- Enabled to initiate a branch (with an ON-event statement)
- The event occurs

Then this "event-checking" routine causes the program to branch (as specified in the ON-event statement).

The process of "event checking" is represented in the following lines. Notice that it is possible for event-initiated branching to occur at the end of any program line, which includes the lines of a subprogram. This can give the appearance of "middle-of-line" branching when it occurs during a user-defined function.

```
100  X=Radius*FNMy_function/COS(Angle)^^Exponent


                      Branch may occur while FNMy_function
                      is being executed.
```

In the following example illustration, these potential branching points are marked by the words *gosub event_check*. This does not refer to a BASIC subroutine, but is just a symbolic reminder of where event-initiated branching can occur. If the operating system finds an event has occurred (and the corresponding branch is currently enabled), then a branch is initiated. If not, program execution resumes with the "normal" program flow.

```
10  PRINT X (gosub event_check)
20  X=X+1   (gosub event_check)
30  GOTO 10 (gosub event_check)
```

Types of Events

Event-initiated branching is established by the ON..event statements. Here is a list of the statements that fall in this category, along with the corresponding event that causes a branch:

ON CDIAL an interrupt generated by turning a knob—a rotary pulse generator—on a Control Dial box (see the "Communicating with the Operator" chapter of this manual)

ON CYCLE cyclical (periodic, repetitive) interrupts from the clock (see the "Clock and Timers" chapter of this manual)

ON DELAY a one-time interrupt from the clock (see the "Clock and Timers" chapter of this manual)

ON END an interrupt upon reaching an end-of-file (EOF) condition (see the "Data Storage and Retrieval" chapter of this manual)

ON ERROR an interrupt when a run-time error is encountered (see the "Handling Errors" chapter of this manual)

ON EOR an interrupt when an end-of-record is encountered during a TRANSFER statement (see the "Data Storage and Retrieval" chapter of this manual)

ON EOT an interrupt when an end-of-TRANSFER condition is encountered (see the "Advanced Transfer Techniques" chapter of *BASIC Interfacing Techniques*)

ON HIL EXT an interrupt generated by an HP HIL (Human Interface Link) device (see the "HIL Interface" chapter of *BASIC Interfacing Techniques*)

ON INTR an interrupt generated by an an interface (see the "Interrupts" chapter of *BASIC Interfacing Techniques*)

ON KBD an interrupt generated by pressing a key (see the "Communicating with the Operator" chapter of this manual)

ON KEY an interrupt generated by pressing a softkey: f1 thru f8 on ITF keyboards, or k0 through k9 on 98203 keyboards (see the "Communicating with the Operator" chapter of this manual)

ON KNOB an interrupt generated by turning a knob—a rotary pulse generator: the built-in knob of a 98203 keyboard, an HIL knob, or one of the knobs on a Control Dial box (see the "Communicating with the Operator" chapter of this manual)

ON SIGNAL an interrupt generated by a SIGNAL statement (see the *BASIC Language Reference*)

ON TIME an interrupt from the clock when the clock reaches a specified time (see the "Clock and Timers" chapter of this manual)

ON TIMEOUT an interrupt generated when an interface or device has taken longer than a specified time to respond to a data-transfer handshake (see the "Interrupts" chapter of *BASIC Interfacing Techniques*)

Example of Event-Initiated Branching

The best way to understand how event-initiated branches operate in a program is to sit down at the computer and try a few examples. Start by entering the following short program, which sets up and enables branches when one of the softkeys is pressed.

```
110  ON KEY 1 LABEL "Inc" GOSUB Plus
120  ON KEY 5 LABEL "Dec" GOSUB Minus
130  !
140 Spin:  DISP X
150  GOTO Spin
160  !
170 Plus:  X=X+1
180  RETURN
190  !
200 Minus:  X=X-1
210  RETURN
220  END
```

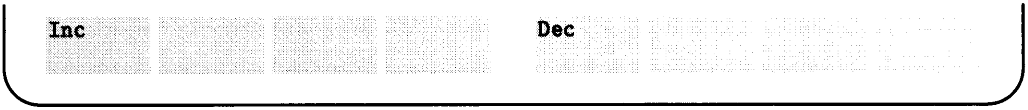
Notice the various structures in this sample program. The ON KEY statements are executed only once at the start of the program. Once defined, these event-initiated branches remain in effect for the rest of the program. (Disabling and deactivating are discussed later.)

The program segment labeled `Spin` is an infinite loop. If it weren't for interrupts, this program couldn't do anything except display a zero. However, there is an implied "if...then" at the end of each BASIC program line due to the ON KEY action. This allows a selection process to occur. Either the `Plus` or the `Minus` subroutine can be selected as a result of softkey presses. These are normal subroutines terminated with a RETURN statement. (In the context of interrupt programming, these subroutines are called **service routines**.) The following section of "pseudo-code" shows what the program flow of the `Spin` segment actually looks like to the computer.

```
  Spin: display X
   if Key0 then gosub Plus
   if Key5 then gosub Minus
goto Spin
```

This pseudo-code is an over-simplification of what is actually happening, but it shows that the `Spin` segment is not really an infinite loop with no decision-making structure. Actually, most programs that use event-initiated branching to control program flow will contain what appears to be an infinite loop. That is the easiest way to "keep the computer waiting" while it is waiting for an interrupt.

Now run the sample program you just entered. Notice that the bottom two lines of the screen display an inverse-video label area (this one is shown when using an ITF keyboard).



These labels are arranged to correspond to the layout of the softkeys. The labels are displayed when the softkeys are active and are not displayed when the softkeys are not active.[1] Any label which your program has not defined is blank unless the system defines it. The label areas are defined in the ON KEY statement by using the keyword "LABEL" followed by a string.

The starting value in the display line is zero, since numeric variables are initialized to zero at prerun. Each time you press k1 or f1, the displayed value of `X` is incremented. Each time you press k5 or f5, the displayed value of `X` is decremented. This simple demonstration should acquaint you with the basic action of the softkeys.

[1] See the "Communicating with the Operator" chapter for additional examples of softkeys and labels.

It is possible to make structures that are much more elaborate, with assignable priorities for each key, and keys that interrupt the service routines of other keys. There are many applications where priorities are not of any real significance, such as the example program running now. However, priorities will sometimes cause unexpected flow problems. One type of priority problem can be shown with a simple modification to our example program. Insert the following line right after line 170.

```
171 GOTO 171
```

Now run the program and press k1 or f1. Notice that the program "locks up" and all subsequent presses of either softkey do nothing. This is not simply because line 171 creates an infinite loop. The program segment at `Spin` was a infinite loop and that didn't bother the softkeys at all. The problem is that the priority for the `Plus` service routine is higher than the main program priority. None of the softkeys have been assigned a high enough priority to interrupt another service routine. A full discussion on interrupt priority can be found in the "Interface Events" chapter of the *BASIC Interfacing Techniques* manual. If you think you have an application that is "priority sensitive," read that section carefully.

Example of Using the Knob

One characteristic of interrupt-driven program flow is that the computer's decisions can be more easily synchronized with the actions of devices connected to it. This type of application is often called **real-time programming**. An important example of real-time programming is machine control. A computer running an automatic packing machine must turn off the flow immediately when the jar is full. It is not acceptable for the computer to wait until the inventory printout is done and peanut butter is dumped all over the conveyor belt. Although machine control applications are very important, their extensive interfacing makes them inconvenient or impossible to use as demonstration programs in a manual such as this.

Another common example of real-time programming is computer games. The computer is expected to respond "instantly" to button presses, lever movement, etc. The operator expects immediate correlation between their input and the computer's output or display. Your BASIC Utilities Disc has a couple of simple games on it that demonstrate interaction between the CRT, softkeys, and knob. Feel free to list any of the programs on that disc if you want further examples of various techniques.

The following program is a very short example that demonstrates a real-time interaction between the knob and the CRT. If you run this example program and turn the knob, you will see the kind of interaction that might be used for cursor control in a text editor. Obviously, a real cursor-control routine would be much more sophisticated, but this demonstrates the basic idea. (The "Communicating with the Operator" chapter also describes using the knob.)

```
10   ON KNOB .1 GOSUB Move_blip
20 Spin:  GOTO Spin
30   !
40 Move_blip:  !
50   PRINT TABXY(Spotx,Spoty);" ";
60   Spotx=Spotx+KNOBX/5
70   Spoty=Spoty+KNOBY/5
80   IF Spoty<1 THEN Spoty=1
90   IF Spoty>18 THEN Spoty=18
100  IF Spotx<1 THEN Spotx=1
110  IF Spotx>50 THEN Spotx=50
120  PRINT TABXY(Spotx,Spoty);CHR$(127);
130  RETURN
140  END
```

This example uses a short infinite loop to wait for pulses from the knob (line 20). Interrupts from the knob are enabled by the ON KNOB statement in line 10. The service routine erases the old "blip", performs some scaling and range checking on the knob input, and prints the new "blip".

The scaling and range checking are very important in this kind of interactive routine. Humans expect their interface to have a certain "feel." Displays should not change too quickly or too slowly. Certain kinds of displays are expected to change logarithmically, others are expected to change linearly. The boundary values of variables are expected to conform to the boundaries of the display. To initiate yourself to some of these concepts, try modifying this simple example. Remove one or more of the range checking lines. (An easy way to do this kind of editing is to place an exclamation point in front of the statement. This turns it into a comment, removing it from the flow of execution. But it can be easily returned to the program by deleting the exclamation point.) Also try changing the scaling factor in lines 60 and 70. Notice the "feel" that results from larger and smaller increments, or even logarithmic scaling.

Deactivating Events

Knowing how to "turn off" the interrupt mechanism is just as important as knowing how to enable it. Often, an event is a desired input during one part of the program, but not during another. You might use softkeys to set certain process parameters at the start of a program, but you don't want interrupts from those keys once the process starts. For example, a report generating program could use a softkey to select single or double spacing. This key should be disabled once the printout starts so that an accidental key press does not cause the computer to abort the printout and return to the questions at the beginning of the program. On the other hand, you might want an "Abort" key that does precisely that. The important thing is that you decide on the desired action and make the computer obey your wishes.

Before going any further, let's explain some important terminology. There are two general methods for "turning off" an interrupt. If an interrupt source is **deactivated**, it no longer has any influence on program flow. You can press a deactivated key all day long and nothing will happen. However, if an event is **disabled**, its action has only been temporarily postponed. The computer remembers that a key was pressed while it was disabled, and the action for that key will occur at the earliest opportunity once the disabled state is removed. There are examples in this section to demonstrate the difference between these two conditions.

All the "ON-event" statements have a corresponding "OFF-event" statement. This is one way to deactivate an interrupt source.

- OFF KEY deactivates interrupts from the softkeys. If a softkey is pressed while deactivated, it does nothing.
- OFF KNOB deactivates the ON KNOB interrupts. Turning the knob while ON KNOB is deactivated causes normal scrolling on the CRT.

The following example shows one use of OFF KEY to disable the softkeys. (Note that k1 is used in the description. If you have an ITF keyboard, just substitute f1.) A softkey is used to select a parameter for a small printing routine. Each press of k1 increments and displays the step size that will be used as an interval between the printed numbers. When the desired step size has been selected, k4 is pressed to start the printout. Enter and run this example. Notice that with line 240 and 250 commented out, the **softkey menu**, or label area, never changes.

```
100 Begin:   !
110   ON KEY 1 LABEL " DELTA" GOSUB Step_size
120   ON KEY 4 LABEL " START" GOTO Process
130   Inc=1
140   DISP "Step Size = 1"
150   !
160 Spin: GOTO Spin               ! Wait for key press
170   !
180 Step_size:  !
190   Inc=Inc+1                   ! Change increment
200   DISP "Step Size =";Inc
210   RETURN
220   !
230 Process:  !
240 ! OFF KEY
250 ! ON KEY 8 LABEL " ABORT" GOTO Leave
260   Number=0
270   FOR I=1 TO 10
280     Number=Number+Inc
290     PRINT Number;
300     WAIT .6
310   NEXT I
320 Leave:  !
330   OFF KEY 8                   ! Deactivate ABORT
340   PRINT                       ! End line
350   GOTO Begin                  ! Start over
360   END
```

Now run the example again and press k1 or k4 while the printout is in progress. Notice that the keys are still active and produce undesired effects on the printing process. To "fix this bug," remove the exclamation point from line 240. This disables all the softkeys when the printing process starts. Notice that the softkey menu goes away when no softkeys are active. This is a very handy feature while you are experimenting with interrupts. It provides immediate feedback to indicate when interrupts are active and when they are not.

Finally, remove the exclamation point from line 250. Now, the softkey menu appears during the printing process. However, the choices are different than at the start of the program. The keys used to select the parameter and start the process are not active, because they are not needed at this point in the program. Instead, k8 can be used to "gracefully" abort the process and return to the start of the program.

The OFF KEY statement can include a key number to deactivate a selected key. This was done in line 330.

Disabling Events

All the previous examples have shown complete deactivation of the softkeys. It is also possible to temporarily disable an event-initiated branch. This is done when an active event is desired in a process, but there is a special section of the program that you don't want to be interrupted. Since it is impossible to predict when an external event will occur, the special section of code can be "protected" with a DISABLE statement. This is sometimes necessary to prevent a certain variable from being changed in the middle of a calculation or to insure that an interface polling sequence runs to completion. It is difficult in a short, simple example to show *why* you would need to do this. But it is not difficult to show *how* to do it.

```
100   ON KEY 9 LABEL " ABORT" GOTO Leave
110   !
120 Print_line:  !
130   DISABLE
140   FOR I=1 TO 10
150     PRINT I;
160     WAIT .3
170   NEXT I
180   PRINT
190   ENABLE
200   GOTO Print_line
210   !
220 Leave: END
```

This example shows a DISABLE and ENABLE statement used to "frame" the `Print_line` segment of the program. The "ABORT" key is active during the entire program, but the branch to exit the routine will not be taken until an entire line is printed. The operator can press the "ABORT" key at any time. The key press will be **logged**, or remembered, by the computer. Then when the ENABLE statement is executed, the event-initiated branch is taken. Enter and run the example to observe this method of delaying interrupt servicing.

Chaining Programs

With this BASIC system, it is also possible to "chain" programs together; that is, one program may be executed, which in turn loads and runs another, which in turn loads and runs yet another, and so forth. This method is often used when you have several large program segments that will not all fit into memory at the same time[1]. This section describes the available methods.

Using LOAD

The LOAD statement clears the current program, brings in a program from a PROG file, with the *option* of beginning program execution at a specified line. This type of LOAD is performed by adding a line identifier. For example, the following command tells the computer to load the program in file "STONE" and begin execution at line 10:

```
100  LOAD "STONE",10
```

The line identifier may be a label or a line number, but it must identify a line in the main program segment (not in a subprogram or user-defined function).

If you want to communicate any information to the program that is being loaded, you have the following two methods:

- Store the information in a file which both programs can access. (File access is fully explained in the "Data Storage and Retrieval" chapter.)
- Store the information in "common" (COM) variables which both programs can access. (Note that the programs must have *identical* COM declarations. COM is fully discussed in the "Subprograms and User-Defined Functions" chapter; a simple example is provided in the subsequent section describing how to use GET to chain programs.)

The LOAD command cannot be used to bring in arbitrary program segments or append to a main program like GET can.

[1] This technique is similar to loading and running subprograms, but not nearly as powerful and flexible. See the "Subprograms and User-Defined Functions" chapter of this manual for details.

Using GET

The GET command is used to bring in programs or program segments from an ASCII file. with the options of appending them to an existing program and/or beginning program execution at a specified line.

The following statement:

```
GET "George",100
```

first deletes all program lines from 100 to the end of the program, and then and appends the lines in the file named "George" to the lines that remained at the beginning of the program. The program lines in file "George" would be renumbered to start with line 100.

GET can also specify that program execution is to begin. This is done by specifying two line identifiers. For example:

```
100  GET "RATES",Append_line,Run_line
```

specifies that:

1. Existing program lines from the line label "Append_line" to the end of the program are to be deleted.
2. Program lines in the file named "RATES" are to be appended to the current program, beginning at the line labeled "Append_line"; lines of "RATES" are renumbered if necessary.
3. Program execution is to resume at the line labeled "Run_line".

Although any combination of line identifiers is allowed, the line specified as the start of execution must be in the main program segment (not in a SUB or user-defined function). Execution will not begin if there was an error during the GET operation.

Example of Chaining with GET

A large program can be divided into smaller segments that are run separately by using GET (or LOAD). The following example shows a technique for implementing this method.

First Program Segment:

```
10  COM Ohms,Amps,Volts
20  Ohms=120
30  Volts=240
40  Amps=Volts/Ohms
50  GET "Wattage"
60  END
```

Program Segment in File Named "Wattage":

```
10  COM Ohms,Amps,Volts
20  Watts=Amps*Volts
30  PRINT "Resistance (in ohms)   = ";Ohms
40  PRINT "Power usage (in watts) = ";Watts
50  END
```

Lines 10 through 40 of the first program are executed in normal, serial fashion. Upon reaching line 50, the system deletes all program lines of the program and then GETs the lines of the "Wattage" program. Note that since they have similar COM declarations, the COM variables are preserved (and used by the second program). This feature is very handy to have while chaining programs.

Program-to-Program Communications

As shown in the preceding example, if chained programs are to communicate with one another, you can place values to be communicated in COM variables. The only restriction is that these COM declarations must *match exactly*, or the existing COM will be cleared when the chained program is loaded. For a description of using COM declarations, see the "Subprograms" chapter of this manual.

One important point to note is the use of the COM statement. The COM statement places variables in a section of memory that is preserved during the GET operation. Since the program saved in the file named "Wattage" also has a COM statement that contains three scalar REAL variables, the COM is preserved (it matches the COM declaration of the "Wattage" program being appended with GET).

If the program segments did not contain matching COM declarations, all variables in the mis-matched COM statements would be destroyed by the "pre-run"[1] that the system performs after appending the new lines but before running the first program line.

[1] For a definition of pre-run, see the "Loading and Running Programs" chapter of *Using the BASIC System*; or see the "Subprograms and User-Defined Functions" chapter of this manual.

Here is another example of chaining. It is a "file executive" program that gets and runs other programs. This example further demonstrates how several chained programs can use COM.

```
100   ! This program manages the utility files for
110   ! some hypothetical data.
115   !
120   ! Data items were stored in REAL array "Weights".
125   !
130   ! The printer is selected in line xxx.
135   !
140   ! Each utility sends control back to the line "Start"
150   ! when it finishes.
155   !
160   ! (Note that the disc containing these programs must
170   !  be on-line while the "executive" program is running.)
180   !
185   OPTION BASE 1
190   COM Weights(5000),Samples,Printer
200   Printer=701
210   Samples=5000
220   !
230 Start:               !  Main "entry point".
240   PRINTER IS CRT     !  PRINT on screen.
250   PRINT
260   PRINT "Enter P to print weights."
270   PRINT "Enter A for an analysis."
280   !
290 Ask:  INPUT "Enter Command Letter.",In$
300   IF UPC$(In$[1,1])="P" THEN GET "Printout",330,330
300   IF UPC$(In$[1,1])="A" THEN GET "Analysis",330,330
320   GOTO Ask
330   !
340   END
```

Here is the "Printout" program.

```
100  ! This segment prints the data in the array "Weights"
110  ! on a printer.
120  ! Necessary variables are initialized in the
130  ! main "executive" program.
140  !
150  PRINTER IS Printer  ! Print on an external printer.
160  FOR I=1 to Samples
170     PRINT "Sample #";I;" weighs ";Weights(I)
180  NEXT I
190  PRINT CHR$(12)
200  GOTO Start          ! Return to "main executive".
210  !
220  END
```

Here is the "Analysis" program.

```
100  ! This segment calculates the mean and standard deviation
110  ! of the data in "Weights".
120  ! Necessary variables are also initialized in the
130  ! main "executive" program.
140  !
150  Sumx=0
160  Sumx2=0
170  FOR I=1 to Samples
180     Sumx=Sumx+Weights(I)     ! Sum values.
190     Sumx2=Sumx2+Weights(I)^2 ! Sum squares of values.
200  NEXT I
210  Mean=Sumx/Samples
220  Std_dev=SQRT(Sumx2-Sumx^2/Samples)
230  PRINT
240  PRINT "Number of samples =";Samples
250  PRINT "Mean weight       =";Mean
260  PRINT "Std. deviation    =";Std_dev
270  GOTO Start          ! Return to "main executive".
280  !
290  END
```

Notice that any information shared by all routines is placed in COM. The individual task files do not contain COM statements, because the COM in the executive program is never deleted. In that manner, a standard characteristic can be changed in the executive, with no alterations required in any of the other files. The "shared" variables in this example are the number of weights (Weights array), the number of weight values (Samples), and the device selector of the external printer (Printer).

Numeric Computation 3

Numeric Computation 3

When most people think about computers, the first thing that they think of is number-crunching, the giant calculator with a brain. Whether this is an accurate impression or not, numeric computations are an important part of computer programming.

Numeric computations deal exclusively with numeric values. Thus, adding two numbers and finding a sine or a logarithm are all numeric operations; while converting bases and converting a number to a string or a string to a number are not. (Converting bases and converting numbers to strings and strings to numbers are covered in the chapter on "String Manipulation.")

Numeric Data Types

There are three numeric data types available in BASIC: INTEGER, REAL and COMPLEX. This section covers these data types.

REAL Data Type

Any numeric variable that is not declared an INTEGER or COMPLEX variable is a REAL variable. The valid range for REAL numbers is approximately:

$-1.797\,693\,134\,862\,315 \times 10^{308}$ thru $1.797\,693\,134\,862\,315 \times 10^{308}$

or

−`MAXREAL` thru +`MAXREAL`

which are the functions used to obtain the above range values.

The smallest non-zero REAL value allowed is approximately:

$\pm\ 2.225\,073\,858\,507\,202 \times 10^{-308}$ or ±`MINREAL`

A `REAL` can also have the value of zero.

INTEGER Data Type

An INTEGER can have any whole-number value from:

−32 768 thru +32 767

COMPLEX Data Type

A complex number is an ordered pair (x,y) denoted by Mathematicians as:

$x + iy$

where:

x is the real part of the complex number.

y is the imaginary part of the complex number. The i in front of the y forms the imaginary number iy and is the same as multiplying y by the $\surd$-1. For example, the $\surd$-9 could be written as: $\surd\text{-}1 \times \surd 9$ or $3i$.

BASIC complex numbers are stored as two REAL numbers. This means that a complex number requires 16 bytes of memory (each REAL component takes 8 bytes).

REAL, INTEGER, and COMPLEX variables may be declared as arrays.

Declaring Variables

It is good programming practice to declare the data type of all variables used in a program. The INTEGER, REAL, and COMPLEX statements have been provided to accomplish this task. However, BASIC is forgiving and implicitly assumes a variable is REAL if its type is not explicitly declared. Here are some examples of explicitly declaring variables:

```
INTEGER I, J, Days(5), Weeks(5:17)
REAL X, Y, Voltage(4), Hours(5,8:13)
COMPLEX S, T, Phasor_1(10), Phasor_2(10)
```

Each of the above statements declares two scalar and two array variables. A scalar is a variable which can, at any given time, represent a single value. An array is a subscripted variable that may contain multiple values accessed by subscripts. It is possible to specify both the lower and upper bounds of an array or to specify the upper bound only, and use the existing OPTION BASE as the lower bound. Details on declarations of arrays and how to use them are provided later in this chapter when arrays are dealt with in detail. The DIM statement may also be used to declare a REAL array.

```
DIM R(4,5)
```

An ALLOCATE statement can be used to declare REAL, INTEGER, and COMPLEX arrays. The ALLOCATE statement allows you to dynamically allocate memory in programs which need tight control over memory usage.

```
ALLOCATE REAL Co_ords(2,1:Points), INTEGER Status(1:Points)
ALLOCATE COMPLEX Poles(2,1:Points), REAL Location(2,1:Points)
```

Dynamic memory can be deallocated with the DEALLOCATE statement. Some examples are:

```
DEALLOCATE Co_ords(*), Status(*)
DEALLOCATE Poles(*), Location(*)
```

Assigning Variables

The most fundamental numeric operation is the assignment operation, achieved with the LET statement. The LET statement originally required the keyword LET for BASIC interpreters, but your computer makes it optional. Thus the following statements are equivalent:

```
LET A = A + 1
A = A + 1
```

To assign values to COMPLEX variables, the variables must first be declared as COMPLEX. The following program lines show examples of assigning REAL and COMPLEX values to COMPLEX variables B, C, and D.

```
10  COMPLEX B,C,D
20  B=3.0          ! Real part = 3.0; imaginary part = 0.0.
30  C=CMPLX(3,4) ! Creates a COMPLEX value and assigns it to C.
40  D=CMPLX(Real_part,Imaginary_part)
50  B=D ! Assigns both the real and imaginary parts of D to B.
60  .
70  .
```

Implicit Type Conversions

The computer will automatically convert between REAL, INTEGER, and COMPLEX values in assignment statements and when parameters are passed by value in function and subprogram calls. The type conversion rules are:

- When a value is assigned to a variable, the value is converted to the data type of that variable.

  For example, the following program shows a REAL value being converted to an INTEGER:

  ```
  100 REAL Real_var
  110 INTEGER Integer_var
  120 Real_var = 2.34
  130 Integer_var = Real_var ! Type conversion occurs here.
  140 DISP Real_var, Integer_var
  150 END
  ```

  Executing this program returns the following result:

  ```
  2.34          2
  ```

INTEGER and REAL data types are converted to COMPLEX data types by adding an imaginary part of `0`.

```
100 COMPLEX Complex_var1, Complex_var2
110 REAL Real_var
120 INTEGER Integer_var
130 Real_var=1.22
140 Integer_var=4
150 Complex_var1=Real_var
160 Complex_var2=Integer_var
170 DISP Complex_var1, Complex_var2
180 END
```

Executing this program produces the following results:

```
1.22  0      4  0
```

COMPLEX data types are converted to INTEGER and REAL data types by dropping the imaginary part.

```
100 COMPLEX Complex_var
110 REAL Real_var
120 INTEGER Integer_var
130 Complex_var=CMPLX(1.22,4.11)
140 Real_var=Complex_var
150 Integer_var=Complex_var
160 DISP Real_var, Integer_var
170 END
```

Executing this program produces the following results:

```
1.22     1
```

- Conversions that occur within expression convert to the "highest" or most complicated data type before the operation occurs. For example:

  ```
  CMPLX(3,-1) + 4.56
  ```

  converts the REAL data type `4.56` to a COMPLEX value before the addition operation is performed.

When parameters are passed by value, the type conversion is from the data type of the calling statement's parameter to the data type of the subprogram's parameter. This type conversion occurs automatically. When parameters are passed by reference, the type conversion is not made and a type mismatch error will be reported if the calling parameter and the subprogram parameters are of different types.

Whenever numbers are converted from REAL to INTEGER representations, information can be lost. There are two potential problem areas in this conversion: rounding errors and range errors.

BASIC will automatically convert between types when an assignment is made. This presents no problem when an INTEGER is converted to a REAL. However, when a REAL is converted to an INTEGER, the REAL is rounded to the closest INTEGER value. When this is done, all information about the number to the right of the radix (decimal point) is lost. If the fractional information is truly not needed, there is no problem, but converting back to a REAL will not reconstruct the lost information — it stays lost.

Another potential problem with REAL to INTEGER conversions is the difference in ranges. While REAL values range from approximately -10^{308} to $+10^{308}$, the INTEGER range is only from −32 768 to +32 767 (approximately -10^4 thru $+10^4$). Obviously, not all REAL values can be rounded into an equivalent INTEGER value. This problem can generate INTEGER Overflow errors.

While the rounding problem is important, it does not generate an execution error. The range problem **can** generate an execution error, and you should protect yourself from crashing the program by either testing values before assignments are made, or by using ON ERROR to trap the error, and making corrections after the fact.

The following program segment shows a method to protect against INTEGER overflow errors (note that the variable X is REAL):

```
200  IF X > 32767 THEN X= 32767
210  IF X < -32768 THEN X = -32768
220  Intx = X
```

It is possible to achieve the same effect using MAX and MIN functions:

```
200 Y = MAX(MIN(X, 32767), -32768)
```

Both these methods avoid the overflow errors, but lose the fact that the values were originally out of range. If out-of-range is a meaningful condition, an error handling trap is more appropriate.

```
200 IF (-32768<=X) AND (X<=32767) THEN
210    Y = X
220 ELSE
230    GOSUB Out_of_range
240 END IF
```

Precision and Accuracy: The Machine Limits

Your computer stores all REAL variables with a sign, approximately 15 significant digits, and the exponent value. For most engineering and other applications, rounding errors are not a problem because the resolution of the computer is well beyond the limitations of most scientific measuring devices. However, when high-resolution numerical analysis requires accuracy approaching the limits of the computer, round-off errors must be considered.

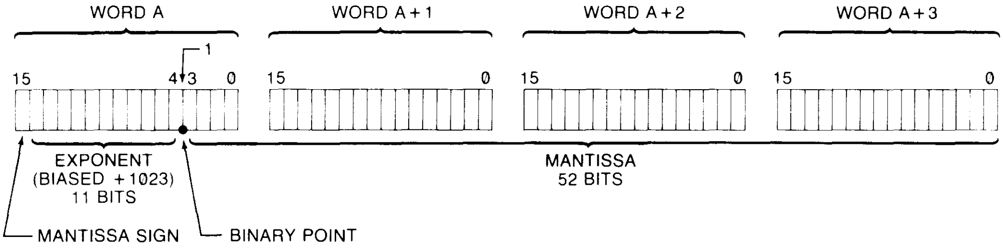
Rounding errors should be considered when conversions are made between decimal digits and binary form (the form used by the computer internally to store the values). Input and output operations are occasions when this can occur. Given the format used for REALs, the conversion REAL → decimal → REAL will yield an identity only if the REAL → decimal conversion produces a 17-decimal-digit mantissa and the calculations for the conversions are done in extra precision. This is not the case on Series 200/300 BASIC. Therefore, several things can be said about these conversions on Series 200/300 BASIC:

- Up to and including 16 decimal digits are allowed when storing a number in internal form. If there are more digits, they are ignored.
- Up to and including 15 decimal digits may be output when converting a REAL for printing, display, etc.. A full 16-digit conversion is not allowed because there are not 16 full digits of precision.
- It is possible for two distinct decimal numbers to map onto the same REAL number because the binary mantissa does not have enough bits to represent all 16 decimal digits. This can happen only if the decimal numbers are specified to 16-digits.
- It is possible for two distinct REAL numbers to convert to the same decimal number even if the conversion is done to 15-decimal-digit accuracy. Therefore, you cannot use a comparison of the digits in printed or displayed numbers to check for equality.
- All distinct 15 digit decimal strings have a correct distinct REAL representation, but it is not always possible to map them onto their correct representation because REAL multiplies are not done in extra precision, and the table entries are only 64 bits. In other words, the decimal → REAL conversion may produce a REAL that differs from the true representation by a maximum of two bits.

There are references at the end of this chapter to documents that contain further information on the subject of representing real numbers.

Internal Numeric Formats

The storage format for REAL and INTEGER numbers in memory are as follows:



$$-1^{\text{mantissa sign}} \text{ x } 2^{\text{exponent} \quad 1023} \text{ x } 1._{\text{mantissa}}$$

Figure 4-1. Storage Format for REAL Variables

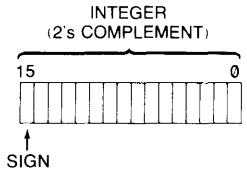


Figure 4-2. Storage Format for INTEGER Variables

Note that COMPLEX values are stored as 2 REAL numbers with the real part first and the imaginary part following.

Evaluating Scalar Expressions

This section covers the following topics as they relate to evaluating scalar expressions.

- Hierarchy of expression evaluation
- Delayed assignment surprise
- BASIC operators: monadic, dyadic, and relational

The Hierarchy

If you look at the expression 2+4/2+6, it can be interpreted several ways:

- 2+(4/2)+6 = 10
- (2+4)/2+6 = 9
- 2+4/(2+6) = 2.5
- (2+4)/(2+6) = .75

Computers do not deal well with ambiguity, so a hierarchy is used for evaluating expressions to eliminate any questions about the meaning of an expression. When the computer encounters a mathematical expression, an expression evaluator is called. If you do not understand the expression evaluator, you can easily be surprised by the value returned for a given expression. In order.to understand the expression evaluator, it is necessary to understand the valid elements in an expression and the evaluation hierarchy (the order of evaluation of the elements).

Six items can appear in a numeric expression:

- Operators (+, −, etc.)—modify other elements of the expression.
- Constants (7.5, 10, etc.)—represent literal, non-changing numeric values.
- Variables (Amount, X_coord, etc.)— represent changeable numeric values.
- Intrinsic functions (SQRT, DIV, etc.)—return a value which replaces them in the evaluation of the expression.
- User-defined functions (FNMy_func, FNReturn_val, etc.)—also return a value which replaces them in the evaluation of the expression.
- Parentheses—are used to modify the evaluation hierarchy.

The following table defines the hierarchy used by the computer in evaluating numeric expressions.

Table 3-1. Math Hierarchy

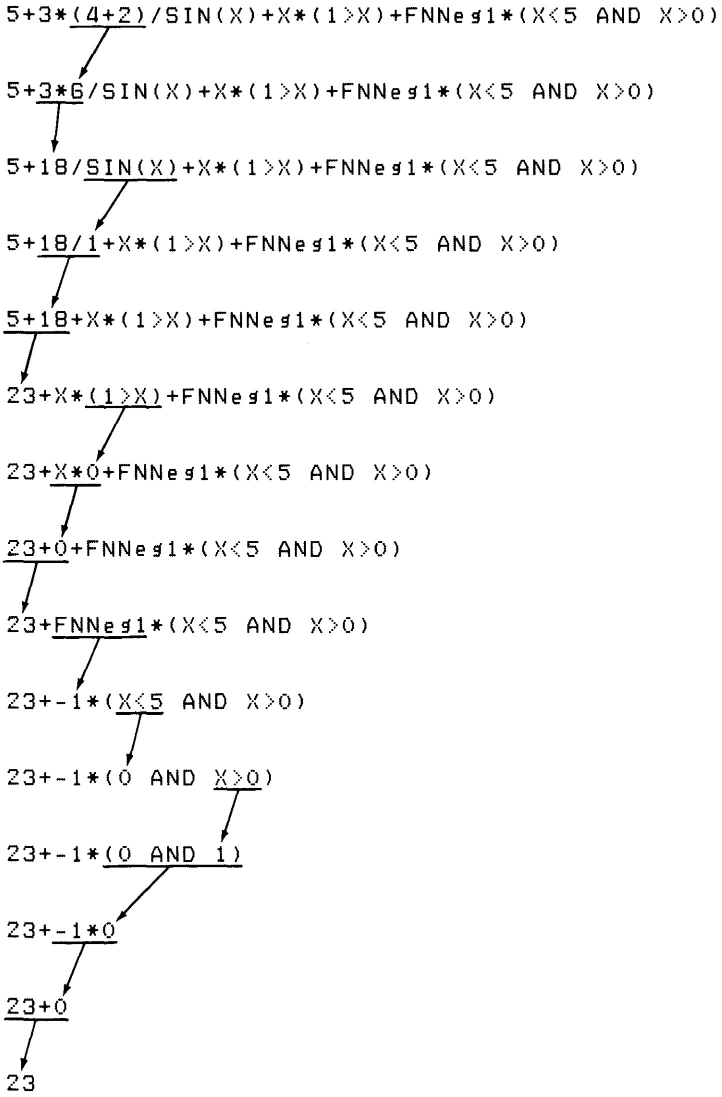
Precedence	Operator
Highest	Parentheses; they may be used to force any order of operation
	Functions, both user-defined and intrinsic
	Exponentiation: ^
	Multiplication and division: * / MOD DIV MODULO
	Addition, subtraction, monadic plus and minus: + −
	Relational Operators: = <> < > <= >=
	NOT
	AND
Lowest	OR, EXOR

When an expression is being evaluated it is read from left to right and operations are performed as encountered, unless a higher precedence operation is found immediately to the right of the operation encountered, or unless the hierarchy is modified by parentheses. If BASIC cannot deal immediately with the operation, it is stacked, and the evaluator continues to read until it encounters an operation it can perform. It is easier to understand if you see an example of how an expression is actually evaluated.

The following expression is complex enough to demonstrate most of what goes on in expression evaluation.

```
A = 5+3*(4+2)/SIN(X)+X*(1>X)+FNNeg1*(X<5 AND X>0)
```

In order to evaluate this expression, it is necessary to have some historical data. We will assume that DEG has been executed, that `X= 90`, and that `FNNeg1` returns `-1`. Evaluation proceeds as follows:



```
5+3*(4+2)/SIN(X)+X*(1>X)+FNNeg1*(X<5 AND X>0)
5+3*6/SIN(X)+X*(1>X)+FNNeg1*(X<5 AND X>0)
5+18/SIN(X)+X*(1>X)+FNNeg1*(X<5 AND X>0)
5+18/1+X*(1>X)+FNNeg1*(X<5 AND X>0)
5+18+X*(1>X)+FNNeg1*(X<5 AND X>0)
23+X*(1>X)+FNNeg1*(X<5 AND X>0)
23+X*0+FNNeg1*(X<5 AND X>0)
23+0+FNNeg1*(X<5 AND X>0)
23+FNNeg1*(X<5 AND X>0)
23+-1*(X<5 AND X>0)
23+-1*(0 AND X>0)
23+-1*(0 AND 1)
23+-1*0
23+0
23
```

The Delayed Assignment Surprise

BASIC delays assigning a value to a variable as long as possible. In the actual evaluation a pointer to the location of a variable is what is stacked. This means that when a variable is in an area of COM accessible to both the main program and a user-defined function is used in an expression that also calls the user-defined function—and is modified in the function—the value of the expression can be surprising, although not unpredictable. For example, if we define a function `FNNeg1` that returns a minus 1, we would expect the following lines to print 2.

```
10 COM X
20 X = 3
30 Y = X + FNNeg1
40 PRINT Y
```

However, if the user-defined function looks like this:

```
1000  DEF FNNeg1
2000     COM X
1020     X = 500
1030     RETURN -1
1040  FNEND
```

The actual result will be 499. Surprising, but not unpredictable. The same thing will happen if the variable is passed by reference and modified in the user-defined function. Therefore, don't use a user-defined function to modify values of variables. They are designed for returning a single value, and are best reserved for that.

Operators

There are three types of operators in BASIC: monadic, dyadic, and relational.

- A **monadic** operator performs its operation on the expression immediately to its right. `+ - NOT`

- A **dyadic** operator performs its operation on the two values it is between. The operators are as follows: `^`, `*`, `/`, `MOD`, `DIV`, `+`, `-`, `=`, `<>`, `<`, `>`, `<=`, `>=`, `AND`, `OR`, and `EXOR`.

- A **relational** operator returns a 1 (true) or a 0 (false) based on the result of a relational test of the operands it separates. The relational operators are a subset of the dyadic operators that are considered to produce *boolean* results. These operators are as follows: `<`, `>`, `<=`, `>=`, `=`, and `<>`.

NOTE

The **only** relational operators allowed with COMPLEX values are: = and <>. The **only** dyadic operators allowed with COMPLEX values are: ^, +, −, *, /, <>, and =. The **only** monadic operators allowed with COMPLEX values are: + and −.

While the use of most operators is obvious from the descriptions in the language reference, some of the operators have uses and side-effects that are not always apparent.

Expressions as Pass Parameters

All numeric expressions are passed by value to subprograms. Thus 5+X is obviously passed by value. Not quite so obviously, +X is also passed by value. The monadic operator makes it an expression.

For more information on pass parameters, read the chapter entitled "Subprograms and User-Defined Functions."

Strings in Numeric Expressions

String expressions can be directly included in numeric expressions if they are separated by relational operators. The relational operators always yield boolean results, and boolean results are numeric values in BASIC. For example:

```
110   Day_number=1*(Day$="Sun")+2*(Day$="Mon")
```

Executing the program line above would result in `Day_number` being equal to 1 if `Day$` equals "`Sun`" and 2 if `Day$` equals `"Mon"` (or 0 otherwise).

Step Functions

The relational operators are obviously useful for conditional branching (IF...THEN statements), but are also valuable for creating numeric expressions representing step-functions. For example, lets try to represent the function:

- If `Select < 0`

  Then `Result = 0`

- If `0 <= Select < 1`

  Then `Result` equals the square root of $A^2 + B^2$.

- If `Select >= 1` (any other value)

  Then `Result = 15`

It is possible to generate the required response through a series of IF...THEN statements, but it can also be done with the following expression:

```
1210 Result=(Select<0)*0+(Select>=0 AND Select<1)*SQR(A^2+B^2)+(Select>1)*15
```

While the technique may not please the purist, it actually represents the step function very well. The boolean expressions each return a 1 or 0 which is then multiplied by the accompanying expression. Expressions not matching the selection return 0, and are not included in the result. The value assigned to `Select` before the expression is evaluated determines the computation placed in the result. This technique can be used to represent other functions, but the program statement cannot exceed the maximum allowable line length.

Comparisons Between Two REAL or COMPLEX Values

If you are comparing INTEGER numbers, no special precautions are necessary since these values are represented *exactly*. However, if you are comparing REAL or COMPLEX values, especially those which are the results of calculations and functions, it is possible to run into problems due to rounding and other limits inherent in the system. For example, consider the use of relational operators in IF..THEN statements to check for equality in any situation resembling the following:

```
1220   DEG
1230   A=25.3765477
1240   IF SIN(A)^2+COS(A)^2=1 THEN
1250     PRINT "Equal"
1260   ELSE
1270     PRINT "Not Equal"
1280   END IF
```

You may find that the equality test fails due to rounding errors or other errors caused by the inherent limitations of finite machines. A repeating decimal or irrational number cannot be represented exactly in any finite machine.

For additional information on rounding errors read the subsequent section entitled "Rounding Functions."

Numerical Functions

Intrinsic functions are the built-in functions that are part of the BASIC language. Numerous functions are included in the BASIC you are using to make mathematical modeling easier. This section covers these functions by placing them in the categories given below. For a list of all the Numerical Functions, see the "Keyword Summary" in the *BASIC Language Reference* and *BASIC Condensed Reference.*

- Arithmetic Functions
- Array Functions
- Exponential Functions
- Trigonometric Functions
- Hyperbolic Functions
- Binary Functions
- Limit Functions
- Rounding Functions
- Random Number Function
- Complex Functions
- Time and Date Functions
- Base Conversion Functions
- General Functions

Arithmetic Functions

Numeric computations at times require you to:

- determine the square root of an expression,
- find the absolute value of an expression,
- return the sign of an expression,
- return the fractional part of an expression,
- return the greatest integer that is less than or equal to an expression.

It is not always convenient to write a program segment or subprogram to perform these numeric operations. To eliminate this inconveniences, BASIC provides you with the following functions:

`ABS` Returns the absolute value of an expression. Takes a REAL, INTEGER, or COMPLEX number as its argument.

`FRACT` Returns the "fractional" part of the argument.

`INT` Returns the greatest integer that is less than or equal to an expression. The result is of the same type (INTEGER or REAL) as the original number.

`MAXREAL` Returns the largest positive REAL number available in BASIC. Its value is approximately 1.797 693 134 862 32E+308.

`MINREAL` Returns the smallest positive REAL number available in BASIC. Its value is approximately 2.225 073 858 507 24E−308.

`SQRT` or `SQR` Return the square root of an expression. Takes a REAL, INTEGER, or COMPLEX number as their argument.

`SGN` Returns the sign of an expression: 1 if positive, 0 if 0, −1 if negative.

Array Functions

These functions are available when the MAT binary is loaded. They return specific information about numeric arrays (e.g., how many dimensions does the array have, the determinant of an array, etc.). For more information on the numeric array functions listed below, read the "Numeric Arrays" chapter.

`BASE` Returns the lower subscript bound of a dimension of an array.

`DET` Returns the determinant of a matrix.

`DOT` Returns the inner (dot) product of two numeric vectors.

`RANK` Returns the number of dimensions in an array.

`SIZE` Returns the number of elements in a dimension of an array.

`SUM` Returns the sum of all the elements in a numeric array.

Exponential Functions

These functions are used for determining the natural and common logarithm of an expression, as well as the Napierian e raised to the power of an expression. Note that all exponential functions take REAL, INTEGER, or COMPLEX numbers as their argument.

`EXP` Raise the Napierian e to an power. $e \approx 2.718\,281\,828\,459\,05$.

`LGT` Returns the base 10 logarithm of an expression.

`LOG` Returns the natural logarithm (Napierian base e) of an expression.

Trigonometric Functions

Six trigonometric functions and the constant π are provided for dealing with angles and angular measure. Note that all trigonometric functions take REAL, INTEGER, or COMPLEX numbers as their argument.

`ACS` Returns the arccosine of an expression.

`ASN` Returns the arcsine of an expression.

`ATN` Returns the arctangent of an expression.

`COS` Returns the cosine of the angle represented by the expression.

`SIN` Returns the sine of the angle represented by an expression.

`TAN` Returns the tangent of the angle represented by an expression.

`PI` Returns the constant $3.141\,592\,653\,589\,79$, an approximate value for π.

Trigonometric Modes: Degrees and Radians

The default mode for all angular measure is radians. Degrees can be selected with the DEG statement. Radians may be re-selected by the RAD statement. It is a good idea to explicitly set a mode for any angular calculations, even if you are using the default (radian) mode. This is especially important in writing subprograms, as the subprogram inherits the angular mode from the context that calls it. The angle mode is part of the calling context. If it is changed in a subprogram, it is restored when the calling context is restored.

Hyperbolic Functions

Six hyperbolic functions are available with the BASIC system when the COMPLEX binary is loaded:

SINH	returns the hyperbolic sine of a number.
COSH	returns the hyperbolic cosine of a number.
TANH	returns the hyperbolic tangent of a number.
ASNH	returns the hyperbolic arcsine of a number.
ACSH	returns the hyperbolic arccosine of a number.
ATNH	returns the hyperbolic arctangent of a number.

Binary Functions

We humans usually think of numbers being represented as decimal numbers, so this is the default representation for most input and output operations (such as INPUT and DISP). However, all operations that BASIC performs use a binary number representation. You usually don't see this, because BASIC changes decimal numbers you input into its own binary representation, performs operations using these binary numbers, and then changes them back to their decimal representation before displaying or printing them.

Here are the functions in BASIC which deal with binary numbers:

BINAND	Returns the bit-by-bit "logical and" of two arguments.
BINCMP	Returns the bit-by-bit "complement" of its argument.
BINEOR	Returns the bit-by-bit "exclusive or" of two arguments.
BINIOR	Returns the bit-by-bit "inclusive or" of two arguments.
BIT	Returns the state of a specified bit of the argument.
ROTATE	Returns a value obtained by shifting an INTEGER representation of an argument a specific number of bit positions, *with* wraparound.
SHIFT	Returns a value obtained by shifting an INTEGER representation of an argument a specific number of bit positions, *without* wraparound.

When any of these functions are used, the arguments are first converted to INTEGER (if they are not already in the correct form) and then the specified operation is performed. It is best to restrict bit-oriented binary operations to declared INTEGERs. If it is necessary to operate on a REAL, make sure the precautions described under "Conversions," at the beginning of this chapter, are employed to avoid INTEGER overflow. Given a COMPLEX argument, the above functions give error 620 (COMPLEX value not allowed).

Limit Functions

It is sometimes necessary to limit the range of values of a variable (as in the discussion of REAL to INTEGER conversions mentioned in the introduction to this chapter). While it is possible to do this with the IF...THEN statements:

```
200 IF X>Maxx THEN X = Maxx
210 IF X<Minx THEN X = Minx
```

it is more convenient to use the MAX and MIN functions (these functions require the MAT binary).

```
200  X = MIN(MAX(X,Minx),Maxx)
```

where:

MAX	Returns a value equal to the greatest value in the list of arguments.
MIN	Returns a value equal to the least value in the list of arguments.

These functions work with INTEGER and REAL values.

Rounding Functions

Rounding occurs frequently in computer operations. The most common rounding occurs in printouts and displays, where it can be handled effectively with a USING clause in the output operation. For details see the section on Formatted Output in the "Using a Printer" chapter. This works in statements such as PRINT, LABEL, OUTPUT, and DISP.

Sometimes it is necessary to round a number in a calculation to eliminate unwanted resolution. There are two basic types of rounding, rounding to a total number of decimal digits and rounding to a number of decimal places (limiting fractional information). Both types of rounding have their own application in programming.

The functions which perform the types of rounding mentioned above are as follows:

DROUND	Rounds a numeric expression to the specified number of digits. If the specified number of digits is greater than 15, no rounding takes place. If the number of digits specified is less than 1, zero is returned.
PROUND	Returns the value of the argument rounded to a specified power of ten.

There is a tendency for the number of decimal places to grow as calculations are performed on the results of other calculations. One of the first things covered in training for engineering and the sciences is how to handle the growth of the number of decimal places in a calculation. If the initial measurements from an experiment produced three digits of information per reading, it is very misleading to produce a seven-digit number as the result of a long series of calculations. The DROUND function allows you to eliminate the unwanted digits, to produce more realistic calculations and answers.

```
X=DROUND (SQR(Y^3+Gr^3),3)
```

It is also possible to round to a number of decimal places. The idea is to eliminate decimal representation beyond a specific power of ten. The PROUND function allows you to perform a round to any specified power of ten.

```
200  X = PROUND (X1, Places)
```

Rounding Errors Resulting from Comparisons

Equality errors occur when multiplying or dividing data values and comparing their result to another non-integer data value. This happens because the product of two non-integer values nearly always results in more digits beyond the decimal point than exists in either of the two numbers being multiplied. Any tests for equality must consider the **exact** variable value to its greatest resolution. If you cannot guarantee that all digits beyond the required resolution are zero, here are three methods that can be used to eliminate equality errors which could occur as a result of this:

- Use the DROUND function to eliminate unwanted resolution **before** comparing results.
- Use the absolute value of the difference between the two values, and test for the difference less than a specified limit.
- Use the absolute value of the relative difference between two values, and test for the difference less than a specified limit:

```
IF ABS((C-F)/C) < 10^(-Delta_power) THEN PRINT "C is equal to F"
```

The following example shows the DROUND technique:

```
1050   A=32.5087
1060   B=31.625
1070   C=A*B           ! Product is 1028.08763750
1080   D=32.5122
1090   E=31.621595509
1100   F=D*E           ! Product is 1028.08763751
1110   IF C=F THEN 1130
1120   PRINT "C is not equal to F"
1130   C=DROUND(C,7)
1140   F=DROUND(F,7)
1150   IF C=F THEN
1160     PRINT "C equals F after DROUND"
1170   ELSE
1180     PRINT "C not equal to F after DROUND"
1190   END IF
1200   END
```

You can experiment with the concept by substituting other values for variables A, B, D, and E, and by changing the number of digits specified in the DROUND function.

Here is an example of the absolute value method of testing equality. In this case, a difference of less than 0.001 is assumed to be evidence of adequate equality. Using the previous example, we change methods starting at line 1130.

```
1130   IF ABS(C-F)<.001 THEN
1140     PRINT "C is equal to F within 0.001"
1150   ELSE
1160     PRINT "C is not equal to F within 0.001"
1170   END IF
1180   END
```

This technique has the advantage that no additional statements are invested in overhead while preparing the data for evaluation. It also enables you to easily establish tolerance limits in making value comparisons, a capability that is useful in production and testing applications.

Finally, here is an example of the relative difference method. Once again, we change methods starting at line 1130.

```
1130   IF ABS((C-F)/C)< 10^(-3) THEN
1140     PRINT "Relative difference between C and F less than 10^-3"
1150   ELSE
1160     PRINT "Relative difference between C and F greater than 10^-3"
1170   END IF
1180   END
```

Random Number Function

Since many modeling systems require random numbers with arbitrary ranges, it is necessary to scale the numbers.

```
200   R= INT(RND*Range)+Offset
```

where:

RND   Returns a pseudo-random number greater than 0 and less than 1.

Note that the above statement will return an integer between **Offset** and **Offset** + **Range**.

The random number generator is seeded with the value 37 480 660 at power-on, SCRATCH, SCRATCH A, and prerun. The pattern period is $2^{31} - 2$. You can change the seed with the RANDOMIZE statement, which will give a new pattern of numbers.

Complex Functions

These functions are available when the COMPLEX binary is loaded. Topics which are covered in this section are:

- Creating COMPLEX Values
- Evaluating COMPLEX Numbers
- COMPLEX Arguments and the Trigonometric Mode
- Determining the Parts of COMPLEX Numbers
- Converting from Rectangular to Polar Coordinates
- An Application for COMPLEX Numbers

Creating COMPLEX Values

The CMPLX function creates a COMPLEX value by using its first argument as the real part and the second argument as the imaginary part of the COMPLEX value. Note that there are no COMPLEX constants in BASIC, but this function provides the same functionality. For example, the following program creates a COMPLEX value and assigns it to the COMPLEX variables C and B. It then displays the results.

```
10  COMPLEX B,C
20  C=CMPLX(3.5,.5)
30  B=C
40  PRINT C,B
50  END
```

Executing the above program produces these results:

```
3.5  .5       3.5  .5
```

Evaluating COMPLEX Numbers

The BASIC expression evaluation uses two separate routines for dealing with REAL, INTEGER, and COMPLEX data types. There is a routine for dealing with REAL and INTEGER numbers and one for COMPLEX numbers. For example, taking the square root of a negative INTEGER or REAL number will produce an error. For instance; executing this statement:

```
SQRT(-1)
```

results in this error:

```
ERROR 30
```

The square root of a COMPLEX value whose real part is negative is defined so the operation is allowed. For example, executing this statement:

```
SQRT(CMPLX(-1,0))
```

returns the value:

```
0   1
```

where 0 is the real part and 1 is the imaginary part of the complex number.

COMPLEX Arguments and the Trigonometric Mode

When a trigonometric function call is made using a COMPLEX value as its parameter, BASIC will evaluate that call using the radian mode regardless of the current trigonometric mode setting (DEG or RAD). After the function call has been evaluated, the system returns to the current trigonometric mode. For example, enter and run this program:

```
10 DEG
20 PRINT SIN(30)
30 PRINT
40 PRINT SIN(CMPLX(30,0)) ! Always evaluated in the RAD mode.
50 PRINT
60 PRINT SIN(30)
70 END
```

The results from executing this program are as follows:

```
 .5              (degree mode)

-.988031624093   0  (radian mode)

 .5              (degree mode)
```

NOTE

Any complex function whose definition includes a sine or cosine function will be evaluated in the radian mode (RAD) regardless of the current angle mode (RAD or DEG).

Determining the Parts of COMPLEX Numbers

In many cases, such as network design, it is useful to be able to determine the real and imaginary parts of complex numbers, and the conjugate of a complex number.

`REAL(C)` returns the real part of a complex number.

`IMAG(C)` returns the imaginary part of a complex number.

`CONJG(C)` returns the complex conjugate of a complex number. That is:

```
CONJG(CMPLX(3,4))
```

is the same as

```
CMPLX(3,-4)
```

For example, executing the following statement:

```
DISP REAL(CMPLX(10,-3))
```

produces this result:

```
10
```

This next statement:

```
DISP IMAG(CMPLX(10,-3))
```

produces:

```
-3
```

This last example:

```
DISP CONJG(CMPLX(10,-3))
```

produces:

```
10  3
```

Converting from Rectangular to Polar Coordinates

BASIC stores and uses complex numbers in a representation called rectangular coordinates. The rectangular coordinate representation of the complex plane is a Cartesian coordinate system where the X axis represents the real part of the complex number and the Y axis represents the imaginary part of the complex number. An alternate representation is polar coordinates. Polar coordinates consist of a magnitude and an argument (angle). The representation for polar coordinates is given as follows:

$$M \angle \theta$$

where M is the magnitude and θ is the argument. The BASIC function used to obtain the magnitude is ABS, and the function used to obtain the argument is ARG.

The following program converts the rectangular coordinates 5 and 6 of the complex number *5 + i6* to polar coordinates.

```
140 RAD
150 DISP "The magnitude of 5 + i6 is: ";ABS(CMPLX(5,6))
160 DISP "The argument of 5 + i6 is: ";ARG(CMPLX(5,6))
170 END
```

Executing this program produces the following:

```
The magnitude of 5 + i6 is: 7.81024967591    (in RAD mode)
The argument of 5 + i6 is: .876058050598
```

An Application for COMPLEX Numbers

An example for the use of COMPLEX numbers is phasors. Phasors are COMPLEX numbers that represent sinusoidal waves. A phasor has both amplitude and phase and can be represented in the following forms:

Rectangular

$$M(\cos\theta + i\sin\theta)$$

Exponential

$$Me^{i\theta}$$

Polar

$$M\angle\theta$$

The figure below shows the phasor C in the complex plane where θ is the phase angle, C is the magnitude, y is the imaginary part of the complex number, and x is the real part of the complex number.

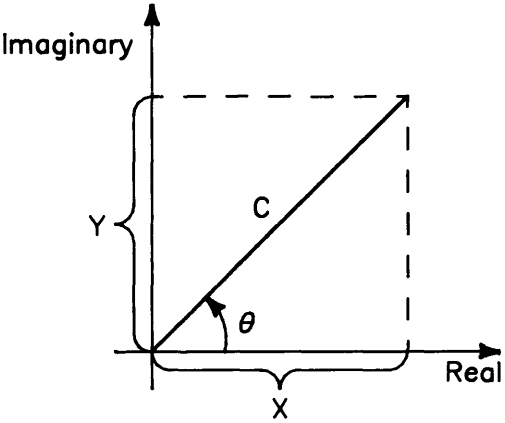


Figure 4-3. A Phasor in the Complex Plane

Given two phasors `Phasor_1` and `Phasor_2`, determine their sum using rectangular coordinates and call this sum `Phasor`. The values of the phasors are:

```
Phasor_1 =
```

$$10 \angle 30^\circ$$

```
Phasor_2 =
```

$$5 \angle 30^\circ$$

In the program, we will need to covert these polar coordinates to rectangular coordinates and add them. The program provided below will do this. Note that the magnitude for Phasor_1 is 10 and its phase angle is 30°, and the magnitude for Phasor_2 is 5 and its phase angle is 30°. You will need to enter these values as the program requests them.

```
100   COMPLEX Phasor,Phasor_1,Phasor_2
110   REAL Mag_1,Mag_2,Phase_1,Phase_2
120   !
130   DEG ! Select degrees as the unit of measure for angles.
140   !
150   INPUT "Enter the magnitude and phase angle for Phasor_1.",Mag_1,Phase_1
160   INPUT "Enter the magnitude and phase angle for Phasor_2.",Mag_2,Phase_2
170   !
180   Phasor_1=CMPLX(Mag_1*COS(Phase_1),Mag_1*SIN(Phase_1)) ! Create Phasor_1.
190   Phasor_2=CMPLX(Mag_2*COS(Phase_2),Mag_2*SIN(Phase_2)) ! Create Phasor_2.
200   !
210   Phasor=Phasor_1+Phasor_2 ! Add both phasors.
220   !
230   DISP "The sum of Phasor_1 and Phasor_2 = ";ABS(Phasor);ARG(Phasor)
240   END
```

The result of executing the above program is:

```
The sum of Phasor_1 and Phasor_2 =  15  30
```

Time and Date Functions

The following functions return the time and date in seconds:

`TIMEDATE` Returns the current clock value (in Julian seconds).

(If there is no battery-backed clock, the clock value set at power-on is 2.086 629 12E+11, which represents midnight March 1, 1900. If the computer is connected to an SRM system, the SRM clock's value is read from the SRM System's clock when the SRM binary is loaded.)

The time value accumulates from its initial value unless it is changed by SET TIME or SET TIMEDATE. For example, executing this function

```
TIMEDATE
```

returns a value in seconds similar to the following:

```
2.11404868285E+11
```

`TIME` converts a formatted time-of-day string into a numeric value of seconds passed midnight. For example, executing this statement:

```
TIME("8:37:30")
```

returns the following numeric value in seconds:

```
31050
```

`DATE` converts a formatted date string into a numeric value in seconds. For example, executing this statement:

```
DATE("26 OCT 1986")
```

returns the following numeric value in seconds:

```
2.11397472E+11
```

For more information on this subject read the chapter in this manual entitled "The Real-Time Clock." Also included in this chapter are the DATE$ and TIME$ string functions.

Base Conversion Functions

The two functions IVAL and DVAL convert a binary, octal, decimal, or hexadecimal string value into a decimal number.

`IVAL` returns the INTEGER value of a binary, octal, decimal, or hexadecimal 16-bit integer. The first argument is a string and the second argument is the radix or base to convert from. For example, executing this statement

```
IVAL("12740",8)
```

returns the following numeric value:

```
5600
```

`DVAL` returns the decimal whole number value of a binary, octal, decimal, or hexadecimal 32-bit integer. The first argument is a string and the second argument is the radix or base to convert from. For example, executing this statement

```
DVAL("1111111111111111111111111111100",2)
```

returns the following numeric value:

```
-4
```

For more information and examples of these functions, read the section "Number-Base Conversion" found in the "String Manipulation" chapter.

General Functions

When you are specifying select code and device selector numbers, it is more descriptive to use a function, such as KBD (returns the select code of the keyboard), to represent that device as opposed to a numeric value. For example, the following command allows you to enter a numeric value from the keyboard.

```
ENTER 2;Numeric_value
```

The above statement used in a program is not as easy to read as this one is:

```
ENTER KBD;Numeric_value
```

where you know the function KBD must stand for keyboard. Therefore, you know the statement is asking you to enter a numeric value from the keyboard.

Functions which return a select code or device selector are listed below:

CRT Returns the INTEGER 1. This is the select code of the internal CRT.

KBD Returns the INTEGER 2. This is the select code of the keyboard.

PRT Returns the INTEGER 701. This is the default (factory set) device selector for an external printer (connected through the built-in HP-IB interface at select code 7).

SC Returns the interface select code associated with an I/O path name.

Another function which fits in the general function category is the RES function. This function returns the last live keyboard numeric result (same as RECALL key).

Numeric Arrays 4

Numeric Arrays 4

An array is a multi-dimensioned structure of variables that are given a common name. The array can have one through six dimensions. Each location in an array can contain one variable value, and each value has the characteristics of a single variable, depending on whether the array consists of `REAL`, `INTEGER` or `COMPLEX` values (string arrays are discussed in the chapter, "String Manipulation.") Note that many of the statements that deal with arrays (such as `MAT`) require the `MAT` binary.

A one-dimensional array consists of *n* elements, each identified by a single subscript. A two-dimensional array consists of *m* times *n* elements where *m* and *n* are the maximum number of elements in the two respective dimensions. Arrays require a subscript in each dimension, in order to locate a given element of the array. Up to six dimensions can be specified for any array in a program. `REAL` arrays require eight bytes of memory for each element, plus overhead, and `COMPLEX` arrays require 16 bytes of memory for each element, plus overhead. It is easy to see that large arrays can demand massive memory resources.

An undeclared array is given as many dimensions as it has subscripts in its lowest-numbered occurrence. Each dimension of an undeclared array has an upper bound of ten. Space for these elements is reserved whether you use them or not.

Dimensioning an Array

Before you use an array, you should tell the system how much memory to reserve for it. This is called "dimensioning" an array. You can dimension arrays with the DIM, COM, ALLOCATE, INTEGER, REAL or COMPLEX statements. For example:

```
COMPLEX Array_complex(2,4)
```

An array is a type of variable and as such follows all rules for variable names. Unless you explicitly specify INTEGER or COMPLEX type in the dimensioning statement, arrays default to REAL type. The same array can only be dimensioned once in a context[1]. However, as we explain later in this section, arrays can be REDIMensioned.

When you dimension an array, the system reserves space in internal memory for it. The system also sets up a table which it uses to locate each element in the array. The location of each element is designated by a unique combination of subscripts, one subscript for each dimension. For a two-dimensional array, for instance, each element is identified by two subscript values. An example of dimensioning a two-dimensional array is as follows:

```
DIM Array(3,5)
```

Each unique set of subscript values points to one, and only one, array element. For example, assuming an OPTION BASE of 1, to indicate the location of the 3rd element in the 2nd row of the above array, you would use the following subscript values:

```
Array(2,3)
```

The actual size of an array is governed by the number of dimensions and the subscript range of each dimension. If **A** is a three-dimensional array with a subscript range of 1 thru 4 for each dimension,

```
DIM A(1:4,1:4,1:4)
```

then its size is 4×4×4, 64 elements. Note that 1 on the left side of the colon in the dimension statement above is the lower bound and 4 on the right is the upper bound.

[1] There is one exception to this rule: If you **ALLOCATE** an array, and then **DEALLOCATE** it, you can dimension the array again.

When you dimension an array, therefore, you must give not only the number of dimensions but also the subscript range of each dimension. Subscript ranges can be specified by giving the lower and upper bounds, as shown above, or by giving just the upper bound. If you give only the upper bound, the lower bound defaults to the current option base setting.

Each context initializes to an option base of 0 (but arrays appearing in COM statements with an (*) will keep the base with which they were originally dimensioned). However, you can set the option base to 1 using the `OPTION BASE` statement. You can have only one `OPTION BASE` statement in a context, and it must precede all explicit variable declarations.

Some Examples of Arrays

The following examples illustrate some of the flexibility you have in dimensioning arrays.

```
10  OPTION BASE 1
20  DIM A(3,4,0:2)
```

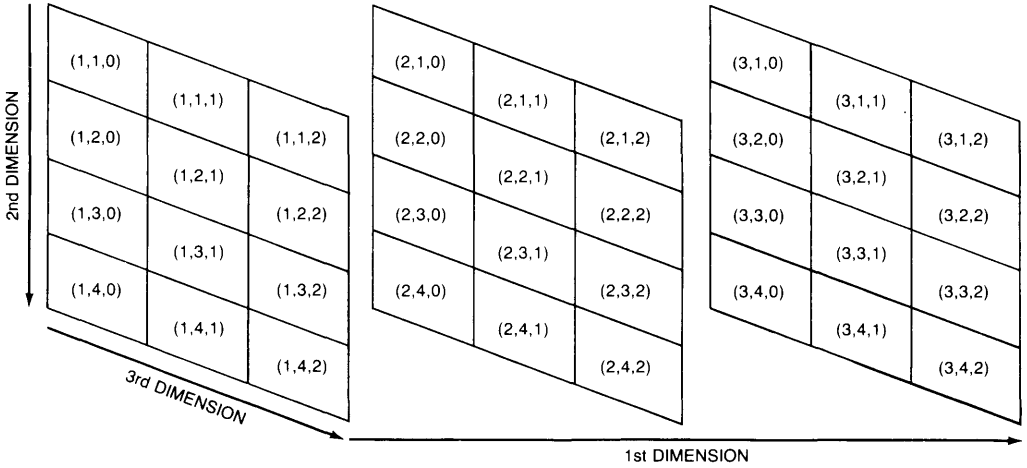


Figure 5-1. Planes of a Three-Dimensional REAL Array

Dimension	Size	Lower Bound	Upper Bound
1st	3	1	3
2nd	4	1	4
3rd	3	0	2

In this example we portray the first dimension as planes, the second dimension as rows, and the third dimension as columns. In general, the last two dimensions of any array always refer to rows and columns, respectively. When we discuss two-dimensional arrays, the first dimension will always represent rows, and the second dimension will always represent columns. Note also in the above example that the first two dimensions use the default setting of 1 for the lower bound, while the third dimension explicitly defines 0 as the lower bound. The numbers in parentheses are the subscript values for the particular elements. These are the numbers you use to identify each array element.

```
10  OPTION BASE 1
20  COM B(1:5,2:6)
        .
        .
```

(1,2)	(1,3)	(1,4)	(1,5)	(1,6)
(2,2)	(2,3)	(2,4)	(2,5)	(2,6)
(3,2)	(3,3)	(3,4)	(3,5)	(3,6)
(4,2)	(4,3)	(4,4)	(4,5)	(4,6)
(5,2)	(5,3)	(5,4)	(5,5)	(5,6)

Figure 5-2. Two-Dimensional REAL Array

Dimension	Size	Lower Bound	Upper Bound
1st	5	1	5
2nd	5	2	6

```
10   OPTION BASE 1
20   ALLOCATE INTEGER C(2:4,-2:2)
       .
       .
```

(2, −2)	(2, −1)	(2,0)	(2,1)	(2,2)
(3, −2)	(3, −1)	(3,0)	(3,1)	(3,2)
(4, −2)	(4, −1)	(4,0)	(4,1)	(4,2)

Figure 5-3. A Dynamically Allocated, Two-Dimensional INTEGER Array

Dimension	Size	Lower Bound	Upper Bound
1st	3	2	4
2nd	5	−2	2

```
10   OPTION BASE 0
20   COMPLEX D(1,0)
     .
     .
```

(0,0)	(0,1)
(1,0)	(1,1)

Figure 5-4. A Two-Dimensional COMPLEX Array

Dimension	Size	Lower Bound	Upper Bound
1st	2	0	1
2nd	2	0	1

```
10    COM COMPLEX E(-3:0)
      .
      .
```

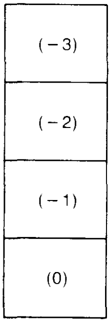


Figure 5-5. A One-Dimensional COMPLEX Array in Common

Dimension	Size	Lower Bound	Upper Bound
1st	4	−3	0

```
10 OPTION BASE 0
20 INTEGER F(1,4,-1:2)
      .
      .
```

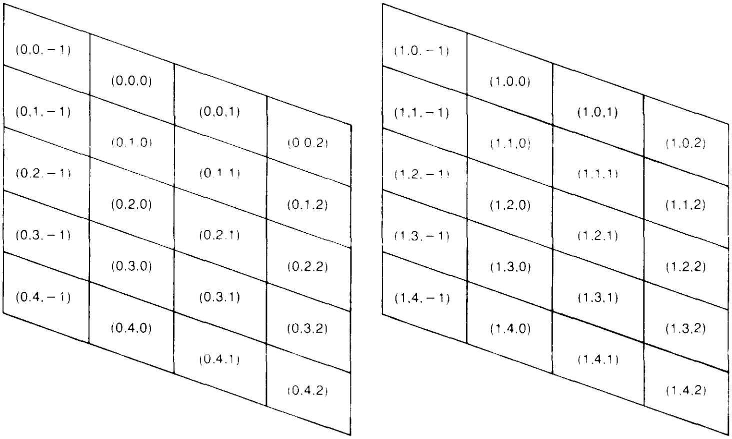


Figure 5-6. A Three-Dimensional INTEGER Array

Dimension	Size	Lower Bound	Upper Bound
1st	2	0	1
2nd	5	0	4
3rd	4	−1	2

Arrays are limited to six dimensions, and the subscript range for each dimension must lie between -32767 and 32767. (`REDIM` and `ALLOCATE` allow the subscript range to go down to -32768, but the total size of each dimension must be less than 32768 elements.) For the most part, we use only two-dimensional examples since they are easier to illustrate. However, the same principles apply to arrays of more than two dimensions as well.

Note

Throughout this chapter we will be using `DIM` statements without specifying what the current option base setting is. Unless explicitly specified otherwise, all examples in this chapter use option base 1.

As an example of a four-dimensional array, consider a five-story library. On each floor there are 20 stacks, each stack contains 10 shelves, and each shelf holds 100 books. To specify the location of a particular book you would give the number of the floor, the stack, the shelf, and the particular book on that shelf. We could dimension an array for the library with the statement:

```
DIM Library(5,20,10,100)
```

This means that there are 100,000 book locations. To identify a particular book you would specify its subscripts. For instance, **Library(2,12,3,35)** would identify the 35th book on the 3rd shelf of the 12th stack on the 2nd floor.

We can imagine accessing a particular page of a book by using a 5-dimensional array. For instance, if we dimension an array,

```
DIM Page(5,20,10,100,200)
```

then **Page(1,7,2,19,130)** would designate page 130 of the 19th book on the 2nd shelf of the 7th stack on the 1st floor.

We could specify words on pages by using a 6-dimensional array. Six dimensions is the maximum, though, so we could not specify letters of words.

Also, you can dimension more than one array in a single statement by separating the declarations with a comma. For instance,

```
10  DIM A(1,3,4),B(-2:0,2:5),C(5)
```

would dimension all three arrays: **A**, **B**, and **C**.

Problems with Implicit Dimensioning

In any environment, an array must have a dimensioned size. This size can be passed into an environment through a passed parameter list or a COM statement. It may be explicitly dimensioned through COM, INTEGER, REAL, COMPLEX or ALLOCATE. It can also be implicitly dimensioned through a subscripted reference to it in a program statement **other than** a MAT or a REDIM statement. An attempt to use an array that does not have a dimensioned size in the current environment in a MAT or REDIM statement will result in an error. In other words, MAT and REDIM statements cannot be used to implicitly dimension an array.

Finding Out the Dimensions of an Array

There are a number of statements that allow you to determine the size and shape of an array. To find out how many dimensions are in an array, use the RANK function. For instanced:

```
    OPTION BASE 0
    DIM F(1,4,-1:2)
    PRINT RANK (F)
```

would print 3.

The `SIZE` function returns the size (number of elements) of a particular dimension. For instance,

```
   SIZE (F,2)
```

would return 5, the number of elements in **F**'s second dimension.

To find out what the lower bound of a dimension is, use the `BASE` function. Referring again to array **F**,

```
   BASE (F,1)
```

would return a 0, while,

```
   BASE (F,3)
```

would return a -1.

By using the `SIZE` and `BASE` functions together, you can determine the upper bounds of any dimension (e.g., `SIZE+BASE-1=Upper Bound`).

It may seem pointless to have all these functions that return the dimension specifications which you yourself assigned. After all, if you assigned the dimensions, you should know what they are; and if you forget, you can always look at the appropriate dimensioning statement. However, these functions are powerful tools for writing programs that perform functions on an array regardless of the array's size or shape. In addition, the system automatically redimensions arrays during certain operations. The functions discussed above provide you with a means for determining the new dimensions. The section in this chapter entitled "Examples of Formatting Arrays for Display" provides examples of some general purpose subprograms utilizing the statements covered in this section.

Using Individual Array Elements

This section deals with assigning and extracting individual elements from arrays.

Assigning an Individual Array Element

Once an array has been dimensioned, the next step is to fill it with useful values. Initially, every element in an array equals zero. There are a number of different ways to change these values. The most obvious is to assign a particular value to each element. This is done by specifying the element's subscripts. For instance, the statement,

```
A(3,4)=13
```

would assign the value 13 to the element in the third row and fourth column of **A**. You must give enough subscripts for the system to identify a single element. For a three-dimensional array, for instance, you would provide three subscripts. All subscripts, moreover, must lie within the dimensioned range. If you use out-of-range subscripts, the system returns an error.

Extracting Single Values From Arrays

As with entering values into arrays, there are a number of ways to extract values as well. To extract the value of a particular element, simply specify the element's subscripts. For instance, the statement,

```
X=A(3,4,2)
```

would assign the value of the element occupying the given location in **A** to the variable X. The system will automatically convert variable types. For example, if you assign an element from a COMPLEX array to an INTEGER variable, the system will perform the necessary rounding and ignore the imaginary part of the COMPLEX number.

Filling Arrays

This section will provide you with three methods for filling an entire array. The topics covered are as follows:

- Assigning Every Element in an Array the Same Value
- Using the READ Statement to Fill an Entire Array
- Copying Arrays into Other Arrays

Assigning Every Element in an Array the Same Value

For some applications, you may want to initialize every element in an array to some particular value. You can do this by assigning a value to the array name. However, you must precede the assignment with the MAT keyword. For example,

```
MAT A= (10)
```

will assign the value 10 to every element in array **A**, regardless of **A**'s size. Note that the numeric expression on the right-hand side of the assignment must be enclosed in parentheses and that this expression may be **INTEGER**, **REAL** or **COMPLEX**. Let's look at an example of assigning a **COMPLEX** value to every element of a **COMPLEX** array,

```
MAT C= (CMPLX(1,2))
```

This statements assigns the complex number *1 + i2* to every element of the complex array C.

Using the READ Statement to Fill an Entire Array

You can assign values to an array by using the **READ** and **DATA** statements. The **DATA** statement allows you to create a stream of data items, and the **READ** statement enables you to enter the data stream into an array. For example:

```
100   OPTION BASE 1
110   DIM A(3,3)
120   DATA -4,36,2.3,5,89,17,-6,-12,42
130   READ A(*)
140   PRINT USING "3(3DD.DD,3DD.DD,3DD.DD,/)";A(*)
150   END
```

The asterisk in line 140 is used to designate the entire array rather than a single element. Note also that the right-most subscript varies fastest. In this case, it means that the system fills an entire row before going to the next one. The **READ/DATA** statements are discussed further in the chapter "Data Storage and Retrieval".

Executing the above program, produces the following results:

```
-4.00    36.00     2.30
 5.00    89.00    17.00
-6.00   -12.00    42.00
```

Copying Entire Arrays into Other Arrays

Another way to fill an array is to copy all elements from one array into another[1]. Suppose, for example, that you have the two arrays **A** and **B** shown below.

$$\begin{array}{cc} \mathbf{A} & \mathbf{B} \\ \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} & \begin{pmatrix} 3 & 5 \\ 8 & 2 \\ 1 & 7 \end{pmatrix} \end{array}$$

Note that **A** is a 3×3 array which is filled entirely with 0's, while **B** is a 3×2 array filled with non-zero values. To copy **B** to **A**, we would execute:

```
MAT A= B
```

Again, you must precede the assignment with MAT. The system will automatically redimension the "result array" (the one on the left-hand side of the assignment) so that it is the same size as the "operand array" (the one on the right side of the equation.) There are two restrictions on redimensioning an array.

- The two arrays must have the same rank (e.g., the same number of dimensions.)
- The dimensioned size of the result array must be at least as large as the current size of the operand array.

If the system cannot redimension the result array to the proper size, it returns an error.

[1] Copying sub-sets of arrays is discussed in the subsequent section called "Copying Subarrays".

Automatic redimensioning of an array will not affect the lower bounds, only the upper bounds. So the `BASE` values of each dimension of the result array will remain the same. Also keep in mind that the size restriction applies to the *dimensioned* size of the result array and the *current* size of the operand array. Suppose we dimension arrays **A**, **B** and **C** to the following sizes:

```
10    OPTION BASE 1
20    DIM A(3,3),B(2,2),C(2,4)
      .
      .
```

We can execute,

```
MAT A= B
```

since **A** is dimensioned to 9 elements and **B** is only 4 elements. The copy automatically redimensions **A** to a 2×2 array. Nevertheless, we can still execute:

```
MAT A= C
```

This works because the nine elements originally reserved for **A** remain available until the program is scratched. **A** now becomes a 2×4 matrix. After `MAT A= C`, we could not execute:

```
MAT B= A
```

or

```
MAT B= C
```

since in each of these cases, we are trying to copy a larger array into a smaller one. But we could execute

```
MAT C= A
```

after the original `MAT A= B` assignment, since **C**'s dimensioned size (8) is larger than **A**'s current size (4).

Printing Arrays

Once an array has been filled with values, it is nice to know what those values are. The best way to do this is to display them on the screen or printer. This section provides information on how to perform this task for REAL, INTEGER and COMPLEX values.

Printing an Entire Array

Certain operations (e.g., PRINT, OUTPUT, ENTER and READ) allow you to access all elements of an array merely by using an asterisk in place of the subscript list. The statement,

```
PRINT A(*);
```

would display every element of **A** on the current PRINTER IS device. The elements are displayed in order, with the rightmost subscripts varying fastest. The semi-colon at the end of the statement is equivalent to putting a semi-colon between each element. When they are displayed, therefore, they will be separated by a space. (The default is to place elements in successive columns.)

Examples of Formatting Arrays for Display

This section provides two subprograms which have both been given the name Printmat. The first subprogram is used to display a two-dimensional INTEGER array and the second subprogram is used to display a three-dimensional INTEGER array.

To display a two dimensional array, you can use the following subprogram:

```
240   SUB Printmat(INTEGER Array(*))
250   OPTION BASE 1
260   FOR Row=BASE(Array,1) TO SIZE(Array,1)+BASE(Array,1)-1
270     FOR Column=BASE(Array,2) TO SIZE(Array,2)+BASE(Array,2)-1
280       PRINT USING "DDDD,XX,#";Array(Row,Column)
290     NEXT Column
300     PRINT
310   NEXT Row
320   SUBEND
```

Assuming that the array you intended to display is a 5×5 array, your results should look similar to this:

```
11     12     13     14     15
21     22     23     24     25
31     32     33     34     35
41     42     43     44     45
51     52     53     54     55
```

If you were to expand the above subprogram to print three-dimensional INTEGER arrays, your subprogram would be similar to the following:

```
250   SUB Printmat(INTEGER Array(*))
260     OPTION BASE 1
270     FOR Zplane=BASE(Array,3) TO SIZE(Array,3)+BASE(Array,3)-1
280       PRINT TAB(6),"Plane ";Zplane
290       PRINT
300       FOR Yplane=BASE(Array,2) TO SIZE(Array,2)+BASE(Array,2)-1
310         FOR Xplane=BASE(Array,1) TO SIZE(Array,1)+BASE(Array,1)-1
320           PRINT USING "DDDD,XX,#";Array(Zplane,Yplane,Xplane)
330         NEXT Xplane
340         PRINT
350       NEXT Yplane
360       PRINT
370     NEXT Zplane
380   SUBEND
```

This subprogram displays a three-dimensional array as three subarrays called "planes". As the subarrays are being displayed by the subprogram, each subarray is given a "plane" number which represents a "plane" in the first dimension.

If you had a three dimensional array with the following dimensions:

```
DIM Array1(3,3,3)
```

filled with all 3's, the results from executing the above subprogram would be as follows:

```
  Plane 1

3     3     3
3     3     3
3     3     3

  Plane 2

3     3     3
3     3     3
3     3     3

  Plane 3

3     3     3
3     3     3
3     3     3
```

Passing Entire Arrays

The asterisk is also used to pass an array as a parameter to a function or subprogram. For instance, to pass an array **A** to the `Printmat` subprogram listed earlier, we would write:

```
Printmat (A(*))
```

Copying Subarrays

An earlier section discussed copying the contents of an entire array into another entire array.

```
MAT Array55= Array33
```

Each element of `Array33` is copied into the corresponding element of `Array55` which is redimensioned if necessary.

Now suppose you would like to copy a portion of one array and place it in a special location within another array. This process is called copying subarrays.

$$\overset{\textbf{Array4x4}}{\begin{pmatrix} 11 & 12 & 13 & 14 \\ 21 & -9 & 16 & 24 \\ 31 & 91 & -5 & 34 \\ 41 & 42 & 43 & 44 \end{pmatrix}} \Leftarrow \overset{\textbf{Array3x4}}{\begin{pmatrix} 45 & 67 & -8 & 1 \\ -4 & -9 & 16 & 2 \\ 99 & 91 & -5 & 9 \end{pmatrix}}$$

Figure 5-7. Copying a Subarray into Another Subarray

Topics discussed in this section are:

- Subarray Specifier
- Copying a Subarray into an Array
- Copying an Array into a Subarray
- Copying a Subarray into a Subarray
- Copying a Portion of an Array into Itself
- Rules for Copying Subarrays

Dimensions for the arrays covered in the above topics will assume an option base of 1 (`OPTION BASE 1`) unless stated differently.

Subarray Specifier

A subarray is a subset of an array (an array within an array). A subarray is indicated after the array name as follows:

`Array_name(`*`subarray_specifier`***`)`**

`String_array$(`*`subarray_specifier`***`)`**

The above subarray could take on many "sizes" and "shapes" depending on what you used as dimensions for the array and the values used in the *subarray_specifier*. Note that "size" refers to the number of elements in the subarray and "shape" refers to the number of dimensions and elements in each dimension, respectively [e.g. both of these subscript specifiers have the same shape: `(-2:1,-1:10)` and `(1:4,9:20)`]. Before looking at ways you can express a subarray lets learn a few terms related to the subarray specifier.

subscript range is used to specify a set of elements starting with a beginning element position and ending with a final element position. For example, `5:8` represents a range of four elements starting with element 5 and ending at element 8.

subscript expression is an expression which reduces the `RANK` of the subarray. For example if you wanted to select a one-element subarray from a two-dimensional array which is located in the 2nd row and 3rd column, you would use the following subarray specifier: (2,3:3). The subscript expression in this subarray specifier is 2 which restricts the subarray to row 2 of the array.

default range is denoted by an asterisk (i.e. `(1,*)`) and represents all of the elements in a dimension from the dimension's lower bound to its upper bound. For example, suppose you wanted to copy the entire first column of a two dimensional array, you would use the following subarray specifier: `(*,1)`, where `*` represents all the rows in the array and `1` represents **only** the first column.

Some examples of subarray specifiers are as follows:

`(1,*)` a subscript expression and a default range which designate the first row of a two-dimensional array.

`(1:2)` a given subscript range which represents the first two elements of a one-dimensional array.

`(*,-1:2)` a default range and subscript range which represents all of the elements in the first four columns of a two-dimensional array (base of 2nd dimension assumed to be -1).

`(3,1:2)` a subscript expression and subscript range which represent the first two elements in the third row of a two-dimensional array.

`(1,*,*)` a subscript expression and two default ranges which represent a plane consisting of all the rows and columns of the first plane in the first-dimension.

`(1,1:2,*)` a subscript expression, subscript range and default range which represent the first two rows in the first plane of the first-dimension.

`(1,2,*)` two subscript expressions and a default range which represent the entire second row in the first plane of the first-dimension.

`(1:2,3:4)` two subscript ranges which represent elements located in the third and fourth columns of the first and second rows of a two-dimensional array.

For more information on string arrays, see the "String Manipulation" chapter found in this manual.

Copying an Array into a Subarray

In order to copy a source array into a subarray of a destination array, the destination array's subarray must have the same size and shape as the source array.

A destination and source array are dimensioned as follows:

```
100  OPTION BASE 1
110  DIM Des_array(-3:1,5),Sor_array(2,3)
```

Suppose these arrays contain the following `INTEGER` values:

Des_array **Sor_array**

$$\begin{pmatrix} 11 & 12 & 13 & 14 & 15 \\ 21 & 22 & 23 & 24 & 25 \\ 31 & 32 & 33 & 34 & 35 \\ 41 & 42 & 43 & 44 & 45 \\ 51 & 52 & 53 & 54 & 55 \end{pmatrix} \qquad \begin{pmatrix} 11 & 12 & 13 \\ 21 & 22 & 23 \end{pmatrix}$$

you would copy the source array (`Sor_array`) into a subarray of the destination array (`Des_array`) by using program **line 190** given below:

```
190   MAT Des_array(-1:0,2:4)= Sor_array
```

`Des_array` would have the following values in it as the result of executing the above statement.

Des_array

$$\begin{pmatrix} 11 & 12 & 13 & 14 & 15 \\ 21 & 22 & 23 & 24 & 25 \\ 31 & 11 & 12 & 13 & 35 \\ 41 & 21 & 22 & 23 & 45 \\ 51 & 52 & 53 & 54 & 55 \end{pmatrix}$$

Copying a Subarray into an Array

A subarray can be copied into an array as long as the array can be re-dimensioned to be the size and shape of the subarray specifier.

A destination and source array are dimensioned as follows:

```
100  OPTION BASE 1
110  DIM Des_array(8),Sor_array(-5:4)
```

Suppose both of these one-dimensional arrays contain the following values:

$$\begin{array}{cc} \textbf{Des\_array} & \textbf{Sor\_array} \\ \begin{pmatrix} -1 & 14 & 8 & 4 & 98 & 43 & 90 & -3 \end{pmatrix} & \begin{pmatrix} -11 & -4 & 1 & 2 & 3 & 4 & 78 & 100 & 8 & 18 \end{pmatrix} \end{array}$$

you would copy a subarray of the source array (`Sor_array`) into a destination array (`Des_array`) by using program **line 190** given below:

```
190  MAT Des_array= Sor_array(-4:1)
```

`Des_array` will be re-dimensioned to have 6 elements with the following values in it as a result of executing the above statement.

$$\begin{array}{c} \textbf{Des\_array} \\ \begin{pmatrix} -4 & 1 & 2 & 3 & 4 & 78 \end{pmatrix} \end{array}$$

Copying a Subarray into another Subarray

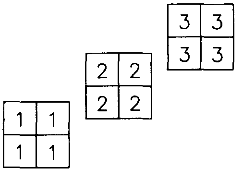
Subarray specifiers must have the same size and shape when you are copying one subarray into another.

A destination and source array are dimensioned as follows:

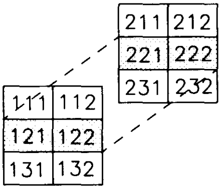
```
100   OPTION BASE 1
110   DIM Des_array(3,2,2),Sor_array(2,3,2)
120   .
130   .
```

Suppose these three dimensional arrays contain the following values:

Des_array



Sor_array

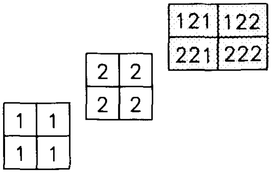


in order to properly copy a source subarray (`Sor_array(*,2,*)`) into a destination subarray using asterisks to represent the ranges of dimensions, you would use **line 190** given below:

```
190   MAT Des_array(3,*,*)= Sor_array(*,2,*)
```

A three dimensional array with the following values in it would be the result of executing the above statement.

Des_array



Copying a Portion of an Array into Itself

If you are going to copy a subarray of an array into another portion of the same array, the two subarray locations should not overlap (e.g., `MAT Array(2:4,1:3)= Array(1:3,2:4)` is an improper assignment). No error message will result from this misuse, but the result is undefined.

A destination and source array are dimensioned as follows:

```
100   OPTION BASE 1
110   DIM Array(4,4)
```

Suppose this two dimensional array contains the following values:

Array

$$\begin{pmatrix} 11 & 12 & 13 & 14 \\ 21 & 22 & 23 & 24 \\ 31 & 32 & 33 & 34 \\ 41 & 42 & 33 & 44 \end{pmatrix}$$

to copy a slice of this array into another portion of the same array, you would use program **line 190** given below:

```
190   MAT Array(3:4,1:2)= Array(1:2,3:4)
```

Array will have the following values in it as a result of executing the above statement.

Array

$$\begin{pmatrix} 11 & 12 & 13 & 14 \\ 21 & 22 & 23 & 24 \\ 13 & 14 & 33 & 34 \\ 23 & 24 & 43 & 44 \end{pmatrix}$$

Note that you **cannot** copy a subarray into the array it is part of with an implied redimensioning of the array. A statement of the form:

```
Array= Array(subarray_specifier)
```

will always generate a run-time error.

Rules for Copying Subarrays

This section should help limit the number of syntax and runtime errors you could make when copying subarrays. A previous section entitled "Subarray Specifier" provided you with examples of the correct way of writing subarray specifiers for copying subarrays. In this section, you will be given rules to things you should not do when copying subarrays. The rules are as follows:

- Subarray specifiers **must not** contain all subscript expressions (i.e. `(1,2,3)` is not allowed, it will produce a syntax error). This rule applies to all subscript specifiers.
- Subarray specifiers **must not** contain all asterisks (*) or default ranges (i.e. `(*,*,*)` is not allowed, it will produce a syntax error). This rule applies to all subscript specifiers.
- If two subarrays are given in a `MAT` statement, there **must be** the same number of ranges in each subarray specifier. For example:

  ```
  MAT Des_array1(1:10,2:3)= Sor_array(5:14,*,3)
  ```

  is the correct way of copying a subarray into another subarray provided the default range given in the source array (`Sor_array`) has only two elements in it. Note that the source array is a three-dimensional array. However, it still meets the criteria of having the same number of ranges as the destination array because two of its entries are ranges and one is an expression.
- If two subarrays are given in a `MAT` statement, the subscript ranges in the source array **must be** the same shape as the subscript ranges in the destination array. For example, the following example is **legal**:

  ```
  MAT Des_array(1:5,0:1)= Sor_array(3,1:5,6:7)
  ```

  however, the following example is **not legal**:

  ```
  MAT Des_array(0:1,1:5)= Sor_array(1:5,0:1)
  ```

  because both of its subarray specifiers do not have the same shape (i.e. the rows and columns in the destination array do not match the rows and columns in the source array).

Redimensioning Arrays

In our discussion of copying arrays we saw that the system automatically redimensions an array if necessary. BASIC also allows you to explicitly redimension an array with the `REDIM` statement. As with automatic redimensioning, the following two rules apply to all `REDIM` statements:

- A `REDIM`ed array must maintain the same number of dimensions.
- You cannot `REDIM` an array so that it contains more elements than it was originally dimensioned to hold.

Suppose **A** is the 3×3 array shown below.

$$\begin{array}{c} \mathbf{A} \\ \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \end{array}$$

To redimension it to a 2×4 array, you would execute:

```
REDIM A(2,4)
```

The new array now looks like the figure below:

$$\begin{array}{c} \mathbf{A} \\ \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{pmatrix} \end{array}$$

Note that it retains the values of the elements, though not necessarily in the same locations. For instance, **A**(2,1) in the original array was 4, whereas in the redimensioned array it equals 5. For example, if we `REDIM`ed **A** again, this time to a 2×2 array, we would get:

```
REDIM A(0:1,0:1)
```

$$\begin{array}{c} \mathbf{A} \\ \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \end{array}$$

We could then initialize all elements to 0:

```
MAT A= (0)
```

$$\begin{array}{c} \mathbf{A} \\ \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix} \end{array}$$

It is also important to realize that elements that are out of range in the REDIMed array still retain their values. The fifth thru ninth elements in **A** still equal 5 thru 9 even though they are now inaccessible. If we REDIM **A** back to a 3×3 array, these values will reappear. For example:

```
REDIM A(3,3)
```

results in:

$$\overset{\mathbf{A}}{\begin{pmatrix} 0 & 0 & 0 \\ 0 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}}$$

One of the major strengths of the REDIM statement is that it allows you to use variables for the subscript ranges: this is not allowed when you originally dimension an array. In effect, this enables you to dynamically dimension arrays. This should not be confused with the ALLOCATE statement which allows you to dynamically reserve memory for arrays. In the example below, for instance, we enter the dimensions from the keyboard.

```
10    OPTION BASE 1
20    COMPLEX A(100,100)
30    INPUT "Enter lower and upper bounds of dimensions",
      Low1,Up1,Low2,Up2
40    IF (Up1-Low1+1)*(Up2-Low2+1)>10000 THEN Too_big
50    REDIM A(Low1:Up1,Low2:Up2)
```

Line 40 tests to see whether the new dimensions are too big. If so, program control is passed to a line labelled "Too_big". If **line 40** were not present, the REDIM statement would return an error if the dimensions were too large.

Arrays and Arithmetic Operators

BASIC allows you to multiply, divide, add, and subtract scalars to an array, as well as to add, subtract, multiply, and divide one array to another. It is also possible for you to add all the elements in an array to produce a single result. This section covers a function and operations which allow you to perform these tasks with `INTEGER`, `REAL`, and `COMPLEX` data types.

Using the MAT Statement

All arithmetic functions involving arrays must be preceded by the MAT keyword. The specified operation is performed on each individual element in the operand array(s) and the results are placed in the result array. The result array must be dimensioned to be at least as large as the current size of the operand array(s). If it is of a different shape than the operand array(s), the system will redimension it. Given the array **A** below, note how these arithmetic functions are performed.

$$\mathbf{A}$$

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

To add `3` to each element of array `A`, you would use the following statement:

```
MAT B= A+(3)
```

The result of the above addition is array `B` below:

$$\mathbf{B}$$

$$\begin{pmatrix} 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{pmatrix}$$

To divide each element of array `B` above by 2, you would use the following statement:

```
MAT C= B/(2)
```

The result of the above division is array `C` given below:

$$\mathbf{C}$$

$$\begin{pmatrix} 2 & 2.5 & 3 \\ 3.5 & 4 & 4.5 \\ 5 & 5.5 & 6 \end{pmatrix}$$

To multiply each element in array C by a scalar expression, you would use a statement similar to the following:

```
MAT C= C*(1+1+1)
```

The above statement multiplied each element in array C by 3 and placed that result in array C as shown below:

$$\begin{array}{c} \mathbf{C} \\ \begin{pmatrix} 6 & 7.5 & 9 \\ 10.5 & 12 & 13.5 \\ 15 & 16.5 & 18 \end{pmatrix} \end{array}$$

Note that the result array can be the same as the operand array. Also, the scalar must be enclosed in parentheses.

In addition to performing arithmetic operations with scalars, you can also add, subtract, divide and multiply two arrays together. Except for multiplication with an asterisk, which is described later, these functions proceed as follows: Corresponding elements of each operand array are processed according to the specified operation, and the result is placed in the result array. The two operand arrays must be exactly the same size though their particular subscript ranges can be different. For multiplication, use a period rather than an asterisk. Using arrays **A** and **B** above, the statement,

```
MAT D= A+B
```

would give the array:

$$\begin{array}{c} \mathbf{D} \\ \begin{pmatrix} 5 & 7 & 9 \\ 11 & 13 & 15 \\ 17 & 19 & 21 \end{pmatrix} \end{array}$$

The statement,

```
MAT B= A.B
```

would give:

$$\begin{array}{c} \mathbf{B} \\ \begin{pmatrix} 4 & 10 & 18 \\ 28 & 40 & 54 \\ 70 & 88 & 108 \end{pmatrix} \end{array}$$

Again, the dimensioned size of the result array must be as large as the current size of each operand array. The two operand arrays must be identical in shape and size, but not necessarily in subscript ranges. For instance, **A** and **B** could have been dimensioned:

```
10    DIM A(1:3,2:4),B(-1:1,0:2)
        .
        .
```

Performing Arithmetic Operations with Complex Arrays

Remember that each of the operations mentioned in the previous section can be performed with complex arrays. The resulting array, if it is of type `COMPLEX`, will have both a real and an imaginary part in each element location. For example, you may have a two-dimensional complex array that looks like this:

Op_array

$$\begin{pmatrix} 2 & 4 & -1 & 5 \\ \\ -6 & 1 & 9 & 3 \end{pmatrix}$$

where the dimension statement is given as follows:

```
COMPLEX Op_array(-1:0,1:2)
```

The element `Op_array(-1,1)` contains the value:

```
2  4
```

where `2` is the real part of the complex number and `4` is the imaginary part.

If you were to multiply each of the complex values in the above matrix by a scalar value of 2, you would use the following statement:

```
MAT Complex_result= Op_array*(2)
```

The above statement would produce the following complex array:

Complex_result

$$\begin{pmatrix} 4 & 8 & -2 & 10 \\ \\ -12 & 2 & 18 & 6 \end{pmatrix}$$

Note that if the resulting array (`Complex_result`) had been of type `REAL` or `INTEGER`, the results in array `Complex_result` would look like this:

$$\begin{pmatrix} 4 & -2 \\ -12 & 18 \end{pmatrix}$$

This is due to the automatic type conversion made from `COMPLEX` to `REAL` or `INTEGER`. Notice that the imaginary part of the complex numbers in the array were dropped.

Summing the Elements in an Array

SUM is a function that returns the sum of *all* elements in an array. It works for arrays of any dimension. Given the array **A** below:

$$\begin{array}{c}\mathbf{A}\\ \begin{pmatrix} 4 & 2 & -1 \\ 3 & 8 & 16 \\ -5 & 2 & 0 \end{pmatrix}\end{array}$$

The following use of the SUM function:

```
SUM(A)
```

would return 29.

There are also functions that compute the sum of an entire row or column of an array. However, these functions are limited to two-dimensional arrays and are discussed in the subsequent section of this chapter entitled "Summing Rows and Columns of a Matrix."

Boolean Arrays

In addition to the arithmetic operators, you can also use relational operators with arrays. The result is a boolean[1] array (e.g., an array composed entirely of 1's and 0's). Given array **B**, suppose you wanted to know how many elements were greater than 50. First you execute the statement,

```
MAT F= B>(50)
```

which results in the array:

$$\mathbf{F} \quad \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 1 \end{pmatrix}$$

assuming array **B** has 4 elements in it greater than 50. Then you execute the statement,

```
PRINT SUM(F)
```

which causes the computer to display "4" on the current `PRINTER IS` device.

NOTE

The **only** comparison operators allowed with `COMPLEX` arrays are = and <>.

[1] Strictly speaking, these are not really boolean arrays since the values of the elements are not TRUE and FALSE.

You can also compare two arrays to each other. If, for example, you wanted to compare the two arrays below,

$$\begin{array}{cc} \mathbf{A} & \mathbf{B} \\ \begin{pmatrix} 1 & 3 & 5 \\ 2 & 8 & 7 \\ 1 & 4 & 6 \end{pmatrix} & \begin{pmatrix} 1 & 3 & 4 \\ 2 & 0 & 7 \\ 1 & 4 & 4 \end{pmatrix} \end{array}$$

you could execute the statement:

```
MAT C= A=B
```

By looking at **C**, you can tell which elements are the same for both **A** and **B**.

$$\begin{array}{c} \mathbf{C} \\ \begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix} \end{array}$$

Reordering Arrays

The MAT REORDER statement allows you to re-arrange an array so that one dimension is in a particular order. The new order is specified in a vector (a vector is a one-dimensional array). The vector contains the subscripts of the reordered dimension in their new order. The subscripts must correspond to the array's current dimensions and subscript ranges. Note that MAT REORDER works with REAL, INTEGER, COMPLEX and string arrays. However, as you might suspect, the reordering vector **cannot** be a COMPLEX vector.

Suppose **A** is the array below. Let us also assume that **A** has been dimensioned in OPTION BASE 1, and that the upper bound to both dimensions is 3.

$$\mathbf{A}\quad\begin{pmatrix}1 & 3 & 2\\ 4 & 5 & 7\\ 6 & 8 & 9\end{pmatrix}$$

To reverse the order of the rows, we would first dimension a vector,

```
10 DIM Reverse(3)
```

and then assign its elements the following values:

```
Reverse(1)=3
Reverse(2)=2
Reverse(3)=1
```

The vector **Reverse** now contains:

$$\mathbf{Reverse}\quad\begin{pmatrix}3 & 2 & 1\end{pmatrix}$$

If we execute the statement,

```
MAT REORDER A BY Reverse
```

the result array will be:

$$\mathbf{A}\quad\begin{pmatrix}6 & 8 & 9\\ 4 & 5 & 7\\ 1 & 3 & 2\end{pmatrix}$$

Note that the rows are exchanged, rather than the columns. This is because the default is to re-order the 1st dimension. However, you can override the default by specifying a particular dimension to be re-ordered. For example, if we wanted to reverse columns rather than rows, we could use the same vector, but this time specify dimension 2:

```
MAT REORDER A BY Reverse,2
```

The transformation would be:

$$\overset{\mathbf{A}}{\begin{pmatrix} 6 & 8 & 9 \\ 4 & 5 & 7 \\ 1 & 3 & 2 \end{pmatrix}} \Rightarrow \overset{\mathbf{A}}{\begin{pmatrix} 9 & 8 & 6 \\ 7 & 5 & 4 \\ 2 & 3 & 1 \end{pmatrix}}$$

Remember that although our examples are confined to two dimensions for illustrative purposes, the same principles apply to arrays of three and more dimensions. In a three-dimensional array, for instance, reordering the 1st dimension would reorder planes rather than rows or columns.

In most cases, rather than creating a reorder vector and assigning values to it, you will already have a vector as the result of a sort operation. This is true execept in the case of COMPLEX arrays which **can not** be sorted because the < and > operators **are not** defined for them.

Sorting Arrays

A frequent operation performed on arrays is a sort. Sorting an array rearranges the array so that one dimension (which you specify) is in numerical order. This section covers:

- Sorting with Automatic REORDER
- Sorting to a Vector

> **NOTE**
>
> You cannot sort COMPLEX arrays because < and > operations are not defined for them.

Sorting with Automatic REORDER

Given the array **A** below, watch how the MAT SORT changes it.

$$\begin{array}{c}\mathbf{A}\\ \begin{pmatrix}5&6&8\\3&5&1\\2&4&8\end{pmatrix}\end{array}$$

```
MAT SORT A(*,1)
```

$$\begin{array}{c}\mathbf{A}\\ \begin{pmatrix}2&4&8\\3&5&1\\5&6&8\end{pmatrix}\end{array}$$

The asterisk specifies the dimension to be sorted, and the subscript(s) tells which elements in that dimension to use as the sorting values. In the example above, we told the system to sort rows (asterisk is located in the first subscript position), and to use the first element in each row as the sorting value. With the new array **A** (from the sort performed above), the following statement will sort columns using the second element in each column as the sorting value.

```
MAT SORT A(2,*)
```

$$\begin{array}{c}\mathbf{A}\\ \begin{pmatrix}8&2&4\\1&3&5\\8&5&6\end{pmatrix}\end{array}$$

The key values in this sort are 1, 3 and 5, the second elements in each column. Sorting by placing the lowest values first is know as sorting by "ascending" order. This is the default. You can also sort by "descending" order by specifying the secondary keyword DES. For instance, the statement,

```
MAT SORT A(*,2) DES
```

would produce the following transformation:

$$\overset{\mathbf{A}}{\begin{pmatrix} 8 & 2 & 4 \\ 1 & 3 & 5 \\ 8 & 5 & 6 \end{pmatrix}} \Rightarrow \overset{\mathbf{A}}{\begin{pmatrix} 8 & 5 & 6 \\ 1 & 3 & 5 \\ 8 & 2 & 4 \end{pmatrix}}$$

Sometimes the values of two or more sorting elements are the same. For instance, if we sorted **A** by rows using the first element,

```
MAT SORT A(*,1)
```

we get:

$$\overset{\mathbf{A}}{\begin{pmatrix} 8 & 5 & 6 \\ 1 & 3 & 5 \\ 8 & 2 & 4 \end{pmatrix}} \Rightarrow \overset{\mathbf{A}}{\begin{pmatrix} 1 & 3 & 5 \\ 8 & 5 & 6 \\ 8 & 2 & 4 \end{pmatrix}}$$

The first elements in the last two rows are the same, so the system leaves them in the order they held before the sort. However, you can specify a second sort element to be used in the case of ties. We could execute:

```
MAT SORT A(*,1),(*,2)
```

This tells the system to sort by rows using the first element as the sorting value; and in the case of ties, to use the second element. The result array would be:

$$\overset{\mathbf{A}}{\begin{pmatrix} 1 & 3 & 5 \\ 8 & 5 & 6 \\ 8 & 2 & 4 \end{pmatrix}} \Rightarrow \overset{\mathbf{A}}{\begin{pmatrix} 1 & 3 & 5 \\ 8 & 2 & 4 \\ 8 & 5 & 6 \end{pmatrix}}$$

If a key is specified that is recognized by the system as rendering all other keys redundant (such as a non-substringed key for a one dimensional string array) no other keys can be specified. However, if the computer cannot tell that keys are redundant (such as `MAT SORT A(*,X),A(*,Y)` with `X` equal to `Y`) it will permit redundant keys. Redundant keys will slow down execution of the `MAT SORT` statement. If you include the `DES` secondary word, it refers only to the sort element which immediately precedes it.

Sorting to a Vector

So far, all of our sort examples have actually re-arranged the array in question. Alternatively, you can record the new order in a vector and leave the array intact. The vector must have been dimensioned to have at least as many elements as the current size of the array being sorted. If necessary, the system will redimension the vector. Thus, executing the statement:

```
MAT SORT A(3,*) TO Vect
```

with the array **A**:

$$\mathbf{A} \quad \begin{pmatrix} 1 & 3 & 5 \\ 8 & 2 & 4 \\ 8 & 5 & 6 \end{pmatrix}$$

The array **A** remains unchanged, but the vector **Vect** now contains the values:

$$\mathbf{Vect} \quad \begin{pmatrix} 2 & 3 & 1 \end{pmatrix}$$

This assumes that the array **A** has been dimensioned so that the subscript range is 1 thru 3. If **A** had been dimensioned:

```
10 DIM A(1:3,-1:1)
```

then **Vect** would contain the values:

$$\mathbf{Vect} \quad \begin{pmatrix} 0 & 1 & -1 \end{pmatrix}$$

This vector should look very reminiscent of the vectors used to reorder arrays. And, in fact, you can use these vectors in a `MAT REORDER` statement to rearrange the array. That is, we could now execute:

```
MAT REORDER A BY Vect,2
```

and the new array would be:

$$\mathbf{A} \quad \begin{pmatrix} 3 & 5 & 1 \\ 2 & 4 & 8 \\ 5 & 6 & 8 \end{pmatrix}$$

Note that the dimension number in the MAT REORDER statement corresponds to the position of the asterisk in the MAT SORT statement.

Sorting to a vector is particularly useful if you want to sort the same array along different dimensions or using different sort elements. Each sort can be stored in a vector to be used later. Meanwhile, the original array remains unchanged.

In addition, sorting to a vector allows you to use the same sorting order with parallel arrays. That is, if you have several arrays that contain data about the same elements, you can sort one of them, and then use that same sorting order to reorder the others.

Finally, sorting to a vector enables you to manipulate an unsorted array as if it were sorted. For instance, suppose you have the array shown below.

$$\mathbf{A}$$

$$\begin{pmatrix} 2 & 7 & 4 \\ 0 & 1 & 8 \\ 5 & 3 & 1 \end{pmatrix}$$

Let us also assume that the subscript range for each dimension in **A** is 1 thru 3. If we sort **A** to a vector **B**,

```
MAT SORT A(*,1) TO B
```

we can then use **B** to define elements in **A**. For instance to get the value of A(1,1) in its sorted form, we could write:

```
X=A(B(1),1)
```

In this case, **X** would equal 0. By incrementing the subscript value of **B**, we can simulate a sorted **A**

We should point out again that although these examples are two-dimensional, the same principles apply to arrays of any rank. You must have one, and only one, asterisk in the subscript list of a sort. The other subscripts specify the particular elements to be used as the sorting keys.

Searching Numeric Arrays

The purpose of the MAT SEARCH statement is to search for user-defined conditions within an array. This information is returned to a variable for recall and examination. Topics covered in this section are:

- Searching a Vector
- Numeric Comparisons in MAT SEARCH
- Searching a Three-dimensional Array
- Searching for Multiple Occurrences

For information on searching string arrays, read the chapter in this manual entitled "String Manipulation."

Searching a Vector

The following program called Mat_search (on the "Manual Examples Disc") demonstrates a search for maximum and minimum values and their locations and the number of occurrences of the maximum and minimum values. It also includes a search for the location of a value less than a given expression. Note that within this program is a sample of some of the possible types of searches you can make using the MAT SEARCH statement. **Lines 170** to **230** contain these sample searches.

```
100  OPTION BASE 1                    ! Select option base.
110  DIM Numbers(11)                  ! Dimension source array.
120  !
130  DATA 6,1,9,2,8,3,8,9,1,7,5       ! Random data.
140  !
150  READ Numbers(*)                  ! Input data to source array.
160  !
170  MAT SEARCH Numbers,MAX;Max       ! Search for maximum value.
180  MAT SEARCH Numbers,LOC MAX;Loc_max ! Find location of maximum value.
190  MAT SEARCH Numbers,MIN;Min       ! Search for minimum value.
200  MAT SEARCH Numbers,LOC MIN;Loc_min ! Find location of minimum value.
210  MAT SEARCH Numbers,#LOC(Max);Num_max ! Search for # of maximums.
220  MAT SEARCH Numbers,#LOC(Min);Num_min ! Search for # of minimums.
230  MAT SEARCH Numbers,LOC(<2);Loc_num,4 ! Starting with element 4,
240  !                                      return the first location of
250  !                                      a number less than 2.
260  !
```

```
270   ! Print the results.
280   !
290   PRINT "The maximum value is";Max;".";
300   PRINT " Its first occurrence is in array element";Loc_max;"."
310   PRINT
320   PRINT "The minimum value is";Min;"."; ! Print results.
330   PRINT " Its first occurrence is in array element";Loc_min;"."
340   PRINT
350   PRINT "The number of maximum value occurrences is";Num_max;"."
360   PRINT
370   PRINT "The number of minimum value occurrences is";Num_min;"."
380   PRINT
390   PRINT "Starting at array element 4, the first occurrence ";
400   PRINT "of a number"
410   PRINT "less than 2 is in array element";Loc_num;"."
420   END
```

If this program is run, the following results are obtained.

```
The maximum value is 9 . Its first occurrence is in array element 3 .

The minimum value is 1 . Its first occurrence is in array element 2 .

The number of maximum value occurrences is 2 .

The number of minimum value occurrences is 2 .

Starting at array element 4, the first occurrence of a number
less than 2 is in array element 9 .
```

The following is an explanation of the program `Mat_search`. All searches in this program start with the lower bound and work their way to the upper bound except where stated otherwise.

Line 170 of `Mat_search` uses the condition field `MAX` to search for the first occurrence of the maximum value in the array called `Numbers`. `Max` is the variable which receives the result of the search.

Line 180 of `Mat_search` uses the condition field `LOC MAX` to search for the location of the first occurrence of the maximum value in the array called `Numbers`. `Loc_max` is the variable which receives the result of the search.

Line 190 of Mat_search uses the condition field MIN to search for the first occurrence of the minimum value in the array called Numbers. Min is the variable which receives the result of the search.

Line 200 of Mat_search uses the condition field LOC MIN to search for the location of the first occurrence of the minimum value in the array called Numbers. Loc_min is the variable which receives the result of the search.

Line 210 of Mat_search uses the condition field #LOC(Max) to search for the total number of occurrences of the value of the variable Max in the array Numbers. Max was determined as the maximum value in the array on **line 170** and the new variable Num_max receives the result of executing program **line 210**.

Line 220 of Mat_search uses the condition field #LOC(Min) to search for the total number of occurrences of the value of the variable Min in the array Numbers. Min was determined as the minimum value in the array on **line 190** and the new variable Num_min receives the result of executing program **line 220**.

Line 230 of Mat_search uses the condition field LOC(<2) to search for the location in the array Numbers of the first occurrence of a number less than 2. There is another field added to the MAT SEARCH statement called the "starting subscript." This field allows you to begin a search from a location other than the lower bound. In the case of program **line 230**, the starting subscript is 4 which says the search will begin at array element 4 and continue toward the upper bound of the array.

Searching an Array by Descending Subscripts

If for some reason you need to search an array by descending subscript values, use the MAT SEARCH statement's DES option after the array's key specifier (i.e. Array(1,*) DES). This option causes a search to begin at the upper bound of a dimension in an array and proceed toward the lower bound of that same dimension. If a *starting subscript* is specified in the MAT SEARCH statement, then the search will begin at that specified location in the dimension being searched and proceeds toward the lower bound of that dimension. For example,

```
MAT SEARCH Array(1,*) DES,MAX;Max_value,6
```

searches the first row of a two-dimensional array called Array starting from the 6th element of that row and going toward the first element.

Before continuing with our discussion of searching an array by descending subscripts, let's look at the following table to clarify the difference between a search done by ascending subscripts and one done by descending subscripts.

Search Order	Starting Subscript Given	No Starting Subscript Given
ascending (default)	starting subscript → upper bound	lower bound → upper bound
descending	starting subscript → lower bound	upper bound → lower bound

If you substitute program **lines 170** to **230** given below for the same program lines in the program called `Mat_search` explained in the previous section, you will find that the results produced are different.

```
170   MAT SEARCH Numbers DES,MAX;Max         ! Search for maximum value.
180   MAT SEARCH Numbers DES,LOC MAX;Loc_max !Find location of maximum value.
190   MAT SEARCH Numbers DES,MIN;Min         ! Search for minimum value.
200   MAT SEARCH Numbers DES,LOC MIN;Loc_min !Find location of minimum value.
210   MAT SEARCH Numbers DES,#LOC(Max);Num_max ! Search for # of maximums.
220   MAT SEARCH Numbers DES,#LOC(Min);Num_min ! Search for # of minimums.
230   MAT SEARCH Numbers DES,LOC(<2);Loc_num,4 ! Starting with element 4,
240   !                                          return the first location of
250   !                                          a number less than 2.
```

Executing the above program lines within the program called `Mat_search`, will result in the following output:

```
The maximum value is 9 . Its first occurrence is in array element 8 .

The minimum value is 1 . Its first occurrence is in array element 9 .

The number of maximum value occurrences is 2 .

The number of minimum value occurrences is 2 .

Starting at array element 4, the first occurrence of a number
less than 2 is in array element 2 .
```

Notice that the results given for the maximum and minimum values did not change and neither did the number of times they occurred within the output on the display. However, the locations of the data values did change. The reason for the change in data location is you began the search from the upper bound of the dimension being searched. Since there were two 9's in the dimension being searched the first one encountered was in a different location than the first time this program was run. The same thing is true for the search for the location of a minimum value and the value less than 2.

Numeric Comparisons in MAT SEARCH

Numeric comparisons are made when using the MAT SEARCH statement. The type of comparison made is determined by the *condition* option (e.g. MAX, LOC, MIN, etc.) of this statement. The MAX, MIN, LOC MAX, and LOC MIN conditions imply the use of the > and < relational operators (for MAX and MIN, respectively).

The *condition* options LOC and #LOC of a MAT SEARCH statement have as their arguments a relational operator along with an expression. There are six relational operators which can be used with this argument, they are: <, <= , =, <>, >=, >. If none of these operators are used, the default operator = is assumed. Some examples of these options being used to search a one-dimensional array are as follows:

`MAT SEARCH Array,LOC(<>4);Location` assigns the location of the first element found not equal to 4 to the variable `Location`.

`MAT SEARCH Array,#LOC(>=7);Num_value` assigns the total number of occurrences of values greater than or equal to 7 in the array called `Array` to the variable `Num_value`.

The expression is converted to the same data type as the array before the comparisons are done. For example, if `Array` is an INTEGER array, then the following statement:

```
MAT SEARCH Array,LOC(2.7);Location
```

assigns the *location* of the first occurrence of the value 3 (in `Array`) to the variable `Location`.

NOTE

COMPLEX arrays can **only** be searched using the = and <> relational operators.

For a complete discussion of the *condition* option in the MAT SEARCH statement read the *BASIC Language Reference.*

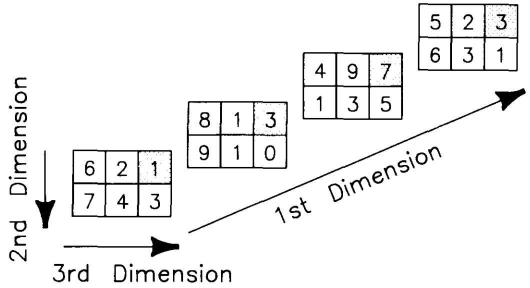
Searching a Three-Dimensional Array

It is important to know that the MAT SEARCH statement works on only one dimension of an array. If you are searching a one-dimensional array there is no problem with searching the whole array. However, if you are searching a multi-dimensional array you need to specify only one dimension of that array in the *key specifier* of the MAT SEARCH statement. For example, assume that array Search_array is a three dimensional numeric array dimensioned using the following dimension statement:

```
100  DIM Search_array(4,2,3)
```

and that this array contains the following random numbers:

Search_array



Assume that the shaded locations are to be searched until one is found which contains a value greater than 5. Let Value_loc represent the variable to which the location of the specified condition is returned.

The shaded memory locations must be described by a *key specifier.* Since they lie along the first dimension, the planes in which they lie are defined accordingly. The correct *key specifier* is:

```
(*,1,3)
```

Note that the *key specifier* is written in the same format as that used for a MAT SORT *key specifier.* The subscripts are written in the same order as the array dimensions (see the numbered arrows). The first subscript is an asterisk which indicates that the subscript is varied over its range of values, and the remaining subscripts define the fixed "row" and "column" locations.

The correct search statement for this example is

```
MAT SEARCH Search_array(*,1,3),LOC(>5);Value_loc
```

where:

`Search_array`	is the array being searched.
`(*,1,3)`	defines the locations to be searched.
`LOC(>5)`	specifies the condition to be satisfied.
`Value_loc`	is the variable to which the location value is returned.

After execution of the search statement, the number 3 is returned to the variable `Value_loc`. This is the first plane where the value of the specified location satisfies the condition option (i.e.. it contains a value greater that 5). Searching begins at the lower bound of the *key specifier's* dimension which contains the asterisk, and proceeds toward the upper bound of the same dimension until a value satisfying the *condition* option is located.

If a condition is not satisfied upon completion of the searching process, a value one greater than the upper limit of the varied subscript is returned to the numeric variable. For example, if the specified locations in the previous example are searched for a value greater than 8, none is found. Therefore, a value of 5 representing a number one greater than the plane containing the last searched location is returned to `Value_loc`. Note that if the upper limit of the varied subscript is 32 767, the value returned by an unsuccessful search is −32 768.

The *key specifier* initially defines the range of locations to be searched. If you do not wish to search the entire range, a *starting subscript* specifier can be used to designate where the search is to begin. Using the previous example, assume that the search process is to scan only the last three planes for the location of a value less than 5. The correct search statement to be used is

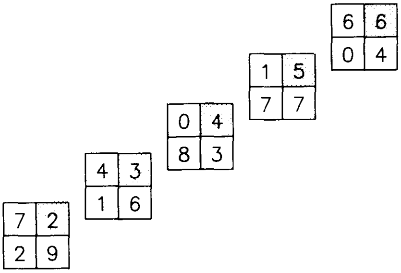
```
MAT SEARCH Search_array(*,1,3),LOC(<5);Value_loc,2
```

where the number 2 directs the search to begin at the specified location in the second plane and proceed to the last plane. Upon execution, the number 2 indicating the second plane is returned to the variable `Value_loc` because the content of its specified location is the first to satisfy the *condition.*

Searching for Multiple Occurrences

Normally, a LOC search ends at the first location which satisfies the specified condition. However, additional satisfactory values may exist beyond that location. By setting the starting address to be one greater than the content of the Value_loc variable, a search can be continued past the first location which satisfied the LOC condition. This automatically continues a search from where a previous search left off. All values which satisfy the given condition can be obtained in this way. As an example, assume that array Search_array is a three dimensional numeric array containing random numbers as shown.

Search_array



Assume that the shaded memory locations are to searched for values greater than 2. A single MAT SEARCH scan would stop at the second plane since it is the first one encountered whose content satisfies the condition.

However, by constructing a loop and using the proper *starting subscript* specifier, the search can be made to continue through the range of specified locations. The following program demonstrates this feature.

```
100   OPTION BASE 1              ! Select the option base.
110   DIM Array(5,2,2)           ! Dimension Array.
120   !
130   DATA 7,2,2,9,4,3,1,6,0,4,8,3,1,5,7,7,6,6,0,4 ! Random data.
140   !
150   READ Array(*)              ! Read data into Array.
160   !
170   Subscript=0                ! Initialize Subscript so first
180   !                            search is in plane 1 of Array.
190   !
200   LOOP
210     MAT SEARCH Array(*,1,2),LOC(>2);Subscript,Subscript+1 ! Search
220     !            for locations containing numbers greater than 2.
230     EXIT IF Subscript=6      ! Test: Have all locations been searched?
240     DISP Subscript;          ! Display most recent search results.
250     EXIT IF Subscript=5      ! Test: Has search reached last location?
260     !                                If not, continue.
270   END LOOP
280   END
```

The variable `Subscript` is initially set to zero. **Lines 200** through **270** form a search loop. **Line 210** begins the search routine. Since `Subscript` was set to zero, the *starting subscript* specifier (`Subscript + 1`) directs the search to begin at plane one. **Line 230** tests to see if the specified locations have all been searched. (Remember, if all locations have been searched, a number one greater than the last plane searched is returned to the variable. In this case, that number is six.) If all locations have not yet been searched, **line 240** displays the contents of `Subscript` (it now contains a satisfactory value). If `Subscript` contains a value less than 5, the search is not finished and **line 270** directs the program to search again until a loop exit is made. Due to the nature of the *starting subscript* specifier, the search begins this time at the next memory location beyond that which satisfied the condition previously. In other words, the search resumes where it left off. If you run this program, the following should be displayed:

```
2  3  4  5
```

Matrices and Vectors

A two-dimensional numeric array is called a "matrix" and a one-dimensional numeric array is called a "vector". An entire branch of mathematics is devoted to matrices and vectors, and their applications are surprisingly broad. Keep in mind that the functions described in this section apply only to two-dimensional, and occasionally one-dimensional arrays, but never to arrays of more than two dimensions. Also, all functions described in this section apply to `REAL`, `INTEGER` and `COMPLEX` data types.

Matrix Multiplication

You may recall from our discussion of arrays and arithmetic operations that the asterisk (*) is reserved for matrix multiplication. If **A** is an *i-by-k* matrix and **B** is a *k-by-j* matrix. then **C=A*B** is defined by the following equation:

$$C_{ij} = \sum_{k=1}^{n} A_{ik}B_{kj}$$

Translated into english, this equation means that the element in the ith row and *jth* column of the product (**C**) is the sum of the products by pairs of the elements in the *ith* row of **A** and the *jth* column of **B**. A couple of examples will help make this clear.

Suppose **A** and **B** are the matrices shown below.

$$\begin{array}{cc} \mathbf{A} & \mathbf{B} \\ \begin{pmatrix} 3 & 8 & 2 \\ 1 & 6 & 5 \\ 4 & 2 & 0 \end{pmatrix} & \begin{pmatrix} 1 & -2 & 3 \\ -6 & 4 & 7 \\ 0 & 8 & 2 \end{pmatrix} \end{array}$$

If `C=A*B`, then:

```
C(1,1)=A(1,1)*B(1,1)+A(1,2)*B(2,1)+A(1,3)*B(3,1)=(3*1)+(8*-6)+(2*0)=-45
C(2,1)=A(2,1)*B(1,1)+A(2,2)*B(2,1)+A(2,3)*B(3,1)=(1*1)+(6*-6)+(5*0)=-35
C(3,2)=A(3,1)*B(1,2)+A(3,2)*B(2,2)+A(3,3)*B(3,2)=(4*-2)+(2*4)+(0*8)=0
```

Following this procedure for each element in `C`, we get the matrix shown below.

```
MAT C= A*B
```

$$\begin{array}{c} \mathbf{C} \\ \begin{pmatrix} -45 & 42 & 69 \\ -35 & 62 & 55 \\ -8 & 0 & 26 \end{pmatrix} \end{array}$$

Note that the product is a 3×3 matrix. There are three general rules to matrix multiplication:

- Multiplication between two matrices is legal only if the second dimension of the first array is the same size as the first dimension of the second array. That is, the two inner dimensions must be the same.
- The result matrix will have the same number of rows as the first operand matrix and the same number of columns as the second operand matrix. That is, the dimensions of the result matrix will be the same as the outer dimensions of the operand matrices.
- The result array cannot be the same as either of the operand arrays. For example,

  ```
  MAT A= A*B
  ```

  is an illegal statement.

If **A** is a 2×3 matrix and **B** is a 3×2 matrix, **A*****B** will result in a 2×2 matrix. **B*****A**, on the other hand, produces a 3×3 matrix. Given the two matrices below, you can see how their position in the equation affects the product.

$$\mathbf{A} \quad \begin{pmatrix} 6 & 8 & -1 \\ 2 & -3 & 4 \end{pmatrix} \qquad \mathbf{B} \quad \begin{pmatrix} -1 & 1 \\ 2 & -2 \\ 3 & 4 \end{pmatrix}$$

$$\mathbf{A*B} \quad \begin{pmatrix} 7 & -14 \\ 4 & 24 \end{pmatrix} \qquad \mathbf{B*A} \quad \begin{pmatrix} -4 & -11 & 5 \\ 8 & 22 & -10 \\ 26 & 12 & 13 \end{pmatrix}$$

Multiplication With Vectors

We described a vector as a one-dimensional array. For instance,

```
10 DIM A(3)
```

would create a vector with three elements and a rank of 1. Suppose we give **A** the values shown below.

$$\mathbf{A} \quad \begin{pmatrix} 1 & 2 & 3 \end{pmatrix}$$

Notice that we have portrayed **A** as a row vector. We could have just as easily portrayed **A** as a column vector:

$$\begin{array}{c}\mathbf{A}\\ \begin{pmatrix}1\\2\\3\end{pmatrix}\end{array}$$

So which is it? A row vector or a column vector? Actually, a vector can behave like either depending on its position in an equation. If a vector is the first operand in a multiplication, then it acts like an 1×n array (row vector); if it's the second operand, it behaves like a n×1 array (column vector); and if it's the result array, it can act like either. A few examples will help illustrate these principles. Let **A** be the vector shown above, and **B**, **C**, and **D** be the arrays shown below.

$$\begin{array}{ccc}\mathbf{B} & \mathbf{C} & \mathbf{D}\\ \begin{pmatrix}1&2&3\\4&5&6\\7&8&9\end{pmatrix} & \begin{pmatrix}0&0&0\\0&0&0\\0&0&0\end{pmatrix} & \begin{pmatrix}2\\2\\2\end{pmatrix}\end{array}$$

Let us suppose that **D** has been explicitly defined as a two-dimensional array:

```
10  DIM D(3,1)
```

If we execute:

```
MAT C= A*D
```

we get:

$$\begin{array}{c}\mathbf{C}\\ \begin{pmatrix}12\end{pmatrix}\end{array}$$

Since **A** is the first operand, it behaves like a 1×3 matrix. The equation, therefore, is:

$$C = \begin{pmatrix}1&2&3\end{pmatrix} * \begin{pmatrix}2\\2\\2\end{pmatrix}$$

The result is a 1×1 matrix. If we try to reverse the order:

```
MAT C= D*A
```

the system returns:

```
ERROR 16  Improper dimensions
```

This is because we tried to multiply a 3×1 matrix by a 3×1 matrix:

$$C = \begin{pmatrix} 2 \\ 2 \\ 2 \end{pmatrix} * \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$$

Since the inner dimensions are not the same, the system returns an error. Suppose we try:

```
MAT C= B*A
```

In this case, we are multiplying a 3×3 array to a column vector.

$$C = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} * \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$$

The result is a 3×1 matrix:

$$\begin{array}{c} \mathbf{C} \\ \begin{pmatrix} 14 \\ 32 \\ 50 \end{pmatrix} \end{array}$$

If the result array is a vector, it will behave like either a row vector or a column vector depending on which is called for. The only other possibility is if both operand arrays are vectors. In this case, the result is always a 1×1 array. For instance, if **A** and **B** are vectors which are dimensioned as follows:

```
COMPLEX A(3),B(3)
```

and they contain the following complex values:

$$\begin{array}{cc} \mathbf{A} & \mathbf{B} \\ \begin{pmatrix} 2 & 1 \\ 4 & -1 \\ 6 & -2 \end{pmatrix} & \begin{pmatrix} 0 & -3 \\ 1 & -6 \\ -1 & 5 \end{pmatrix} \end{array}$$

then multiplying **A** by **B** results in a 1×1 array when the following equation is used:

```
MAT C= A*B
```

$$C = \begin{pmatrix} 2 & 1 & \quad 4 & -1 & \quad 6 & -2 \end{pmatrix} * \begin{pmatrix} 0 & -3 \\ 1 & -6 \\ -1 & 5 \end{pmatrix}$$

C equals `5   1`. Reversing the operand arrays, we get:

```
MAT C= B*A
```

$$C = \begin{pmatrix} 0 & -3 & \quad 1 & -6 & \quad -1 & 5 \end{pmatrix} * \begin{pmatrix} 2 & 1 \\ 4 & -1 \\ 6 & -2 \end{pmatrix}$$

Again, **C** equals `5   1`. Because the product of two vectors is always a single element, BASIC has a `DOT` function that multiplies two vectors and comes up with a `REAL`, `INTEGER` or `COMPLEX` numeric. For example,

```
X=DOT(A,B)
```

would assign the value `5   1` to `X`. If both vectors are `INTEGER`, then the product is `INTEGER`. If one is `COMPLEX`, the product is `COMPLEX`. Otherwise, the product is `REAL`. The two vectors must be the same size or the system will return an error.

Identity Matrix

An "identity matrix" is defined as a matrix which, when multiplied to another matrix **A**, produces the same matrix **A**. It is analogous to a 1 in normal arithmetic. For example, if **I** stands for an identity matrix, then **A=I*A** and also **A=A*I**. In order for an identity matrix to exist at all, **A** must be a square matrix (e.g., it must have the same number of columns as rows).

As it turns out, all identity matrices have the same form. They are square and consist of 1's along the main diagonal, and 0's everywhere else. For example, if **A** is a 3×3 matrix, then the identity matrix for **A** is:

$$\begin{array}{c}\mathbf{I}\\ \begin{pmatrix}1 & 0 & 0\\ 0 & 1 & 0\\ 0 & 0 & 1\end{pmatrix}\end{array}$$

For a 4×4 matrix, **I** would be:

$$\begin{array}{c}\mathbf{I}\\ \begin{pmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 1 & 0\\ 0 & 0 & 0 & 1\end{pmatrix}\end{array}$$

Since identity matrices are used frequently in matrix arithmetic, BASIC has a special function (`IDN`) that turns a square matrix into an identity matrix. For instance:

```
10  OPTION BASE 1
20  COMPLEX I(2,2)
30  MAT I= IDN
      .
      .
```

The `COMPLEX` matrix **I** now contains the elements:

$$\begin{array}{c}\mathbf{I}\\ \begin{pmatrix}1\;\;0 & \quad 0\;\;0\\ 0\;\;0 & \quad 1\;\;0\end{pmatrix}\end{array}$$

If **I** was not a square matrix, line 20 would have returned an error.

Inverse Matrix

Although division is not defined for matrices, there is a similar operation which involves finding the inverse of a matrix. As with identity matrices, a matrix must be square in order to have an inverse. Inverse matrices are notated by a superscript -1. If **A** is a square matrix, then $\mathbf{A}^{-1}$ denotes its inverse. The inverse is defined by the equation:

$$\mathbf{A}^*\mathbf{A}^{-1}=\mathbf{I}$$

where **I** is the identity matrix. You can see how similar this is to division since, if **A** were a real number, then:

```
A*(1/A)=1
```

NOTE

When using the inverse function (INV) if the source is INTEGER or REAL, then the destination **must be** REAL. If the source is COMPLEX, then the destination **must be** COMPLEX.

The inverse of a matrix is found by using the INV function. For instance, the inverse of:

$$\begin{array}{c}\mathbf{A}\\ \begin{pmatrix} 0 & 2 & 0 \\ -1 & 2 & 0 \\ 2 & 0 & 2 \end{pmatrix}\end{array}$$

is found by executing:

```
MAT A_inv= INV(A)
```

The system computes the values of the inverse and places them in the matrix **A_inv**:

$$\begin{array}{c}\mathbf{A\_inv}\\ \begin{pmatrix} 1 & -1 & 0 \\ .5 & 0 & 0 \\ -1 & 1 & .5 \end{pmatrix}\end{array}$$

To check that this is really the inverse, you could execute the statement:

```
MAT B= A*A_inv
```

As expected, **B** turns out to be an identity matrix:

$$\begin{array}{c}\mathbf{B}\\ \begin{pmatrix}1 & 0 & 0\\ 0 & 1 & 0\\ 0 & 0 & 1\end{pmatrix}\end{array}$$

Unfortunately, these expectations are not always fulfilled. Some matrices do not have an inverse. In other words, for a certain matrix called **A**, there exists no other matrix that, that when multiplied with (or by) **A** produces an identity matrix. Matrices that don't have an inverse are called "singular". Singular matrices are easily detected and therefore aren't too dangerous. A more troublesome type of matrix is one that is "ill-conditioned". Ill-conditioned matrices are ones whose inverse can't be found by the computer because of round-off errors. These are difficult to detect and almost impossible to correct. We'll talk more about singular and ill-conditioned matrices, but before we do, we should discuss why you'd use an inverse in the first place.

Solving Simultaneous Equations

One of the most common applications of matrices is in the solution of simultaneous equations. Simultaneous equations can be solved for REAL, INTEGER or COMPLEX data types.

Suppose we have the three equations shown below:

$$\begin{aligned} &4X + 2Y - Z = 5 \\ &2X - 3Y + 3Z = 5 \\ &X + Y - 2Z = -3 \end{aligned}$$

Note that there are three unknowns (X,Y, and Z) and three equations. This is a necessity for solving by matrix arithmetic: you must have the same number of equations as unknowns. We can re-write these equations in matrix format as the product of two arrays:

$$\begin{pmatrix} 4 & 2 & -1 \\ 2 & -3 & 3 \\ 1 & 1 & -2 \end{pmatrix} * \begin{pmatrix} X \\ Y \\ Z \end{pmatrix} = \begin{pmatrix} 5 \\ 5 \\ -3 \end{pmatrix}$$

For the sake of simplicity, let's name these three arrays **A**, **B**, and **C**. The equation, therefore, is:

$$\mathbf{A}^*\mathbf{B}=\mathbf{C}$$

If we multiply both sides of the equation by the inverse of **A**, we get:

$$\mathbf{A}^{-1}*\mathbf{A}^*\mathbf{B}=\mathbf{A}^{-1}*\mathbf{C}$$

Since $\mathbf{A}^{-1}*\mathbf{A}$ is simply a 3×3 identity matrix, the equation simplifies to:

$$\mathbf{I}^*\mathbf{B}=\mathbf{A}^{-1}*\mathbf{C}$$

which further simplifies to:

$$\mathbf{B}=\mathbf{A}^{-1}*\mathbf{C}$$

Remember, **B** is the matrix that contains the three variables X,Y and Z. To solve for these variables, therefore, all we have to do is multiply the matrix **C** by $\mathbf{A}^{-1}$. This is accomplished in the program lines listed below.

```
          .
          .
200  DIM Solution(3),A_inv(3,3)
220  MAT A_inv= INV(A)
230  MAT Solution= A_inv*C
240  PRINT "X=";Solution(1)
250  PRINT "Y=";Solution(2)
260  PRINT "Z=";Solution(3)
          .
          .
```

When we run this program, it will print the values of X, Y, and Z. The values are:

```
X=1
Y=2
Z=3
```

For any set of simultaneous equations where there are the same number of unknown variables as there are equations, there are three possible classes of solution.

- There is no solution (e.g., there exist no values for the variables such that all of the equations are true).
- There are an infinite number of solutions.
- There is one, and only one, solution.

The first two cases are called "singular" sets of equations. You may recall that a singular matrix is one that has no inverse. It should not be surprising, therefore, that singular sets of equations always result in singular matrices when they are translated to matrix form. This is explained in the next section.

Singular Matrices

Any set of equations that has no solution or an infinite number of solutions is singular. Likewise, the matrix formed from these equations is also singular. More specifically, we mean the matrix on the left-hand side of the equation, what we've been calling matrix **A**. Consider the two equations listed below:

```
4X+6Y=5
4X+6Y=6
```

Obviously, there is no solution to this set of equations because any values assigned to X and Y will make only one of the equations true, not both. It is important to realize, however, that the singularity of these equations has nothing to do with the values on the right hand side of the equation. If, for example, we made the two equations the same,

```
4X+6Y=6
4X+6Y=6
```

then there would be an infinite number of solutions. For instance, X could equal 0 and Y equal 1, or X could equal 1.5 and Y could equal 0. In fact, so long as X=1.5(1-Y), the two equations will always be true. What is important here is that the two equations,

```
4X+6Y=
4X+6Y=
```

will be singular regardless of what we put on the right-hand side of the equal sign. If we translate these equations into matrix form, we get:

$$\begin{pmatrix} 4 & 6 \\ 4 & 6 \end{pmatrix} * \begin{pmatrix} X \\ Y \end{pmatrix}$$

The matrix,

$$\begin{pmatrix} 4 & 6 \\ 4 & 6 \end{pmatrix}$$

is singular: it has no inverse. If, however, we call this matrix **A** and do an INV on it, the system will not report an error. On the contrary, it will go ahead and find what it thinks is an inverse. However, whatever matrix it comes up with will not be the inverse. Let's see what happens with our singular matrix **A**.

```
MAT A_inv= INV(A)
PRINT A_inv(*)
```

When we execute these statements, the system will display the following:

```
 .666666666667   .166666666667    0    -1
```

Arranging these values in the proper rows and columns, we get:

$$\begin{array}{c}\textbf{A\_inv}\\ \begin{pmatrix} .66666666667 & 0 \\ .16666666667 & -1 \end{pmatrix}\end{array}$$

To see whether this is a real inverse, we can multiply it by **A**. If it is the inverse, the product should be an identity matrix.

```
MAT I= A*A_inv
```

$$\begin{array}{c}\textbf{I}\\ \begin{pmatrix} 3.33333333333 & 5 \\ -4 & -6 \end{pmatrix}\end{array}$$

Obviously, the system has made a mistake — **A_inv** is not the inverse of **A**. So how do we know if an inverse is valid? Or, to put it another way, how do we detect a singular matrix? We have just seen one method: multiply the matrix by its inverse and see whether you get an identity matrix. There is, however, a much easier method. You simply look at the "determinant" of the matrix.

The Determinant of a Matrix

The determinant of a matrix is defined somewhat mysteriously as the sum of all possible products formed by taking one element from each row in order starting from the top and one element from each column, where the sign of each product depends on the permutation of the column indices.

It's not really important that you understand how to calculate a determinant since the computer does it for you whenever you use the DET function. The DET function can be used with REAL, INTEGER and COMPLEX data types. For instance, to print the determinant of matrix **A**, you would write:

```
PRINT DET(A)
```

Also the determinant is a byproduct of inversions. Thus, whenever you invert a matrix, the system computes the determinant and stores it. If you use DET without specifying a matrix, the system will return the determinant of the matrix most recently inverted. For example,

```
MAT A_inv= INV(A)
PRINT DET
```

would print the determinant of **A**.

Although the computation of the determinant is quite complex, its significance is very simple. If the determinant of a matrix equals 0, either a REAL underflow occurred during the inversion or **the matrix is singular**. To find out if an inversion is invalid, therefore, you merely test the matrix's determinant. If the determinant is zero, then the inverse is invalid. For example if **A** is a square matrix, we could execute:

```
          .
          .
100  MAT A_inv= INV(A)
110  IF DET=0 THEN Singular
          .
          .
```

If **A** is singular, program control is passed to a line named Singular. Note that we did not have to specify a matrix in line 110 since **A** was the last matrix inverted.

Unless you know for certain that a matrix is not singular, we recommend that you use the determinant test after each inversion. Otherwise, you may perform calculations using an invalid inverse.

Ill-Conditioned Matrices

In a few unusual cases, the inverse of a matrix will be invalid even though the determinant of the matrix is non-zero. These situations occur due to round-off errors internal to the computer. They are difficult to detect and even more difficult to correct. Fortunately, they occur very rarely. Unless you are having problems with a program involving matrix operations producing unexpected results, you can skip over to "Miscellaneous Matrix Functions." If you **are** having problems, listed below is an example of an ill-conditioned set of equations.

```
X(1) + 0X(2) + 3X(3) + 8X(4) = 12
2X(1) + X(2) + 6X(3) + 15.9X(4) = 24.9
3X(1) + X(2) + 8.9X(3) + 24X(4) = 36.9
4X(1) + X(2) + 11.9X(3) + 32X(4) = 48.9
```

We have selected the numbers on the right-hand side of the equation so that all of the X's equal 1. Watch what happens though when we try to solve these equations through matrix inversion. First, we set up the equations in matrix format.

$$\overset{\textbf{A}}{\begin{pmatrix} 1 & 0 & 3 & 8 \\ 2 & 1 & 6 & 15.9 \\ 3 & 1 & 8.9 & 24 \\ 4 & 1 & 11.9 & 32 \end{pmatrix}} * \overset{\textbf{ar}}{\begin{pmatrix} X1 \\ X2 \\ X3 \\ X4 \end{pmatrix}} = \overset{\textbf{Ans}}{\begin{pmatrix} 12 \\ 24.9 \\ 36.9 \\ 48.9 \end{pmatrix}}$$

Then we execute the program statements below.

```
100   MAT A_inv= INV(A)
110   MAT Var= A_inv*Ans
120   FOR Y=1 TO 4
130     PRINT Using """X("",D,"")="",K";Y,Var(Y)
140   NEXT Y
```

The computer displays:

```
X(1)=256
X(2)=0
X(3)=-32
X(4)=-16
```

Obviously something has gone wrong. The problem is that the inverse found by the computer is far off the mark from the actual inverse. The system of equations, though, is not singular. The determinant, though small, does not equal zero.

Detecting Ill-conditioned Matrices

Now that you've seen how ill-conditioning can affect the solutions to a set of simultaneous equations, you're probably wondering how you can tell an ill-conditioned matrix when you see one. There are a number of different techniques, none of which is entirely failproof. Used together, however, they are quite dependable.

In general, the determinant of an ill-conditioned matrix is very small compared with the elements of the matrix. So one of the first steps you can take is to look at the determinant. The term "very small" is, of course, relative. If a matrix contained elements all greater than 1000, then a determinant that equaled 10 would be very small. On the other hand, if all the elements in an array were less than 20 then a determinant of 10 would be quite reasonable. One equation for determining whether the determinant is "too small" is given below:

$$\frac{\mathrm{DET(A)}}{\sqrt{\sum_{i=1}^{n}\sum_{j=1}^{n} A_{ij}^{2}}} << 1$$

We can execute this equation in a program as follows.

```
100 FOR X=(BASE A,1) TO (SIZE A,1)+(BASE A,1)-1
120   FOR Y=(BASE A,2) TO (SIZE A,2)+(BASE A,2)-1
130     Total=Total+A(X,Y)^2
140   NEXT Y
150 NEXT X
160 Test=DET(A)/SQR(Total)
170 IF Test<.001 THEN Ill_con
```

Note that line 170 can be changed depending on how much accuracy you require for your particular application. If we execute this program for the ill-conditioned matrix discussed earlier, the value of "Test" comes out to 9.527E−19. Since this value is much smaller than .001, this test would have correctly identified **A** as an ill-conditioned matrix.

Another technique for detecting ill-conditioned matrices is to multiply the matrix by its inverse and compare the product with the identity matrix. Again, you can demand as much accuracy as necessary. In the program below, we look for any elements in the product that differ by more than .001 from the identity matrix.

```
100  MAT I= IDN
110  MAT A_inv= INV(A)
120  MAT Product= A_inv*A
130  MAT Differ= Product-I
140  MAT Compare= Differ>(.001)
150  MAT Compare1= Differ<(-.001)
160  IF SUM(Compare)+SUM(Compare1)>0 THEN Ill_con
```

Applying this algorithm to our ill-conditioned matrix, we get:

A*A_inv

$$\begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & -.5 & 0 \\ -2 & -.5 & -4 & -16 \\ -2 & -.5 & -8 & -16 \end{pmatrix}$$

As you can see, 12 of the 16 elements differ from the identity matrix by more than 1, so this test also would have worked.

One drawback of this method is that it requires several additional matrices. If you are strapped for memory, this method could be unsatisfactory.

A third technique is to take the inverse of the inverse and compare it to the original matrix. The program below utilizes this method. Again, we are looking for differences greater than 0.001.

```
100  MAT A_inv= INV(A)
120  MAT A_inv_inv= INV(Ainv)
130  MAT Differ= A_inv_inv-A
140  MAT Compare= Differ>(.001)
150  MAT Compare1= Differ<(-.001)
160  IF SUM(Compare)+SUM(Compare1)>0 THEN Ill_con
```

Applying this technique to our ill-conditioned matrix, we find that all 16 elements of **A_inv_inv** differ from **A** by more than .001.

This technique will in general find more ill-conditioned matrices than the previous one. This is because any round-off errors are exaggerated by the second inverse. By the same token, it will occasionally detect an ill-conditioned matrix which might actually have been alright before the second inverse.

As stated before, none of these methods alone is decisive. What it all c omes down to is that the precision of `MAT INV` falls off as a matrix approaches singularity. By using combinations of the tests described above, it is possible to determine how much precision has been lost, and then compare it to the precision actually required by your application.

Miscellaneous Matrix Functions

These functions are useful for obtaining the transpose of a matrix, summing the rows and columns of a matrix, and performing complex array operations. The topics covered are as follows:

- Transpose Function
- Summing Rows and Columns of a Matrix
- Examples of Complex Array Operations

Transpose Function

There are a few matrix functions that we haven't discussed yet. One of these is the transpose function (TRN). The transpose of a matrix is derived by exchanging rows for columns and columns for rows. If **A** is the matrix below,

$$\mathbf{A} \quad \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

then,

```
MAT B= TRN(A)
```

would result in:

$$\mathbf{B} \quad \begin{pmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{pmatrix}$$

A matrix does not have to be square to have a transpose. If **A** is,

$$\begin{array}{c}\mathbf{A}\\ \begin{pmatrix}0 & 1 & 8 & 3\\ 2 & 9 & 7 & -2\end{pmatrix}\end{array}$$

then,

```
MAT B= TRN(A)
```

would result in:

$$\begin{array}{c}\mathbf{B}\\ \begin{pmatrix}0 & 2\\ 1 & 9\\ 8 & 7\\ 3 & -2\end{pmatrix}\end{array}$$

The result array cannot be the same as the array being transposed. For example,

```
MAT A= TRN(A)
```

is an illegal statement and will cause an error.

The transpose of a `COMPLEX` array is done in the same manner as for `REAL` and `INTEGER` values. Suppose you have a `COMPLEX` array called `Complex_array` and an array to receive the results of a `MAT` operation called `Result_array` dimensioned as follows:

```
COMPLEX Complex_array(2,4),Result_array(4,2)
```

If the values read into the array called `Complex_array` are as given below,

$$\begin{array}{c}\textbf{Complex\_array}\\ \begin{pmatrix}0 \quad -1 & 1 \quad 3 & 8 \quad -11 & 3 \quad 6\\ 2 \quad 0 & 9 \quad -34 & 7 \quad 8 & -2 \quad 16\end{pmatrix}\end{array}$$

then executing the statement

```
MAT Result_array= TRN(Complex_array)
```

would produce the following:

$$\textbf{Result\_array}$$
$$\begin{pmatrix} 0 & -1 & 2 & 0 \\ 1 & 3 & 9 & -34 \\ 8 & -11 & 7 & 8 \\ 3 & 6 & -2 & 16 \end{pmatrix}$$

The transpose function is useful for manipulating tables of data. It also has special significance for a small set of matrices called "orthogonal" matrices. An orthogonal matrix is defined as one whose transpose and inverse are the same.

Summing Rows and Columns of a Matrix

BASIC has a function called RSUM which returns the sum of all rows in an array and a function called CSUM which returns the sum of all columns in an array. The totals are stored in a vector which RSUM and CSUM will re-dimension if necessary. Note that the DIM statement is needed in the following program because all other references to the arrays use (*) to specify the whole array. Let **A** be the matrix shown below.

$$\mathbf{A}$$
$$\begin{pmatrix} 3 & 6 & 18 & 7 \\ 1 & 0 & 41 & 2 \\ 4 & 3 & 12 & 11 \end{pmatrix}$$

If we execute:

```
10    OPTION BASE 1
20    DIM A(3,4),Row_sum(3),Col_sum(4)
30    DATA 3,6,18,7,1,0,41,2,4,3,12,11
40    READ A(*)
50    MAT Row_sum= RSUM(A)
60    MAT Col_sum= CSUM(A)
70    PRINT "The sum of rows is:  ";Row_sum(*)
80    PRINT "The sum of columns is:  ";Col_sum(*)
90    END
```

The system will display:

```
The sum of rows is:  34  44  30
The sum of columns is:  8  9  71  20
```

The following program adds the rows and columns of a 2×3 two-dimensional complex array with the following values in it:

Complex_array

$$\begin{pmatrix} 2 & 3 & & -6 & 9 & & -1 & 1 \\ -3 & -4 & & 1 & 0 & & 2 & 8 \end{pmatrix}$$

If you execute:

```
100    OPTION BASE 1
110    COMPLEX Complex_array(2,3),Sum_rows(2),Sum_columns(3)
120    !
130    DATA 2,3,-6,9,-1,1,-3,-4,1,0,2,8
140    !
150    READ Complex_array(*)
160    !
170    MAT Sum_rows= RSUM(Complex_array)
180    MAT Sum_columns= CSUM(Complex_array)
190    !
200    PRINT "The sum of rows is:  ";Sum_rows(*)
210    PRINT "The sum of columns is:  ";Sum_columns(*)
220    !
230    END
```

The following will be displayed:

```
The sum of rows is:  -5  13    0   4
The sum of columns is:  -1 -1   -5  9    1  9
```

Examples of Complex Array Operations

It is sometimes useful to be able to create a REAL array from the real or imaginary parts of a COMPLEX array, as well as from the arguments or absolute values of each element of a COMPLEX array. It is also useful to be able to create a COMPLEX array from two REAL arrays. This section describes functions which allow you to perform these tasks. The COMPLEX array used in the examples is given below:

Complex_array

$$\begin{pmatrix} -1 & -2 & \quad 3 & 5 & \quad -9 & 8 \\ 1 & 0 & \quad 2 & -7 & \quad 16 & -1 \end{pmatrix}$$

To place the real part of each element in a COMPLEX array called Complex_array into a REAL array called Array, you would use the following statement:

```
MAT Array= REAL(Complex_array)
```

which would result in the following array:

Array

$$\begin{pmatrix} -1 & 3 & -9 \\ 1 & 2 & 16 \end{pmatrix}$$

To place the imaginary part of each element in a COMPLEX array called Complex_array into a REAL array called Array, you would use the following statement:

```
MAT Array= IMAG(Complex_array)
```

which would result in the following array:

Array

$$\begin{pmatrix} -2 & 5 & 8 \\ 0 & -7 & -1 \end{pmatrix}$$

To place the argument of each element in a COMPLEX array called Complex_array into a REAL array called Array, you would use the following statement:

```
MAT Array= ARG(Complex_array)
```

which would result in the following array:

Array

$$\begin{pmatrix} -2.0344 & 1.0304 & 2.4150 \\ 0.0000 & -1.2925 & -.0624 \end{pmatrix}$$

Keep in mind that taking the ARG of an array returns values for the array elements which fall in the range of $-\pi$ to $+\pi$ for the radian mode and $-180°$ to $+180°$ for the degree mode.

To place the absolute value (or magnitude) of each element in a COMPLEX array called Complex_array into a REAL array called Array, you would use the following statement:

```
MAT Array= ABS(Complex_array)
```

which would result in the following array:

Array

$$\begin{pmatrix} 2.23617 & 5.8310 & 12.0416 \\ 1.0000 & 7.2801 & 16.0312 \end{pmatrix}$$

To create a conjugate array out of the COMPLEX array called Complex_array and place that conjugate array into the COMPLEX array called Conjugate, you would use the following statement:

```
MAT Conjugate= CONJG(Complex_array)
```

which would result in the following array:

$$\begin{array}{c}\textbf{Conjugate}\\ \begin{pmatrix} -1 & 2 & 3 & -5 & -9 & -8 \\ 1 & 0 & 2 & 7 & 16 & 1 \end{pmatrix}\end{array}$$

To create a COMPLEX array called New_comp_array from two REAL arrays called Real_array1 and Real_array2,

$$\begin{array}{cc}\textbf{Real\_array1} & \textbf{Real\_array2}\\ \begin{pmatrix} 4 & 6 & 7 \\ 5 & 9 & 1 \end{pmatrix} & \begin{pmatrix} -4 & -8 & -1 \\ -3 & -2 & -9 \end{pmatrix}\end{array}$$

you would use the following statement:

```
MAT New_comp_array= CMPLX(Real_array1,Real_array2)
```

which would result in the following array:

$$\begin{array}{c}\textbf{New\_comp\_array}\\ \begin{pmatrix} 4 & -4 & 6 & -8 & 7 & -1 \\ 5 & -3 & 9 & -2 & 1 & -9 \end{pmatrix}\end{array}$$

Using Arrays for Code Conversion

Suppose you have an input device that provides information in 8-bit ASCII code. On the other hand, an output device in the same system uses a non-ASCII specialized 8-bit code. Examples might include specialized instrumentation, typesetting equipment, or a multitude of other devices. For each ASCII character, there is a corresponding code for the output device. There may be some ASCII characters (such as control characters) that are not to be converted. Let us assume that a null character (all bits set to zero) is used for those special characters. Here is how a conversion array is set up:

1. First, an array is created with 256 elements (0 thru 255). Each element address corresponds to the 8-bit `INTEGER` numeric equivalent of the ASCII character code. The contents of a given array element contains the output code for the corresponding ASCII input code. The array can be `REAL` or `INTEGER`. Usually, it is more efficient to use `INTEGER` arrays for converting 16-bit or shorter codes. The array must be filled by individual program statements (assignments or `DATA` and `READ` statements), or it can be filled from a mass storage file. If a file is used, the data must be created by some prior means. Fixed conversion codes can sometimes be generated by an algorithm in the introductory part of the program that performs the conversions.
2. Input data is placed in a string variable (see the "String Manipulation" chapter for string variables techniques). Characters are then picked off, one character at a time, for conversion. Refer to *BASIC Interfacing Techniques* for more information about output operations.

Here is an example of how such an operation could be implemented:

```
1000   INTEGER Convert(0:255)
1010   DIM In$[80]
1020   Source=18      ! Source device selector
1030   Dest=22        ! Destination device selector
         .
         .
         .
 Initialize the conversion array here.
         .
         .
         .
2470   ENTER Source;Input$    ! Input line of ASCII
2480   FOR I=1 TO LEN(In$)   ! Send converted bytes
2490     OUTPUT Dest;CHR$(Convert(NUM(In$[I,I])));
2500   NEXT I
```

Note that the semicolon in line 2490 prevents sending a carriage-return and line-feed character pair at the end of each output line. This is usually necessary to prevent unwanted behavior when using ASCII strings to output non-ASCII data. This technique can be applied to arbitrary data conversions with virtually no limitations.

It is also possible to handle code conversions automatically in OUTPUT statements with the CONVERT options of the ASSIGN statement. See the ASSIGN Attributes discussion in *BASIC Interfacing Techniques.*

String Manipulation 5

String Manipulation 5

It is often desirable to store non-numerical information in the computer. A word, a name or a message can be stored in the computer as a **string**. Any sequence of characters may be used in a string. Quotation marks are used to delimit the beginning and ending of the string. The following are valid string assignments.

```
LET A$="COMPUTER"
Fail$="The test has failed."
File_name$="INVENTORY"
Test$=Fail$[5,8]
```

The left-hand side of the assignment (the variable name) is equated to the right-hand side of the assignment (the literal).

String variable names are identical to numeric variable names with the exception of a dollar sign ($) appended to the end of the name.

The **length** of a string is the number of characters in the string. In the previous example, the length of A$ is 8 since there are eight characters in the literal "COMPUTER". A string with length 0 (i.e., that contains no characters) is known as a "null" string.

BASIC allows the dimensioned length of a string to range from 1 to 32 767 characters and the current length (number of characters in the string) to range from zero to the dimensioned length. A string of zero characters is often called a null string or an empty string.

The default dimensioned length of a string is 18 characters. The DIM, COM, and ALLOCATE statements are used to define string lengths up to the maximum length of 32 767 characters. An error results whenever a string variable is assigned more characters than its dimensioned length.

A string may contain any character. The only special case is when a quotation mark needs to be in a string. Two quotes, in succession, will embed a quote within a string.

```
10  Quote$="The time is ""NOW""."
20  PRINT Quote$
30  END
```

Produces: `The time is "NOW".`

String Storage

Strings whose length exceeds the default length of 18 characters must have space reserved before assignment. The following statements may be used.

- `DIM Long$[400]` Reserve space for a 400 character string.
- `COM Line$[80]` Reserve an 80 character common variable.
- `ALLOCATE Search$[Length]` Dynamic length allocation.

The maximum length of any string must not exceed 32 767 characters. A string may also be dimensioned to a length less than the default length of 18 characters.

The DIM statement reserves storage for strings.

```
DIM Part_number$[10],Description$[64],Cost$[5]
```

The COM statement defines common variables that can be used by subprograms.

```
COM Name$[40],Phone$[14]
```

The ALLOCATE statement allows dynamic allocation of string storage. When the maximum length of a string cannot be determined ahead of time, the ALLOCATE statement can be used to reserve enough memory space for the string without wasting space.

```
ALLOCATE Line$[Length]
```

Strings that have been dimensioned but not assigned return the null string.

String Arrays

Large amounts of text are easily handled in arrays. For example:

```
DIM File$(1000)[80]
```

This statement reserves storage for 1000 lines of 80 characters per line. Do not confuse the brackets, which define the length of the string, with the parentheses which define the number of strings in the array. Each string in the array can be accessed by an index. For example:

```
PRINT File$(27)
```

Prints the 27th element in the array. Since each character in a string uses one byte of memory and each string in the array requires as many bytes as the length of the string, string arrays can quickly use a lot of memory.

A program saved on a disc as an ASCII type file can be entered into a string array, manipulated, and written back out to disc.

Evaluating Expressions Containing Strings

This section covers the following topics:

- Evaluation Hierarchy
- String Concatenation
- Relational Operations

Evaluation Hierarchy

Evaluation of string expressions is simpler than evaluation of numerical expressions. The three allowed operations are extracting a substring, concatenation, and parenthesization. The evaluation hierarchy is presented in the following table.

Order	Operation
High	Parentheses
—	Substrings and Functions
Low	Concatenation

String Concatenation

Two separate strings are joined together by using the oncatenation operator "&". The following program combines two strings into one.

```
10 One$="WRIST"
20 Two$="WATCH"
30 Concat$=One$&Two$
40 PRINT One$,Two$,Concat$
50 END
```

Prints:

```
WRIST     WATCH     WRISTWATCH
```

The concatenation operation, in line 30, appends the second string to the end of the first string. The result is assigned to a third string. An error results if the concatenation operation produces a string that is longer than the dimensioned length of the string being assigned.

Relational Operations

Most of the relational operators used for numeric expression evaluation can also be used for the evaluation of strings.

The following examples show some of the possible tests.

`"ABC" = "ABC"`	True
`"ABC" = " ABC"`	False
`"ABC" < "AbC"`	True
`"6" > "7"`	False
`"2" < "12"`	False
`"long" <= "longer"`	True
`"RE-SAVE" >= "RESAVE"`	False

Any of these relational operators may be used: <, >, <=, >=, =, <>.

Testing begins with the first character in the string and proceeds, character by character, until the relationship has been determined.

The outcome of a relational test is based on the characters in the strings not on the length of the strings. For example:

```
"BRONTOSAURUS" < "CAT"
```

This relationship is true since the letter "C" is higher in ASCII value than the letter "B".

NOTE

When the `LEX` binary is loaded, the outcome of a string comparison is based on the character's lexical value rather than the character's ASCII value. See the `LEXICAL ORDER IS` statement later in this chapter for more details.

Substrings

A subscript can be appended to a string variable name to define a **substring**. A substring may comprise all or just part of the original string. Brackets enclose the subscript which can be a constant, variable, or numeric expression. For instance:

`String$[4]` Specifies a substring starting with the fourth character of the original string.

The subscript must be in the range: 1 to the current length of the string plus 1. Note that the brackets now indicate the substring's starting position instead of the total length of the string as when reserving storage for a string.

Subscripted strings may appear on either side of the assignment.

Single-Subscript Substrings

When a substring is specified with only one numerical expression, enclosed with brackets, the expression is evaluated and rounded to an integer indicating the position of the first character of the substring within the string.

The following examples use the variable A$ which has been assigned the literal "DICTIONARY".

Statement	Output
`PRINT A$`	`DICTIONARY`
`PRINT A$[0]`	(error)
`PRINT A$[1]`	`DICTIONARY`
`PRINT A$[5]`	`IONARY`
`PRINT A$[10]`	`Y`
`PRINT A$[11]`	(null string)
`PRINT A$[12]`	(error)

When a single subscript is used it specifies the starting character position, within the string, of the substring. An error results when the subscript evaluates to zero or greater than the current length of the string plus 1. A subscript that evaluates to 1 plus the length of the string returns the null string ("") but does not produce an error.

Double-Subscript Substrings

A substring may have two subscripts, within brackets, to specify a range of characters. When a comma is used to separate the items within brackets, the first subscript marks the beginning position of the substring, while the second subscript is the ending position of the substring. The form is: A$[Start,End]. For example, if A$ = "JABBERWOCKY", then

A$[4,6] Specifies the substring: BER

When a semicolon is used in place of a comma, the first subscript again marks the beginning position of the substring, while the second subscript is now the length of the substring. The form is: A$[Start;Length].

A$[4;6] Specifies the substring: BERWOC

In the following examples the variable B$ has been assigned the literal "ENLIGHTENMENT":

Statement	Output
`PRINT B$`	`ENLIGHTENMENT`
`PRINT B$[1,13]`	`ENLIGHTENMENT`
`PRINT B$[1;13]`	`ENLIGHTENMENT`
`PRINT B$[1,9]`	`ENLIGHTEN`
`PRINT B$[1;9]`	`ENLIGHTEN`
`PRINT B$[3,7]`	`LIGHT`
`PRINT B$[3;7]`	`LIGHTEN`
`PRINT B$[13,13]`	`N`
`PRINT B$[13;1]`	`N`
`PRINT B$[13,26]`	(error)
`PRINT B$[13;13]`	(error)
`PRINT B$[14;1]`	(null string)

An error results if the second subscript in a comma separated pair is greater than the current string length plus 1 **or** if the sum of the subscripts in a semicolon separated pair is greater than the current string length plus 1.

Specifying the position just past the end of a string returns the null string.

Special Considerations

All substring operations allow a subscript to specify the first position past the end of a string. This allows strings to be concatenated without the concatenation operator. For instance:

```
10    A$="CONCAT"
20    A$[7]="ENATION"
30    PRINT A$
40    END
```

Produces: `CONCATENATION`

The substring assignment is only valid if the substring already has characters up to the specified position. Access beyond the first position past the end of a string results in the error:

```
ERROR 18  String ovfl. or substring err
```

A good practice is to dimension all strings including those shorter than the default length of eighteen characters.

Some very interesting assignments can be attempted. For example, a 14-character string can be assigned to a 3-character substring.

```
10    Big$="Too big to fit"
20    Small$="Little string"
30    !
40    Small$[1,3]=Big$
50    !
60    PRINT Small$
70    END
```

Prints: `Tootle string`

When a substring assignment specifies fewer characters than are available, any extra trailing characters are truncated.

The alternate assignment is shown in the next example. Here a 4-character string is assigned to a 8-character substring.

```
10    Big$="A large string"
20    Small$="tiny"
30    !
40    Big$[3,10]=Small$
50    !
60    DISP Big$
70    END
```

Prints: `A tiny    ring`

Since the subscripted length of the substring is greater than the length of the replacement string, enough blanks (ASCII spaces) are added to the end of the replacement string to fill the entire specified substring.

String-Related Functions

Several intrinsic functions are available in BASIC for the manipulation of strings. These functions include conversions between string and numeric values.

Current String Length

The "length" of a string is the number of characters in the string. The LEN function returns an integer whose value is equal to the string length. The range is from 0 (null string) thru 32 767. For example:

```
PRINT LEN("HELP ME")
```

Prints: 7

The following example program prints the length of a string that is typed on the keyboard.

```
10    DIM In$[160]
20    INPUT In$
30    Length=LEN(In$)
40    DISP Length;"characters in """;In$;""""
50    END
```

Try finding the length of a string containing only spaces. When the INPUT statement is used, any leading or trailing spaces are removed from items typed on the keyboard. Change INPUT to LINPUT in line 20 to allow leading and trailing spaces to be entered.

Maximum String Length

The MAXLEN function returns an integer whose value is equal to the dimensioned length of a string variable. For example,

```
100   DIM First_string$[37],Second_string$(2)[15]
110   PRINT "Maximum length of the first string is";
120   PRINT MAXLEN(First_string$)
130   PRINT
140   PRINT "Maximum length of the second string is";
150   PRINT MAXLEN(Second_string$(1))
160   Test("A TEST STRING")
170   END
180   SUB Test(A$)
190     PRINT
200     PRINT "Maximum length of the test string is";
210     PRINT MAXLEN(A$)
220   SUBEND
```

The above program produces the following results:

```
Maximum length of the first string is 37

Maximum length of the second string is 15

Maximum length of the test string is 13
```

Substring Position

The "position" of a substring within a string is determined by the POS function. The function returns the value of the starting position of the substring or zero if the entire substring was not found. For instance:

```
PRINT POS("DISAPPEARANCE","APPEAR")
```

Prints: 4

The following example prints the positions of substrings found within a string.

```
10    DIM Sentence$[40],Word$(1:6)[8]
20    DATA CAT,ON,A,HOT,TIN,NATION
30    READ Word$(*)
40    Sentence$="WHERE IS THE CAT IN CONCATENATION"
50    !

60    FOR I=1 TO 6
70      Position=POS(Sentence$,Word$(I))    ! <- POS function
80      IF Position THEN
90        PRINT Sentence$
100       PRINT TAB(Position);Word$(I);TÂB(35);"is at ";Position
110       PRINT
120     ELSE
130       PRINT "'";Word$(I);"' was not found"
140       PRINT
150     END IF
160   NEXT I
170   END
```

If POS returns a non-zero value, the entire substring occurs in the first string and the value specifies the starting position of the substring.

Note that POS returns the first occurrence of a substring within a string. By adding a subscript, and indexing through the string, the POS function can be used to find all occurrences of a substring. The following program uses this technique to extract each word from a sentence.

```
10    DIM A$[80]
20    A$="I know you think you understand what I said, but you don't."
30    INTEGER Scan,Found
40    Scan=1                            ! Current substring position
50    PRINT A$
60    REPEAT
70      Found=POS(A$[Scan]," ")         ! Find the next ASCII space
80      IF Found THEN
90        PRINT A$[Scan,Scan+Found-1]   ! Print the word
100       Scan=Scan+Found               ! Adjust "Scan" past last match
110     ELSE
120       PRINT A$[Scan]                ! Print last word in string
130     END IF
140   UNTIL NOT Found
150   END
```

As each occurrence is found, the new subscript specifies the remaining portion of the string to be searched.

String-to-Numeric Conversion

The VAL function converts a string expression into a numeric value. The string must evaluate to a valid number or error 32 will result.

```
ERROR 32 String is not a valid number
```

The number returned by the VAL function will be converted to and from scientific notation when necessary. For example:

```
PRINT VAL("123.4E3")
```

Prints: 123400

The following program converts a fraction into its equivalent decimal value.

```
10    INPUT "Enter a fraction  (i.e. 3/4)",Fraction$
20    !
30    ON ERROR GOTO Err
40      Numerator=VAL(Fraction$)
50      !
60      IF POS(Fraction$,"/") THEN
70        Delimiter=POS(Fraction$,"/")
80        Denominator=VAL(Fraction$[Delimiter+1])
90      ELSE
100       PRINT "Invalid fraction"
110       GOTO Quit
120     END IF
130     !
140     PRINT Fraction$;" = ";Numerator/Denominator
150     GOTO Quit
160 Err: PRINT "ERROR Invalid fraction"
170      OFF ERROR
180 Quit:   END
```

Similar techniques can be used for converting: feet and inches to decimal feet or hours and minutes to decimal hours.

The NUM function converts a single character into its equivalent numeric value. The number returned is in the range: 0 to 255. For example:

```
PRINT NUM("A")
```

Prints: 65

The next program prints the value of each character in a name.

```
10    INPUT "Enter your first name",Name$
20    !
30    PRINT Name$
40    PRINT
50    FOR I=1 TO LEN(Name$)
60      PRINT NUM(Name$[I]);  ! Print value of each character
70    NEXT I
80    PRINT
90    END
```

Entering the name: JOHN will produce the following.

```
74  79  72  78
```

Numeric-to-String Conversion

The VAL$ function converts the value of a numeric expression into a character string. The string contains the same characters (digits) that appear when the numeric variable is printed. For example:

```
PRINT 1000000,VAL$(1000000)
```

Prints: `1.E+6    1.E+6`

The next program converts a number into a string so the POS function can be used to separate the mantissa from the exponent.

```
10     CONTROL 2,0;1 ! CAPS LOCK ON
20     INPUT "Enter a number with an exponent",Number
30     !
40     Number$=VAL$(Number)
50     !
60     PRINT Number$
70     E=POS(Number$,"E")
80     IF E THEN
90       PRINT "Mantissa is",Number$[1;E-1]
100      PRINT "Exponent is",Number$[E+1]
110    ELSE
120      PRINT "No exponent"
130    END IF
140    END
```

The CHR$ function converts a number into an ASCII character. The number can be of type INTEGER or REAL since the value is rounded, and a modulo 255 is performed. For example:

```
PRINT CHR$(97);CHR$(98);CHR$(99)
```

Prints: `abc`

The next program prints the values in the data statement as characters.

```
10     PRINT CHR$(12)   ! CLEAR SCREEN
20     PRINT CHR$(7)    ! RING THE BELL
30     !
```

```
40    DATA 34,130,89,111,117,32,103,111,116,32,105,116,33,128,34
50    INTEGER N(1:15)
60    READ N(*)
70    FOR I=1 TO 15
80      PRINT CHR$(N(I));
90    NEXT I
100   PRINT CHR$(7)
110   END
```

CRT Character Set

The following program prints the character set on the screen of the CRT to show the order that strings will be sorted.

```
10    ! Program: CRT Character Set.
20    !
30    PRINT CHR$(12);"CRT Character Set"
40    STATUS 1,9;Line_length ! 50, 80, or 128 Columns
50    Left=Line_length/2-16
60    !
70    FOR I=0 TO 255
80      Col=I MOD 16*2+Left
90      Row=I DIV 16+3
100     IF Col=Left THEN
110       PRINT TABXY(Left-5,Row);
120       PRINT USING "3D";I
130     END IF
140     PRINT TABXY(Col,Row);
150     CONTROL 1,4;1       ! Display Functions on
160     PRINT USING "B,B,#";128,I ! Print the Character
170     CONTROL 1,4;0       ! Display Functions off
180   NEXT I
190   PRINT
200   I=127
210   ON KNOB .08 GOSUB Change
220   DISP USING "5A,5D,X,2A,B,B";"ASCII",I,"=",128,I
230   GOTO 220
240 Change:   I=I-KNOBX/10
250           IF I<0 THEN I=0
260           IF I>255 THEN I=255
270           RETURN
280   END
```

ASCII character values from 128 to 159 are treated differently by different systems. Refer to the section "The Extended Character Set" found this chapter.

String Functions

This section covers string functions which perform the following tasks:

- Reversing the characters in a string,
- Repeating a string a given number of times,
- Trimming the leading and trailing blanks in a string,
- Converting string characters to the desired case.

String Reverse

The **REV$** function returns a string created by reversing the sequence of characters in the given string.

```
PRINT REV$("Snack cans")
```

Prints: `snac kcanS`

A common use for the **REV$** function is to find the last occurrence of an item in a string.

```
10    DIM List$[30]
20    List$="3.22 4.33 1.10 8.55 12.20 1.77"
30    Length=LEN(List$)
40    Last_space=POS(REV$(List$)," ")   ! "SPACE" is delimiter
50    DISP "The last item is:";List$[1+Length-Last_space]
60    END
```

Displays: `The last item is: 1.77`

String Repeat

The RPT$ function returns a string created by repeating the specified string, a given number of times.

```
PRINT RPT$("* *",10)
```

Prints: * ** ** ** ** ** ** ** ** ** *

Here is a short program that uses RPT$ to create an image for a formatted print statement.

```
10    Items=7
20    DATA 50,900,2,444,37,2001,32768
30    ALLOCATE Array(1:Items)
40    READ Array(*)
50    FOR I=1 TO Items
60      Digits=INT(1+LGT(Array(I)))
70      IF Digits>Maxdigits THEN Maxdigits=Digits
80    NEXT I
90    Form$="XX,"&RPT$("D",Maxdigits)&".DD"
100   PRINT "Using the image: ";Form$
110   PRINT USING Form$;Array(*)
120   END
```

Trimming a String

The TRIM$ function returns a string with all leading and trailing blanks (ASCII spaces) removed.

```
PRINT "*";TRIM("    1.23     ");"*"
```

Prints: *1.23*

TRIM$ is often used to extract fields from data statements or keyboard input.

```
10    INPUT "Enter your full name",Name$
20    First$=TRIM$(Name$[1,POS(Name$," ")])
30    Last$=TRIM$(Name$[1+LEN(Name$)-POS(REV$(Name$)," ")])
40    PRINT Name$,LEN(Name$)
50    PRINT Last$,LEN(Last$)
60    PRINT First$,LEN(First$)
70    END
```

Note that the INPUT statement trims leading and trailing blanks from whatever is typed. If you need to enter leading or trailing spaces, use the LINPUT statement.

Case Conversion

The case conversion functions, UPC$ and LWC$, return strings with all characters converted to the proper case. UPC$ converts all lowercase characters to their corresponding uppercase characters and LWC$ converts any uppercase characters to their corresponding lowercase characters. Roman Extension characters will be converted according to the current lexical order. See the LEXICAL ORDER IS statement later in this chapter for the case conversion listings.

```
10    DIM Word$[160]
20    LINPUT "Enter a few characters",Word$
30    PRINT
40    PRINT "You typed: ";Word$
50    PRINT "Uppercase: ";UPC$(Word$)
60    PRINT "Lowercase: ";LWC$(Word$)
70    END
```

A more general character replacement method is obtained by using a buffer that was assigned an indexed conversion. Indexed conversion uses the incoming character's ASCII value as an index into a string of characters and returns the character in that position. In the following program, the conversion string is created in **line 40** and **60**. The conversion string specifies all lowercase characters are to be replaced by their corresponding uppercase character.

```
10     DIM Cipher$[256],A$[80]
20     FOR I=1 TO 255            ! Create conversion string
30       Cipher$=Cipher$&UPC$(CHR$(I))
40     NEXT I
50     Cipher$=Cipher$&UPC$(CHR$(0))
60     ASSIGN @F TO BUFFER [160];CONVERT OUT BY INDEX Cipher$
70     LOOP
80       INPUT A$
90       OUTPUT @F;A$            ! Conversion occurs
100      ENTER @F;A$
110      PRINT A$
120    END LOOP
130    END
```

Copying String Arrays and Subarrays

MAT functions (available with the MAT binary) are commonly used to manipulate data in numeric arrays. However, several of these functions can be used with string arrays. For example, a string array is copied into another string array by the following.

```
MAT Copy$ = Original$
```

Note that only the variable name is necessary. The array specifier "(*)" is not included when using the MAT statement.

Every element in a string array will be initialized to a constant value by the following statement.

```
MAT Array$ = (Null$)
```

The constant value can be a literal or a string expression and is enclosed in parentheses to distinguish it from an array name.

A subarray can be copied into another subarray of the same size and shape. For example, suppose you want to copy the string elements in a two-dimensional string array found in rows 1 through 3 and columns 5 and 6 of the string array called **Sub_array$** into the array called New$, you would execute the following statement:

```
MAT New$= Sub_array$(1:3,5:6)
```

where the above statement assumes an OPTION BASE of 1 and that New$ is dimensioned to be a 3×2 string array.

For more information on copying numeric and string arrays see the MAT statement in the *BASIC Language Reference.*

Searching and Sorting

Information stored in a string array often requires sorting. There are over a dozen common algorithms that may be used. Each algorithm has certain advantages depending on the number of items to be sorted, the current order of the items, the time allowed to sort the items, and the complexity of the algorithm. One of the simplest (and most inefficient) sorts to implement is the "bubble" sort. The following program is a slight variation of the bubble sort.

```
10    ! Program: SORT
20    !
30    READ N
40    DATA 10   ! NUMBER OF ITEMS TO SORT
50    ALLOCATE Word$(N)[5],Temp$[5]
60    READ Word$(*)   ! READ ENTIRE ARRAY
70    DATA zero,one,two,three,four,five,six,seven,eight,nine,ten
80    PRINT Word$(*)
90    PRINT
100 Sort:FOR I=0 TO N-1
110        IF Word$(I)>Word$(I+1) THEN
120          Temp$=Word$(I)
130          Word$(I)=Word$(I+1)
140          Word$(I+1)=Temp$
150          GOTO Sort
160        END IF
170      NEXT I
180   PRINT Word$(*)
190   END
```

This example prints the contents of the array before and after sorting.

Before sorting:

```
zero      one       two       three     four      five
six       seven     eight     nine      ten
```

After sorting:

```
eight     five      four      nine      one       seven
six       ten       three     two       zero
```

The strings are sorted in ascending order. If the relational operator in line 110 is changed from the greater than sign ">" to the less than sign "<", the strings will be sorted in descending order.

A list of items can be sorted very quickly by the MAT SORT statement.

```
10    ! Program: SORT_LIST
20    DIM List$(1:5)[6]
30    DATA Bread,Milk,Eggs,Bacon,Coffee
40    READ List$(*)
50    !
60    PRINT "original order"
70    PRINT List$(*)
80    !
90    PRINT "ascending order"
100   MAT SORT List$
110   PRINT List$(*)
120   !
130   PRINT "descending order"
140   MAT SORT List$ DES
150   PRINT List$(*)
160   END
```

Running this program produces:

```
original order
Bread    Milk     Eggs     Bacon    Coffee

ascending order
Bacon    Bread    Coffee   Eggs     Milk

descending order
Milk     Eggs     Coffee   Bread    Bacon
```

Sorting by Substrings

A substring range can be appended to the end of a `MAT SORT` *key specifier*. For example, to sort the entire first column of a two-dimensional string array called **`Str_ary$`** using the 3rd and 4th characters of each string, you would use this *key specifier*: `(*,1)[3,4]`. The MAT SORT statement would be as follows:

```
MAT SORT Str_ary$(*,1)[3,4]
```

Items will then be sorted by the characters within the substring specified. No error results from specifying a substring position beyond the current length of the string.

```
10    PRINT CHR$(12)     ! Program: SUBSORT
20    DATA 1 OLD  ORANGE,2 TINY TOADS,3  TALL TREES,4 FAT  FOWLS,5 FRIED FISH
30    DATA 6 SLOW SNAILS,7 SLIMY SLUGS,8 AWFUL HOURS,9 NASTY KNIVES
40    DIM Things$(1:9)[38]
50    READ Things$(*)
60    First=1
70    Length=1
80    DISP "Use KNOB and SHIFT-KNOB to change sort field."
90    ON KNOB .2 GOTO Slide
100 Go:MAT SORT Things$(*)[First;Length]
110   FOR I=1 TO 9
120     PRINT TABXY(10,I);Things$(I);RPT$(" ",3)
130   NEXT I
140 W:GOTO W
150   !
160 Slide:STATUS 2,10;Shift       ! Check for SHIFT OR CTRL
170   S=SGN(KNOBX)
180   IF Shift THEN
190     Length=Length+S*(S>0 AND Length<16)+S*(S<0 AND Length>1)
200   ELSE
210     First=First+S*(S>0 AND First<18)+S*(S<0 AND First>1)
220   END IF
230   DISP "MAT SORT Things$(*)[";First;";";Length;"]"
240   PRINT TABXY(9,10);RPT$(" ",First);RPT$("^",Length);RPT$(" ",10)
250   GOTO Go
260   END
```

Adding Items to a Sorted List

Lists of strings can be maintained in sorted order. Every time a new item is added to the list, the list is sorted by the MAT SORT statement. To prevent overwriting any of the items already in the list, items should be added to the top (first array element) of a list sorted in ascending order and to the bottom (last array element) of a list sorted in descending order.

```
10    PRINT CHR$(12)
20    ! Since arrays are in COM, they "remember" old values.
30    ! After running, execute SCRATCH C to clear the arrays.
40    !
50    COM Ascend$(1:18)[18],Descend$(1:18)[18]
60 Again:I=I+1
70    INPUT "Enter a word",Word$
80    Ascend$(1)=Word$             ! Fill array at top
90    Descend$(18)=Word$           ! Fill array at bottom
100   CALL See
110   IF I<18 THEN Again
120   BEEP
130   END
140   !--------------------------------------------------
150   SUB See                      ! DISPLAY THE ARRAYS
160     COM Ascend$(*),Descend$(*)
170     MAT SORT Ascend$           ! <- ascending sort
180     MAT SORT Descend$ DES      ! <- descending sort
190     FOR J=1 TO 18
200       PRINT TABXY(1,J);RPT$(" ",49)
210       PRINT TABXY(1,J);J;TABXY(11,J);Ascend$(J);TABXY(31,J);Descend$(J)
220     NEXT J
230   SUBEND
```

Sorting by Multiple Keys

When sorting a multi-dimensional numeric or string array, it is possible to specify more than one key. The array will be sorted by the first key then the second key and so on until the key specifiers are exhausted. Once the first key sorts items into similar groups, the items within a group can be arranged in any order you choose.

```
10    COM Tool$(1:8,1:3)[10]
20    DATA PENCIL,RED,35,PENCIL,BLUE,12,PENCIL,GREEN,0,PENCIL,BLACK,17
30    DATA PEN,BLACK,17,PEN,BLUE,127,PEN,RED,55,PEN,GREEN,43
40    READ Tool$(*)
50    PRINT
60    PRINT "*** UNSORTED LIST ***"
70    Display
80    PRINT "*** SORT BY COLOR ***"
90    MAT SORT Tool$(*,2)[1,3]       ! Sort color by first three letters.
100   Display
110   PRINT "*** SORT BY COLOR THEN BY NAME ***"
120   MAT SORT Tool$(*,2),(*,1)      ! Two key sort.
130   Display
140   PRINT "*** SORT BY NAME THEN BY COLOR ***"
150   MAT SORT Tool$(*,1),(*,2)[1;3] DES
160   Display
170   END
180   !----------------------
190   SUB Display
200     COM Tool$(*)
210     K=K+1
220     FOR I=1 TO 8
230       FOR J=1 TO 3
240         PRINT Tool$(I,J),
250       NEXT J
260       PRINT
270     NEXT I
280   SUBEND
```

Sorting to a Vector

It is possible to determine the sorting order of items in an array without disturbing the array. This is accomplished by "sorting" to a single-dimensioned numeric array (vector). The vector will then contain the subscripts of the items in the order that the items would have been arranged.

```
10    DIM Month$(1:12)[3],Fix(1:12)
20    DATA JAN,FEB,MAR,APR,MAY,JUN,JUL,AUG,SEP,OCT,NOV,DEC
30    READ Month$(*)
40    MAT SORT Month$ TO Fix    ! Sort to vector
50    PRINT Month$(*)
60    PRINT Fix(*)
70    FOR I=1 TO 12
80      PRINT Month$(Fix(I)),   ! Print months alphabetically
90    NEXT I
100   END
```

Running this program produces:

```
JAN  FEB  MAR  APR  MAY  JUN  JUL  AUG  SEP  OCT  NOV  DEC

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

APR  AUG  DEC  FEB  JAN  JUL  JUN  MAR  MAY  NOV  OCT  SEP
```

The first element of the vector contains a four (4), indicating the fourth element in the array would be the first element if the array were actually sorted.

Reordering an Array

The rows and columns of multiple dimension arrays can be reordered. Reordering is made according to a reorder vector (single dimension array). The vector contains the values of the subscripts of the array. When the array is reordered, the columns (or rows) are arranged according to the the order of the subscripts in the reorder vector. See the following program for an example of reordering.

```
10    PRINT CHR$(12);       ! SORT_DEMO
20    DIM Size$(0:1)[5],Color$(0:2)[5],Shape$(0:1)[5]
30    COM Ident$(0:3)[5],Array$(0:3,0:11)[6],Order(0:3),Field,Down
40    DATA COUNT,SIZE,COLOR,SHAPE
50    DATA small,large,blue,red,green,cube,ball,1,2,3,0
60    READ Ident$(*),Size$(*),Color$(*),Shape$(*),Order(*)
70    FOR I=0 TO 11
80      Array$(0,I)=RPT$(" ",I<9)&VAL$(I+1)
90      Array$(1,I)=Size$(I DIV 6)
100     Array$(2,I)=Color$(I DIV 2 MOD 3)
110     Array$(3,I)=Shape$(I MOD 2)
120   NEXT I
130   ON KBD CALL Do_key
140 Again:D$=" Ascending"
150   IF Down THEN D$="Descending"
160   DISP D$;" sort on field #";Field+1
170   Sort
180   Display
190   GOTO Again
200   END
210   !---------------------------------------------
```

```
220   SUB Display
230     COM Ident$(*),Array$(*),Order(*),Field,Down
240     PRINT TABXY(1,1);
250     PRINT "Press: A for ascending sort"
260     PRINT "       D for descending sort"
270     PRINT "       R to reorder array"
280     PRINT "     1-4 for sort field";TABXY(1,5)
290     PRINT USING "#,3X,5A";Ident$(*)
300     FOR I=0 TO 11
310       PRINT TABXY(1,I+7);
320       FOR J=0 TO 3
330         PRINT USING "#,3X,5A";Array$(J,I)
340       NEXT J
350     NEXT I
360   SUBEND
370   !---------------------------------------------
380   SUB Sort
390     COM Ident$(*),Array$(*),Order(*),Field,Down
400     IF Down THEN
410       MAT SORT Array$(Field,*) DES
420     ELSE
430       MAT SORT Array$(Field,*)
440     END IF
450   SUBEND
460   !---------------------------------------------
470   SUB Do_key
480     COM Ident$(*),Array$(*),Order(*),Field,Down
490     Key$=KBD$
500     SELECT Key$
510     CASE "1" TO "4"
520       Field=VAL(Key$)-1
530     CASE "A","a"
540       Down=0
550     CASE "D","d"
560       Down=1
570     CASE "R","r"
580       MAT REORDER Array$ BY Order
590       MAT REORDER Ident$ BY Order
600     CASE ELSE
610       BEEP
620     END SELECT
630   SUBEND
```

Searching for Strings

The following program outlines a method for replacing a word in a string.

```
100   ! Program: Word_Replace
110   !
120   DIM Text$[80]
130   !
140   Search$="bad"
150   Replace$="good"
160   Text$="I am a bad string."
170   !
180   PRINT Text$
190   S_length=LEN(Search$)
200   Position=POS(Text$,Search$)
210   IF NOT Position THEN Quit
220   !
230   Text$=Text$[1,Position-1]&Replace$&Text$[Position+S_length]
240   !
250   PRINT Text$
260 Quit:   END
```

Print:
```
I am a bad string.
I am a good string.
```

Large groups of strings are usually maintained in arrays. Searching an array for a particular value is shown in the following example.

```
100   OPTION BASE 1
110   DIM List$(4)[20]
120   INTEGER I
130   DATA BLACK  BILL  $100.00
140   DATA BROWN  JEFF  $150.00
150   DATA GREEN  JIM   $200.00
160   DATA WHITE  WILL  $125.00
170   READ List$(*)
180   PRINT USING "20A,/";List$(*)
190   I=1
200   LOOP
210   EXIT IF I>4
220   EXIT IF List$(I)[1,5]="BROWN"
230     I=I+1
240   END LOOP
250   !
260   IF I<=4 THEN PRINT List$(I)[1,5];": ";List$(I)[14,17]
270   END
```

Results:

```
BLACK  BILL  $100.00
BROWN  JEFF  $150.00
GREEN  JIM   $200.00
WHITE  WILL  $125.00

BROWN: $150
```

It is often necessary to find the minimum and maximum values in a string array. The following program illustrates one method.

```
100   OPTION BASE 1
110   INTEGER I,Items
120   Items=5
130   ALLOCATE String_search$(Items)[3]
140   DATA ABC,BCD,CDE,DEF,EFG
150   READ String_search$(*)
160   !
170   Max$=String_search$(1) ! Start with first item for max.
180   Min$=Max$              ! Assume same item is min.
190   FOR I=2 TO Items
200     IF Max$<String_search$(I) THEN Max$=String_search$(I)
210     IF Min$>String_search$(I) THEN Min$=String_search$(I)
220   NEXT I
230   PRINT "The maximum array value is ";Max$;".  ";
240   PRINT "The minimum array value is ";Min$;"."
250   END
```

Results:

```
The maximum array value is EFG.  The minimum array value is ABC.
```

Searching String Arrays

Searching string arrays is similar to searching numeric arrays. For example, assume array `List$` contains a list of names and dollar amounts. The program shown next puts the data into the source array (`List$`). It then searches for a particular name and outputs the corresponding dollar amount.

```
100   OPTION BASE 1                 ! Select option base.
110   DIM List$(4)[20]              ! Dimension source array.
120   DATA BLACK  BILL  $100.00,BROWN  JEFF  $150.00
130   DATA GREEN  JIM   $200.00,WHITE  WILL  $125.00
140   READ List$(*)                 ! Read data into List$.
150   PRINT USING "20A,/";List$(*) ! Output the original list.
160   MAT SEARCH List$(*)[1,5],LOC("BROWN");Person ! Search proper
170   !                        portion of each string in List$ for a
180   !                        particular person.
190   PRINT
200   IF Person<=4 THEN
210     PRINT List$(Person)[1,5];": ";List$(Person)[13,20] ! Output
220     !                           specified name and dollar amount.
230   END IF
240   END
```

In this program a `MAT SEARCH` is used to find the string which contains the required name. Once that string is found, the portion of it containing the dollar amount is displayed. Note that the substring specifier is used in the search and display statements. If you run this program, the following results are obtained.

```
BLACK  BILL  $100.00
BROWN  JEFF  $150.00
GREEN  JIM   $200.00
WHITE  WILL  $125.00

BROWN:  $150
```

MAX and MIN values can also be obtained from a string search as demonstrated by the program shown next.

```
100   OPTION BASE 1                      ! Select option base.
110   DIM String_search$(5)[3]           ! Dimension the strings array.
120   DATA "DEF","BCD","ABC","CDE","EFG"
140   READ String_search$(*)             ! Read data into the array.
150   MAT SEARCH String_search$,MAX;Max_value$ ! Search the strings
160   !                            array for the maximum string value.
170   MAT SEARCH String_search$,MIN;Min_value$ ! Search the strings
180   !                            array for the minimum string value.
190   ! The following statements output the results of the search.
200   PRINT "The maximum array value is ";Max_value$;".  ";
210   PRINT "The minimum array value is ";Min_value$;"."
220   END
```

The array String_search$ is filled with string data. MAT SEARCH statements are then used to find the maximum and minimum values of the data. If this program is run, the following results are obtained.

```
The maximum array value is EFG.  The minimum array value is ABC.
```

Number-Base Conversion

Utility functions are available to simplify the calculations between different number bases. The two functions IVAL and DVAL convert a binary, octal, decimal, or hexadecimal string value into a decimal number. The IVAL$ and DVAL$ functions convert a decimal number into a binary, octal, decimal, or hexadecimal string value. The IVAL and IVAL$ functions are restricted to the range of INTEGER variables (-32 768 thru 32 767). The DVAL and DVAL$ functions allow "double length" integers and thus allow larger numbers to be converted (-2 147 483 648 thru 2 147 483 647).

If you are familiar with binary notation, you will probably recognize the fact that IVAL and IVAL$ operate on 16-bit values while DVAL and DVAL$ operate on 32-bit values.

```
10    PRINT CHR$(12)
20    DIM Radix$(1:4)[7],Radix(1:4),V$[33]
30    DATA  Binary,Octal,Decimal,Hex,2,8,10,16
40    READ Radix$(*),Radix(*)
50    R=3                                  ! Default to decimal mode
60    ON KEY 5 LABEL "NEW RADIX" GOTO Radix
70    ON KBD GOTO Key
80 Erase:V$=""
90    V=0
100 See:FOR I=1 TO 4
110     PRINT TABXY(1,10+I);Radix$(I),DVAL$(V,Radix(I));TABXY(49,10+I)
120   NEXT I
130   DISP "Enter a ";Radix$(R);" number";TAB(28);"(press SPACE to clear)"
140 W:GOTO W
```

```
150 Key:ON ERROR GOTO Bad                        ! Trap overrange
160   Key$=UPC$(KBD$)
170   Test=POS("0123456789ABCDEF",Key$)
180   IF Test AND Test<=Radix(R) THEN
190     V$=V$&Key$
200     V=DVAL(V$,Radix(R))
210   ELSE
220     IF Key$="-" THEN Toggle
230     BEEP 900,.02                             ! Not a digit key
240   END IF
250   IF Key$=" " THEN Erase
260   GOTO See
270 Bad:DISP ERRM$
280   BEEP
290   WAIT 1.5
300   GOTO Erase
310 Radix:R=1+R MOD 4
320   GOTO Erase
330 Toggle:IF V$[1;1]="-" THEN
340     V$[1,1]="0"
350   ELSE
360     V$="-"&V$
370   END IF
380   V=DVAL(V$,Radix(R))
390   GOTO See
400   END
```

The program starts by prompting for a decimal number to be entered. As the digits are typed, the number is displayed in each of the possible number bases. The softkey k5 or f5 lets you select the different number bases. Pressing the spacebar will clear the display.

Introduction to Lexical Order

The LEXICAL ORDER IS statement[1] lets you change the collating sequence (sorting order) of the character set. Changing the lexical order will affect the results of all string relational operators and operations, including the **MAT SORT**, **MAT SEARCH**, and **CASE** statements. In addition to redefining the collating sequence, the case conversion functions, **UPC$** and **LWC$**, are adjusted to reflect the current lexical order.

Predefined lexical orders include: ASCII, FRENCH, GERMAN, SPANISH, SWEDISH, and STANDARD. You can create lexical orders for special applications. The STANDARD lexical order is determined by an internal keyboard jumper, set at the factory to correspond to the keyboard supplied with the computer. The setting can be determined by examining the proper keyboard status register (STATUS 2,8;*Language*). Thus, the STANDARD lexical order on a computer equipped with a French keyboard will actually invoke the FRENCH lexical order.

Why Lexical Order?

A common task for computers is to arrange (sort) a group of items in alphabetical order. However, "alphabetical order" for a computer is normally based on the character sequence of the ASCII[2] character set. While the ASCII character sequence is adequate for many English Language applications, most foreign language alphabets include accented characters which are not part of the standard ASCII character set but must be included in the sequence to correctly sort the characters used in the language.

Since special character combinations often appear in some languages, these combinations and other special cases can be included in the lexical table to simplify working in other languages.

How It Works

The **LEXICAL ORDER IS** statement modifies the collating sequence by assigning a new value to each character. The new value, called a sequence number, is used in place of the character's ASCII value whenever characters are compared. Internally the characters retain their ASCII value, however the outcome of a comparison will be based on the sequence number assigned to the character instead of the character's ASCII value. In the process of comparing two strings, each of the strings is converted to a series of sequence numbers and the test is determined by the greater sequence numbers rather than the greater ASCII values.

[1] Available with LEX.

[2] ASCII stands for "American Standard Code for Information Interchange".

The ASCII Character Set

The ASCII set consists of 128 distinct characters including uppercase and lowercase alpha, numeric, punctuation, and control characters.

The table to the right shows the complete ASCII character set, as displayed on the CRT. Each character is preceded by its ASCII value. The character's value is actually the decimal representation of the binary value (bit pattern) used internally, by the computer, to represent the character.

The characters are arranged in ascending value, which is to say, in ascending lexical order. A character is "less than" another character if its ASCII value is smaller. From the table it can be seen that "A" is less than "B" since the value of the letter "A" (65) is less than the value of the letter "B" (66).

If you have experimented with string comparisons based on the ASCII collating sequence, you may have noticed a few shortcomings. Consider the following words.

RESTORE, RE-STORE, and RE_STORE

Sorting these items according to the ASCII collating sequence will arrange them in the following order.

RE-STORE < RESTORE < RE_STORE

This points out a limitation of string comparisons based on ASCII sequence. Since the hyphen's value (45) is less than any alpha-numeric character, and the underbar's value (95) is greater than all uppercase alpha characters, a word containing a hyphen will be less than the same word without the hyphen, and a word containing an underbar will be greater than the same word without the underbar. The **LEXICAL ORDER IS** statement lets you overcome these limitations by changing the sorting order of the character set.

Displaying Control Characters

Several special display features are available through the use of STATUS and CONTROL registers. Normally, ASCII characters 0 through 31 (control characters) are not displayed on the CRT. To enable the display of control characters, execute the following statement.

`CONTROL 1,4;1` or `DISPLAY FUNCTIONS ON`

Printing a line of text to the CRT will now show the trailing carriage-return and linefeed. Although this mode is useful for some applications, control characters are usually not displayed on the CRT.

`CONTROL 1,4;0` or `DISPLAY FUNCTIONS OFF`

Turns off the special display functions mode.

ASCII Character Set for CRT

Num	Chr.	Num.	Chr.	Num.	Chr.	Num.	Chr.
0	NU	32		64	@	96	`
1	SH	33	!	65	A	97	a
2	SX	34	"	66	B	98	b
3	EX	35	#	67	C	99	c
4	ET	36	$	68	D	100	d
5	EQ	37	%	69	E	101	e
6	AK	38	&	70	F	102	f
7	🔔	39	'	71	G	103	g
8	BS	40	(	72	H	104	h
9	HT	41	)	73	I	105	i
10	LF	42	*	74	J	106	j
11	VT	43	+	75	K	107	k
12	FF	44	,	76	L	108	l
13	CR	45	–	77	M	109	m
14	SO	46	.	78	N	110	n
15	SI	47	/	79	O	111	o
16	DL	48	0	80	P	112	p
17	D1	49	1	81	Q	113	q
18	D2	50	2	82	R	114	r
19	D3	51	3	83	S	115	s
20	D4	52	4	84	T	116	t
21	NK	53	5	85	U	117	u
22	SY	54	6	86	V	118	v
23	EB	55	7	87	W	119	w
24	CN	56	8	88	X	120	x
25	EM	57	9	89	Y	121	y
26	SB	58	:	90	Z	122	z
27	EC	59	;	91	[	123	{
28	FS	60	<	92	\	124	|
29	GS	61	=	93	]	125	}
30	RS	62	>	94	^	126	~
31	US	63	?	95	_	127	▓

Extended Character Set for CRT

Num.	Chr.	Num.	Chr.	Num.	Chr.	Num.	Chr.
128	CL	160		192	â	224	Á
129	IV	161	À	193	ê	225	Ã
130	BG	162	Â	194	ô	226	ã
131	IB	163	È	195	û	227	Đ
132	UL	164	Ê	196	á	228	đ
133	IV	165	Ë	197	é	229	Í
134	BG	166	Î	198	ó	230	Ì
135	IV	167	Ï	199	ú	231	Ó
136	WH	168	´	200	à	232	Ò
137	RD	169	`	201	è	233	Õ
138	YE	170	ˆ	202	ò	234	õ
139	GR	171	¨	203	ù	235	Š
140	CY	172	˜	204	ä	236	š
141	BU	173	Ù	205	ë	237	Ú
142	MG	174	Û	206	ö	238	Ÿ
143	BK	175	£	207	ü	239	ÿ
144	90	176	¯	208	Å	240	Þ
145	91	177	Ý	209	î	241	þ
146	92	178	ý	210	Ø	242	·
147	93	179	°	211	Æ	243	µ
148	94	180	Ç	212	å	244	¶
149	95	181	ç	213	í	245	IO
150	96	182	Ñ	214	ø	246	—
151	97	183	ñ	215	æ	247	¼
152	98	184	¡	216	Ä	248	½
153	99	185	¿	217	ì	249	ª
154	9A	186	¤	218	Ö	250	º
155	9B	187	£	219	Ü	251	«
156	9C	188	¥	220	É	252	■
157	9D	189	§	221	ï	253	»
158	9E	190	ƒ	222	ß	254	±
159	9F	191	¢	223	Ô	255	K

The Extended Character Set

Only 128 characters are defined in the ASCII character set. An additional 128 characters are available in the extended character set. The extended set includes CRT highlighting characters, special symbols, and Roman Extension characters (accented vowels and other characters used in many foreign languages).

Note

Some printers produce different extended characters than those displayed on the CRT. Check the printer manual for details on alternate character sets.

Highlight Characters

The first 32 characters in the extended character set are reserved for controlling various aspects of the CRT. The definition of these characters has been evolving with upgrades to both hardware and system software. Therefore, the action of these characters depends upon your model of computer and the level of BASIC (and Extensions) you have loaded.

With the BASIC system and Series 200/300 hardware, there is a possibility of having CRT highlights such as inverse video and blinking. The first eight characters (ASCII values 128 thru 135) are used to control these highlights, while the Model 226 is an example of a display without highlights. See the "Highlight Characters" tables in the appendix of the *BASIC Language Reference.*

The SYSTEM$ function is available and can be used to determine what CRT highlights are present. The expression

```
SYSTEM$("CRT ID")
```

returns a string containing information such as the CRT width and available highlights. The string returned by this expression is for Series 300 medium resolution monochrome monitors is:

```
6: 80H GB1
```

The 80 is the width of the CRT in characters and the H indicates that monochrome highlights are available. If there were a space instead of the H, then the CRT does not have highlights.

You can also determine if you have CRT highlights by sending a highlight control to the CRT and seeing if anything happens.

For example:

```
PRINT CHR$(132);"This is important.";CHR$(128)
```

On a display with highlights, this produces:

```
This is important.
```

On a display without highlights, the control characters are ignored and the line is displayed as normal text. Note that these control characters produce an action only in PRINT and DISP statements. When viewed in EDIT mode or on the system message line, these control characters appear as "$^{\mathrm{h}}{}_{\mathrm{p}}$" or as shown in the previous table "Extended Character Set for CRT."

Alternate CRT Characters

There is a keyboard control register for the CRT mapping of character codes, changing the contents of the register may cause different characters to be displayed.

Try the following.

```
PRINT CHR$(247)
CONTROL 1,11;1
PRINT CHR$(247)
CONTROL 1,11;0
```

The first print statement will produce the character expected from the character tables. The second print statement may show a character (double arrow) from an alternate character set. Note that the alternate character set is only available on some displays (such as the Model 236).

Finding "Missing" Characters

By now, you may have noticed that there are more possible CRT characters than keys on the keyboard. If your particular keyboard does not have a key for the character you need, locate the ANY CHAR key (every keyboard has this key).

When you press the ANY CHAR key, the message, "Enter 3 digits, 000 to 255" appears in the lower left corner of the CRT. Enter the three digits: 065 and the character whose value is 65 (the letter "A") will be placed on the screen. Any character can be input by this method. Pressing a non-digit key or entering a value outside the range will cancel the function.

Predefined Lexical Order

When the LEX Binary is first loaded or after a **SCRATCH A**, the computer executes a **LEXICAL ORDER IS STANDARD** statement. This will be the correct lexical order for the language on the keyboard. This can be checked by examining the keyboard status register (**STATUS** 2,8;*Language*) or by either of the following statements.

```
SYSTEM$("LEXICAL ORDER IS")
SYSTEM$("KEYBOARD LANGUAGE")
```

The table below shows the language indicated by the value returned by the **STATUS** statement. Thus, if the value returned indicates a French keyboard, the **STANDARD** lexical order will be the same as the **FRENCH** lexical order. The **STANDARD** lexical order for the Katakana keyboard is ASCII.

Value	Keyboard Language	Lexical Order
0	ASCII	ASCII
1	FRENCH	FRENCH
2	GERMAN	GERMAN
3	SWEDISH	SWEDISH
4	SPANISH[1]	SPANISH
5	KATAKANA	KATAKANA
6	CANADIAN ENGLISH	ASCII
7	UNITED KINGDOM	ASCII
8	CANADIAN FRENCH	FRENCH
9	SWISS FRENCH	FRENCH
10	ITALIAN	FRENCH
11	BELGIAN	GERMAN
12	DUTCH	GERMAN
13	SWISS GERMAN	GERMAN
14	LATIN[2]	SPANISH
15	DANISH	SWEDISH
16	FINNISH	SWEDISH
17	NORWEGIAN	SWEDISH
18	SWISS FRENCH*	FRENCH
19	SWISS GERMAN*	GERMAN

Either the **CHR$** function or [ANY CHAR] may be used to produce characters not readily available on the keyboard.

[1] This is the European Spanish keyboard.
[2] This is the Latin Spanish keyboard.

Lexical Tables

The following tables show the five predefined lexical orders available with the `LEXICAL ORDER IS` statement.

Notation

All of the lexical tables use the following notation.

```
    sequence number  →  113
character displayed  →  a
        ASCII value  →  (97)
```

Characters not available on the keyboard can be entered by pressing the [ANY CHAR] key and typing the value enclosed in parenthesis (with leading zeros, if needed). The character will be collated according to the sequence number shown above the character.

ASCII Lexical Order

The ASCII lexical order uses the character's ASCII value as the sequence number. There are no special cases (mode table entries) used in the ASCII lexical order.

Case Conversions

The following lists show the `UPC$` and `LWC$` transformations for the ASCII lexical order.

UPC$

```
abcdefghijklmnopqrstuvwxyzýçñâêôûáéóúàèòùäëöüîåíøæìïãðõšÿþ
ABCDEFGHIJKLMNOPQRSTUVWXYZÝÇÑÂÊÔÛÁÉÓÚÀÈÒÙÄËÖÜÎÅÍØÆÌÏÃÐÕŠŸÞ
```

LWC$

```
ABCDEFGHIJKLMNOPQRSTUVWXYZÀÂÈÊËÎÏÙÛÝÇÑÅØÆÄÖÜÉÔÁÃÐÍÌÓÒÕŠÚŸÞ
abcdefghijklmnopqrstuvwxyzàâèêëîïùûýçñåøæäöüéôáãðíìóòõšúÿþ
```

Note

There are slight variations in the operation of the `UPC$` and `LWC$` functions depending on the lexical order in effect. In other words, the lexical order determines which character will be returned by the `UPC$` and `LWC$` functions. The case conversion lists show which characters should be expected for each lexical order. To simplify the lists, characters not affected have been excluded.

LEXICAL ORDER IS ASCII

Seq.	Chr.	Num.	Seq.	Chr.	Num.	Seq.	Chr.	Num.	Seq.	Chr.	Num.	Seq.	Chr.	Num.
0	NU	(0)	52	4	(52)	104	h	(104)	156	9C	(156)	208	Å	(208)
1	SH	(1)	53	5	(53)	105	i	(105)	157	9D	(157)	209	î	(209)
2	SX	(2)	54	6	(54)	106	j	(106)	158	9E	(158)	210	Ø	(210)
3	EX	(3)	55	7	(55)	107	k	(107)	159	9F	(159)	211	Æ	(211)
4	ET	(4)	56	8	(56)	108	l	(108)	160		(160)	212	å	(212)
5	EQ	(5)	57	9	(57)	109	m	(109)	161	À	(161)	213	í	(213)
6	AK	(6)	58	:	(58)	110	n	(110)	162	Â	(162)	214	ø	(214)
7	🔔	(7)	59	;	(59)	111	o	(111)	163	È	(163)	215	æ	(215)
8	BS	(8)	60	<	(60)	112	p	(112)	164	Ê	(164)	216	Ä	(216)
9	HT	(9)	61	=	(61)	113	q	(113)	165	Ë	(165)	217	ì	(217)
10	LF	(10)	62	>	(62)	114	r	(114)	166	Î	(166)	218	Ö	(218)
11	VT	(11)	63	?	(63)	115	s	(115)	167	Ï	(167)	219	Ü	(219)
12	FF	(12)	64	@	(64)	116	t	(116)	168	´	(168)	220	É	(220)
13	CR	(13)	65	A	(65)	117	u	(117)	169	`	(169)	221	ï	(221)
14	SO	(14)	66	B	(66)	118	v	(118)	170	ˆ	(170)	222	ß	(222)
15	SI	(15)	67	C	(67)	119	w	(119)	171	¨	(171)	223	Ô	(223)
16	DL	(16)	68	D	(68)	120	x	(120)	172	˜	(172)	224	Á	(224)
17	D1	(17)	69	E	(69)	121	y	(121)	173	Ù	(173)	225	Ã	(225)
18	D2	(18)	70	F	(70)	122	z	(122)	174	Û	(174)	226	ã	(226)
19	D3	(19)	71	G	(71)	123	{	(123)	175	£	(175)	227	Đ	(227)
20	D4	(20)	72	H	(72)	124	|	(124)	176	¯	(176)	228	đ	(228)
21	NK	(21)	73	I	(73)	125	}	(125)	177	B1	(177)	229	Í	(229)
22	SY	(22)	74	J	(74)	126	~	(126)	178	B2	(178)	230	Ì	(230)
23	EB	(23)	75	K	(75)	127	▒	(127)	179	˚	(179)	231	Ó	(231)
24	CN	(24)	76	L	(76)	128	CL	(128)	180	Ç	(180)	232	Ò	(232)
25	EM	(25)	77	M	(77)	129	IV	(129)	181	ç	(181)	233	Õ	(233)
26	SB	(26)	78	N	(78)	130	BG	(130)	182	Ñ	(182)	234	õ	(234)
27	EC	(27)	79	O	(79)	131	IB	(131)	183	ñ	(183)	235	Š	(235)
28	FS	(28)	80	P	(80)	132	UL	(132)	184	¡	(184)	236	š	(236)
29	GS	(29)	81	Q	(81)	133	IV	(133)	185	¿	(185)	237	Ú	(237)
30	RS	(30)	82	R	(82)	134	BG	(134)	186	¤	(186)	238	Ÿ	(238)
31	US	(31)	83	S	(83)	135	IB	(135)	187	£	(187)	239	ÿ	(239)
32		(32)	84	T	(84)	136	WH	(136)	188	¥	(188)	240	Þ	(240)
33	!	(33)	85	U	(85)	137	RD	(137)	189	§	(189)	241	þ	(241)
34	"	(34)	86	V	(86)	138	YE	(138)	190	ƒ	(190)	242	F2	(242)
35	#	(35)	87	W	(87)	139	GR	(139)	191	¢	(191)	243	F3	(243)
36	$	(36)	88	X	(88)	140	CY	(140)	192	â	(192)	244	F4	(244)
37	%	(37)	89	Y	(89)	141	BU	(141)	193	ê	(193)	245	I0	(245)
38	&	(38)	90	Z	(90)	142	MG	(142)	194	ô	(194)	246	–	(246)
39	'	(39)	91	[	(91)	143	BK	(143)	195	û	(195)	247	¼	(247)
40	(	(40)	92	\	(92)	144	90	(144)	196	á	(196)	248	½	(248)
41	)	(41)	93	]	(93)	145	91	(145)	197	é	(197)	249	ª	(249)
42	*	(42)	94	^	(94)	146	92	(146)	198	ó	(198)	250	º	(250)
43	+	(43)	95	_	(95)	147	93	(147)	199	ú	(199)	251	«	(251)
44	,	(44)	96	`	(96)	148	94	(148)	200	à	(200)	252	■	(252)
45	–	(45)	97	a	(97)	149	95	(149)	201	è	(201)	253	»	(253)
46	.	(46)	98	b	(98)	150	96	(150)	202	ò	(202)	254	±	(254)
47	/	(47)	99	c	(99)	151	97	(151)	203	ù	(203)	255	🅺	(255)
48	0	(48)	100	d	(100)	152	98	(152)	204	ä	(204)			
49	1	(49)	101	e	(101)	153	99	(153)	205	ë	(205)			
50	2	(50)	102	f	(102)	154	9A	(154)	206	ö	(206)			
51	3	(51)	103	g	(103)	155	9B	(155)	207	ü	(207)			

FRENCH Lexical Order

The FRENCH lexical order table contains two special entries. The hyphen character (-) is assigned as a "don't care" character and a "2 for 1" character replacement is made for the "ß" character.

```
ß = ss
```

A string containing the hyphen will match the same string without the hyphen and a string containing only a hyphen will match the null string. For example:

```
LEXICAL ORDER IS FRENCH
IF "RE-STORE"="RESTORE" THEN PRINT "TRUE"
```

Prints: `TRUE`

Case Conversions

The following lists show the `UPC$` and `LWC$` transformations for the FRENCH lexical order.

UPC$

```
aâáàäåæãbcçdđeêéèëfghiîíìïjklmnñoôóòöøõpqrsštuûúùüvwxyÿzþý
AAAAAAÆÃBCCDĐEEEEEFGHIIIIIJKLMNÑOOOOOØÕPQRSŠTUUUUUVWXYYZÞY
```

LWC$

```
AÀÂAÆÄÁÃBCÇDĐEÈÊËÉFGHIÎÏÍÌJKLMNÑOØÖÔÓÒÕPQRSŠTUÙÛÜÚVWXYŸZÞÝ
aàâåæäáãbcçdđeèêëéfghiîïíìjklmnñoøöôóòõpqrsštuùûüúvwxyÿzþý
```

LEXICAL ORDER IS FRENCH

Seq.	Chr.	Num.	Seq.	Chr.	Num.	Seq.	Chr.	Num.	Seq.	Chr.	Num.	Seq.	Chr.	Num.
0	–	(45)	51	4	(52)	81	P	(80)	113	l	(108)	153	½	(248)
0	NU	(0)	52	5	(53)	82	Q	(81)	114	m	(109)	154	ª	(249)
1	SH	(1)	53	6	(54)	83	R	(82)	115	n	(110)	155	º	(250)
2	SX	(2)	54	7	(55)	84	S	(83)	116	ñ	(183)	156	«	(251)
3	EX	(3)	55	8	(56)	85	Š	(235)	117	o	(111)	157	■	(252)
4	ET	(4)	56	9	(57)	86	T	(84)	117	ô	(194)	158	»	(253)
5	EQ	(5)	57	:	(58)	87	U	(85)	117	ó	(198)	159	±	(254)
6	AK	(6)	58	;	(59)	87	Ù	(173)	117	ò	(202)	160	▒	(127)
7	🔔	(7)	59	<	(60)	87	Û	(174)	117	ö	(206)	161		(160)
8	BS	(8)	60	=	(61)	87	Ü	(219)	117	ø	(214)	162	B1	(177)
9	HT	(9)	61	>	(62)	87	Ú	(237)	117	õ	(234)	163	B2	(178)
10	LF	(10)	62	?	(63)	88	V	(86)	118	p	(112)	164	F2	(242)
11	VT	(11)	63	@	(64)	89	W	(87)	119	q	(113)	165	F3	(243)
12	FF	(12)	64	A	(65)	90	X	(88)	120	r	(114)	166	F4	(244)
13	CR	(13)	64	À	(161)	91	Y	(89)	121	s	(115)	167	ID	(245)
14	SO	(14)	64	Â	(162)	91	Ÿ	(238)	121	ß	(222)	168	CL	(128)
15	SI	(15)	64	Å	(208)	92	Z	(90)	122	š	(236)	169	IV	(129)
16	DL	(16)	64	Æ	(211)	93	Þ	(240)	123	t	(116)	170	BG	(130)
17	D1	(17)	64	Ä	(216)	94	[	(91)	124	u	(117)	171	IB	(131)
18	D2	(18)	64	Á	(224)	95	\	(92)	124	û	(195)	172	UL	(132)
19	D3	(19)	64	Ã	(225)	96	]	(93)	124	ú	(199)	173	IU	(133)
20	D4	(20)	65	B	(66)	97	^	(94)	124	ù	(203)	174	BG	(134)
21	NK	(21)	66	C	(67)	98	_	(95)	124	ü	(207)	175	IB	(135)
22	SY	(22)	66	Ç	(180)	99	`	(96)	125	v	(118)	176	WH	(136)
23	EB	(23)	67	D	(68)	100	a	(97)	126	w	(119)	177	RD	(137)
24	CN	(24)	68	Đ	(227)	100	â	(192)	127	x	(120)	178	YE	(138)
25	EM	(25)	69	E	(69)	100	á	(196)	128	y	(121)	179	GR	(139)
26	SB	(26)	69	È	(163)	100	à	(200)	128	ÿ	(239)	180	CY	(140)
27	EC	(27)	69	Ê	(164)	100	ä	(204)	129	z	(122)	181	BU	(141)
28	FS	(28)	69	Ë	(165)	100	å	(212)	130	þ	(241)	182	MG	(142)
29	GS	(29)	69	É	(220)	100	æ	(215)	131	{	(123)	183	BK	(143)
30	RS	(30)	70	F	(70)	100	ã	(226)	132	|	(124)	184	90	(144)
31	US	(31)	71	G	(71)	101	b	(98)	133	}	(125)	185	91	(145)
32		(32)	72	H	(72)	102	c	(99)	134	~	(126)	186	92	(146)
33	!	(33)	73	I	(73)	103	ç	(181)	135	´	(168)	187	93	(147)
34	"	(34)	73	Î	(166)	104	d	(100)	136	`	(169)	188	94	(148)
35	#	(35)	73	Ï	(167)	105	đ	(228)	137	ˆ	(170)	189	95	(149)
36	$	(36)	73	Í	(229)	106	e	(101)	138	¨	(171)	190	96	(150)
37	%	(37)	73	Ì	(230)	106	ê	(193)	139	˜	(172)	191	97	(151)
38	&	(38)	74	J	(74)	106	é	(197)	140	£	(175)	192	98	(152)
39	'	(39)	75	K	(75)	106	è	(201)	141	¯	(176)	193	99	(153)
40	(	(40)	76	L	(76)	106	ë	(205)	142	°	(179)	194	9A	(154)
41	)	(41)	77	M	(77)	107	f	(102)	143	¡	(184)	195	9B	(155)
42	*	(42)	78	N	(78)	108	g	(103)	144	¿	(185)	196	9C	(156)
43	+	(43)	79	Ñ	(182)	109	h	(104)	145	¤	(186)	197	9D	(157)
44	,	(44)	80	O	(79)	110	i	(105)	146	£	(187)	198	9E	(158)
45	.	(46)	80	Ø	(210)	110	î	(209)	147	¥	(188)	199	9F	(159)
46	/	(47)	80	Ö	(218)	110	í	(213)	148	§	(189)	200	K	(255)
47	0	(48)	80	Ô	(223)	110	ì	(217)	149	ƒ	(190)			
48	1	(49)	80	Ó	(231)	110	ï	(221)	150	¢	(191)			
49	2	(50)	80	Ò	(232)	111	j	(106)	151	–	(246)			
50	3	(51)	80	Õ	(233)	112	k	(107)	152	¼	(247)			

GERMAN Lexical Order

The GERMAN lexical order table contains seven "2 for 1" character replacements. When the following individual characters are found in a string, two sequence numbers are generated, as if two characters were found in the string.

```
ä = ae
ö = oe
ü = ue
Ä = AE   or   Ae
Ö = OE   or   Oe
Ü = UE   or   Ue
ß = ss
```

Case Conversions

The following lists show the `UPC$` and `LWC$` transformations for the GERMAN lexical order.

UPC$

```
aäæåáàâãbcçdđeéèêëfghiíìîïjklmnñoöóòôõøpqrsštuüúùûvwxyÿzþý
AÄÆÅÁÀÂÃBCCDĐEÉÈÊËFGHIÍÌÎÏJKLMNÑOÖÓÒÔÕØPQRSŠTUÜÚÙÛVWXYŸZÞÝ
```

LWC$

```
AÄÆÅÁÀÂÃBCÇDĐEÉÈÊËFGHIÍÌÎÏJKLMNÑOÖÓÒÔÕØPQRSŠTUÜÚÙÛVWXYŸZÞÝ
aäæåáàâãbcçdđeéèêëfghiíìîïjklmnñoöóoôõøpqrsštuüúùûvwxyÿzþý
```

LEXICAL ORDER IS GERMAN

Seq.	Chr.	Num.	Seq.	Chr.	Num.	Seq.	Chr.	Num.	Seq.	Chr.	Num.	Seq.	Chr.	Num.
0	NU	(0)	52	4	(52)	102	P	(80)	152	l	(108)	201	½	(248)
1	SH	(1)	53	5	(53)	103	Q	(81)	153	m	(109)	202	ª	(249)
2	SX	(2)	54	6	(54)	104	R	(82)	154	n	(110)	203	º	(250)
3	EX	(3)	55	7	(55)	105	S	(83)	155	ñ	(183)	204	«	(251)
4	ET	(4)	56	8	(56)	106	Š	(235)	156	o	(111)	205	■	(252)
5	EQ	(5)	57	9	(57)	107	T	(84)	156	ö	(206)	206	»	(253)
6	AK	(6)	58	:	(58)	108	U	(85)	157	ó	(198)	207	±	(254)
7	🔔	(7)	59	;	(59)	108	Ü	(219)	158	ò	(202)	208	▒	(127)
8	BS	(8)	60	<	(60)	109	Ú	(237)	159	ô	(194)	209		(160)
9	HT	(9)	61	=	(61)	110	Ù	(173)	160	õ	(234)	210	B1	(177)
10	LF	(10)	62	>	(62)	111	Û	(174)	161	ø	(214)	211	B2	(178)
11	VT	(11)	63	?	(63)	112	V	(86)	162	p	(112)	212	F2	(242)
12	FF	(12)	64	@	(64)	113	W	(87)	163	q	(113)	213	F3	(243)
13	CR	(13)	65	A	(65)	114	X	(88)	164	r	(114)	214	F4	(244)
14	SO	(14)	65	Ä	(216)	115	Y	(89)	165	s	(115)	215	IO	(245)
15	SI	(15)	66	Æ	(211)	116	Ÿ	(238)	165	ß	(222)	216	CL	(128)
16	DL	(16)	67	Å	(208)	117	Z	(90)	166	š	(236)	217	IV	(129)
17	D1	(17)	68	Á	(224)	118	Þ	(240)	167	t	(116)	218	BG	(130)
18	D2	(18)	69	À	(161)	119	[	(91)	168	u	(117)	219	IB	(131)
19	D3	(19)	70	Â	(162)	120	\	(92)	168	ü	(207)	220	UL	(132)
20	D4	(20)	71	Ã	(225)	121	]	(93)	169	ú	(199)	221	IV	(133)
21	NK	(21)	72	B	(66)	122	^	(94)	170	ù	(203)	222	BG	(134)
22	SY	(22)	73	C	(67)	123	_	(95)	171	û	(195)	223	IB	(135)
23	EB	(23)	74	Ç	(180)	124	`	(96)	172	v	(118)	224	WH	(136)
24	CN	(24)	75	D	(68)	125	a	(97)	173	w	(119)	225	RD	(137)
25	EM	(25)	76	Đ	(227)	125	ä	(204)	174	x	(120)	226	YE	(138)
26	SB	(26)	77	E	(69)	126	æ	(215)	175	y	(121)	227	GR	(139)
27	EC	(27)	78	É	(220)	127	å	(212)	176	ÿ	(239)	228	CY	(140)
28	FS	(28)	79	È	(163)	128	á	(196)	177	z	(122)	229	BU	(141)
29	GS	(29)	80	Ê	(164)	129	à	(200)	178	þ	(241)	230	MG	(142)
30	RS	(30)	81	Ë	(165)	130	â	(192)	179	{	(123)	231	BK	(143)
31	US	(31)	82	F	(70)	131	ã	(226)	180	|	(124)	232	90	(144)
32		(32)	83	G	(71)	132	b	(98)	181	}	(125)	233	91	(145)
33	!	(33)	84	H	(72)	133	c	(99)	182	~	(126)	234	92	(146)
34	"	(34)	85	I	(73)	134	ç	(181)	183	´	(168)	235	93	(147)
35	#	(35)	86	Í	(229)	135	d	(100)	184	`	(169)	236	94	(148)
36	$	(36)	87	Ì	(230)	136	đ	(228)	185	ˆ	(170)	237	95	(149)
37	%	(37)	88	Î	(166)	137	e	(101)	186	¨	(171)	238	96	(150)
38	&	(38)	89	Ï	(167)	138	é	(197)	187	˜	(172)	239	97	(151)
39	'	(39)	90	J	(74)	139	è	(201)	188	₤	(175)	240	98	(152)
40	(	(40)	91	K	(75)	140	ê	(193)	189	¯	(176)	241	99	(153)
41	)	(41)	92	L	(76)	141	ë	(205)	190	˙	(179)	242	9A	(154)
42	*	(42)	93	M	(77)	142	f	(102)	191	¡	(184)	243	9B	(155)
43	+	(43)	94	N	(78)	143	g	(103)	192	¿	(185)	244	9C	(156)
44	,	(44)	95	Ñ	(182)	144	h	(104)	193	¤	(186)	245	9D	(157)
45	-	(45)	96	O	(79)	145	i	(105)	194	£	(187)	246	9E	(158)
46	.	(46)	96	Ö	(218)	146	í	(213)	195	¥	(188)	247	9F	(159)
47	/	(47)	97	Ó	(231)	147	ì	(217)	196	§	(189)	248	🅺	(255)
48	0	(48)	98	Ò	(232)	148	î	(209)	197	ƒ	(190)			
49	1	(49)	99	Ô	(223)	149	ï	(221)	198	¢	(191)			
50	2	(50)	100	Õ	(233)	150	j	(106)	199	–	(246)			
51	3	(51)	101	Ø	(210)	151	k	(107)	200	¼	(247)			

SPANISH Lexical Order

The SPANISH lexical order table contains five special entries. Four of these entries are "1 for 2" character replacements. When the following character pairs are found in a string, a single sequence number is used to represent the pair.

CH	=	68	cH	=	106
Ch	=	68	ch	=	106
LL	=	79	lL	=	117
Ll	=	79	ll	=	117

The remaining special case is a "2 for 1" entry for the "ß" character.

```
ß = ss
```

Case Conversions

The following lists show the `UPC$` and `LWC$` transformations for the SPANISH lexical order.

UPC$

```
aâáàäåæãbcçdđeêéèëfghiîíìïjklmnñoôóòöøõpqrsštuûúùüvwxyÿzþý
AAAAÄÅÆÃBCCDĐEEEEEFGHIIIIIJKLMNÑOOOOöØÕPQRSšTUUUUüVWXYÿZÞY
```

LWC$

```
AÀÂÅÆÄÁÃBCÇDĐEÈÊËÉFGHIÎÏÍÌJKLMNÑOØÖÔÓÒÕPQRSŠTUÙÛÜÚVWXYŸZÞÝ
aàâåæäáãbcçdđeèêëéfghiîïíìjklmnñoøöôóoõpqrsštuùûüúvwxyÿzþý
```

LEXICAL ORDER IS SPANISH

Seq.	Chr.	Num.	Seq.	Chr.	Num.	Seq.	Chr.	Num.	Seq.	Chr.	Num.	Seq.	Chr.	Num.
0	NU	(0)	52	4	(52)	84	P	(80)	116	l	(108)	157	½	(248)
1	SH	(1)	53	5	(53)	85	Q	(81)	118	m	(109)	158	ª	(249)
2	SX	(2)	54	6	(54)	86	R	(82)	119	n	(110)	159	º	(250)
3	EX	(3)	55	7	(55)	87	S	(83)	120	ñ	(183)	160	«	(251)
4	ET	(4)	56	8	(56)	88	Š	(235)	121	o	(111)	161	■	(252)
5	EQ	(5)	57	9	(57)	89	T	(84)	121	ô	(194)	162	»	(253)
6	AK	(6)	58	:	(58)	90	U	(85)	121	ó	(198)	163	±	(254)
7	🔔	(7)	59	;	(59)	90	Ù	(173)	121	ò	(202)	164	▒	(127)
8	BS	(8)	60	<	(60)	90	Û	(174)	121	ö	(206)	165		(160)
9	HT	(9)	61	=	(61)	90	Ü	(219)	121	ø	(214)	166	B1	(177)
10	LF	(10)	62	>	(62)	90	Ú	(237)	121	õ	(234)	167	B2	(178)
11	VT	(11)	63	?	(63)	91	V	(86)	122	p	(112)	168	F2	(242)
12	FF	(12)	64	@	(64)	92	W	(87)	123	q	(113)	169	F3	(243)
13	CR	(13)	65	A	(65)	93	X	(88)	124	r	(114)	170	F4	(244)
14	SO	(14)	65	À	(161)	94	Y	(89)	125	s	(115)	171	ID	(245)
15	SI	(15)	65	Â	(162)	94	Ÿ	(238)	125	ß	(222)	172	CL	(128)
16	DL	(16)	65	Å	(208)	95	Z	(90)	126	š	(236)	173	IV	(129)
17	D1	(17)	65	Æ	(211)	96	Þ	(240)	127	t	(116)	174	BG	(130)
18	D2	(18)	65	Ä	(216)	97	[	(91)	128	u	(117)	175	IB	(131)
19	D3	(19)	65	Á	(224)	98	\	(92)	128	û	(195)	176	UL	(132)
20	D4	(20)	65	Ã	(225)	99	]	(93)	128	ú	(199)	177	IU	(133)
21	NK	(21)	66	B	(66)	100	^	(94)	128	ù	(203)	178	BL	(134)
22	SY	(22)	67	C	(67)	101	_	(95)	128	ü	(207)	179	IB	(135)
23	EB	(23)	67	Ç	(180)	102	`	(96)	129	v	(118)	180	WH	(136)
24	CN	(24)	69	D	(68)	103	a	(97)	130	w	(119)	181	RD	(137)
25	EM	(25)	70	Đ	(227)	103	â	(192)	131	x	(120)	182	YE	(138)
26	SB	(26)	71	E	(69)	103	á	(196)	132	y	(121)	183	GR	(139)
27	EC	(27)	71	È	(163)	103	à	(200)	132	ÿ	(239)	184	CY	(140)
28	FS	(28)	71	Ê	(164)	103	ä	(204)	133	z	(122)	185	BU	(141)
29	GS	(29)	71	Ë	(165)	103	å	(212)	134	þ	(241)	186	MG	(142)
30	RS	(30)	71	É	(220)	103	æ	(215)	135	{	(123)	187	BK	(143)
31	US	(31)	72	F	(70)	103	ã	(226)	136	|	(124)	188	90	(144)
32		(32)	73	G	(71)	104	b	(98)	137	}	(125)	189	91	(145)
33	!	(33)	74	H	(72)	105	c	(99)	138	~	(126)	190	92	(146)
34	"	(34)	75	I	(73)	105	ç	(181)	139	´	(168)	191	93	(147)
35	#	(35)	75	Î	(166)	107	d	(100)	140	`	(169)	192	94	(148)
36	$	(36)	75	Ï	(167)	108	đ	(228)	141	ˆ	(170)	193	95	(149)
37	%	(37)	75	Í	(229)	109	e	(101)	142	¨	(171)	194	96	(150)
38	&	(38)	75	Ì	(230)	109	ê	(193)	143	˜	(172)	195	97	(151)
39	'	(39)	76	J	(74)	109	é	(197)	144	£	(175)	196	98	(152)
40	(	(40)	77	K	(75)	109	è	(201)	145	¯	(176)	197	99	(153)
41	)	(41)	78	L	(76)	109	ë	(205)	146	°	(179)	198	9A	(154)
42	*	(42)	80	M	(77)	110	f	(102)	147	¡	(184)	199	9B	(155)
43	+	(43)	81	N	(78)	111	g	(103)	148	¿	(185)	200	9C	(156)
44	,	(44)	82	Ñ	(182)	112	h	(104)	149	¤	(186)	201	9D	(157)
45	-	(45)	83	O	(79)	113	i	(105)	150	£	(187)	202	9E	(158)
46	.	(46)	83	Ø	(210)	113	î	(209)	151	¥	(188)	203	9F	(159)
47	/	(47)	83	Ö	(218)	113	í	(213)	152	§	(189)	204	K	(255)
48	0	(48)	83	Ô	(223)	113	ì	(217)	153	ƒ	(190)			
49	1	(49)	83	Ó	(231)	113	ï	(221)	154	¢	(191)			
50	2	(50)	83	Ò	(232)	114	j	(106)	155	–	(246)			
51	3	(51)	83	Õ	(233)	115	k	(107)	156	¼	(247)			

SWEDISH Lexical Order

The SWEDISH lexical order table includes one "2 for 1" character replacement entry. When the "ß" character is found in a string, two sequence numbers are generated, as if two characters were found in the string.

"ß = ss

Case Conversions

The following lists show the `UPC$` and `LWC$` transformations for the SWEDISH lexical order.

UPC$

```
abcdeéfghijklmnopqrstuvwxyzæåáàâäãçđèêëíìîïñóòôöõøšúùûüÿþý
ABCDEÉFGHIJKLMNOPQRSTUVWXYZÆÅÁÀÂÄÃCĐÈÊËÍÌÎÏÑÓÒÔÖÕØŠÚÙÛÜŸÞÝ
```

LWC$

```
ABCDEFGHIJKLMNOPQRSTUVWXYZÆÅÁÀÂÄÃÇĐÉÈÊËÍÌÎÏÑÓÒÔÖÕØŠÚÙÛÜŸÞÝ
abcdefghijklmnopqrstuvwxyzæåáàâäãçđéèêëíìîïñóòôöõøšúùûüÿþý
```

LEXICAL ORDER IS SWEDISH

Seq.	Chr.	Num.	Seq.	Chr.	Num.	Seq.	Chr.	Num.	Seq.	Chr.	Num.	Seq.	Chr.	Num.
0	NU	(0)	52	4	(52)	104	Í	(229)	154	æ	(215)	206	½	(248)
1	SH	(1)	53	5	(53)	105	Ì	(230)	155	å	(212)	207	ª	(249)
2	SX	(2)	54	6	(54)	106	Î	(166)	156	á	(196)	208	º	(250)
3	EX	(3)	55	7	(55)	107	Ï	(167)	157	à	(200)	209	«	(251)
4	ET	(4)	56	8	(56)	108	Ñ	(182)	158	â	(192)	210	■	(252)
5	EQ	(5)	57	9	(57)	109	Ó	(231)	159	ä	(204)	211	»	(253)
6	AK	(6)	58	:	(58)	110	Ò	(232)	160	ã	(226)	212	±	(254)
7	🔔	(7)	59	;	(59)	111	Ô	(223)	161	ç	(181)	213	▒	(127)
8	BS	(8)	60	<	(60)	112	Ö	(218)	162	đ	(228)	214		(160)
9	HT	(9)	61	=	(61)	113	Õ	(233)	163	è	(201)	215	B1	(177)
10	LF	(10)	62	>	(62)	114	Ø	(210)	164	ê	(193)	216	B2	(178)
11	VT	(11)	63	?	(63)	115	Š	(235)	165	ë	(205)	217	F2	(242)
12	FF	(12)	64	@	(64)	116	Ú	(237)	166	í	(213)	218	F3	(243)
13	CR	(13)	65	A	(65)	117	Ù	(173)	167	ì	(217)	219	F4	(244)
14	SO	(14)	66	B	(66)	118	Û	(174)	168	î	(209)	220	IO	(245)
15	SI	(15)	67	C	(67)	119	Ü	(219)	169	ï	(221)	221	CL	(128)
16	DL	(16)	68	D	(68)	120	Ÿ	(238)	170	ñ	(183)	222	IV	(129)
17	D1	(17)	69	E	(69)	121	Þ	(240)	171	ó	(198)	223	BG	(130)
18	D2	(18)	70	F	(70)	122	[	(91)	172	ò	(202)	224	IB	(131)
19	D3	(19)	71	G	(71)	123	\	(92)	173	ô	(194)	225	UL	(132)
20	D4	(20)	72	H	(72)	124	]	(93)	174	ö	(206)	226	IV	(133)
21	NK	(21)	73	I	(73)	125	^	(94)	175	õ	(234)	227	BG	(134)
22	SY	(22)	74	J	(74)	126	_	(95)	176	ø	(214)	228	IB	(135)
23	EB	(23)	75	K	(75)	127	`	(96)	177	š	(236)	229	WH	(136)
24	CN	(24)	76	L	(76)	128	a	(97)	178	ú	(199)	230	RD	(137)
25	EM	(25)	77	M	(77)	129	b	(98)	179	ù	(203)	231	YE	(138)
26	SB	(26)	78	N	(78)	130	c	(99)	180	û	(195)	232	GR	(139)
27	EC	(27)	79	O	(79)	131	d	(100)	181	ü	(207)	233	CY	(140)
28	FS	(28)	80	P	(80)	132	e	(101)	182	ÿ	(239)	234	BU	(141)
29	GS	(29)	81	Q	(81)	132	é	(197)	183	þ	(241)	235	MG	(142)
30	RS	(30)	82	R	(82)	133	f	(102)	184	{	(123)	236	BK	(143)
31	US	(31)	83	S	(83)	134	g	(103)	185	|	(124)	237	90	(144)
32		(32)	84	T	(84)	135	h	(104)	186	}	(125)	238	91	(145)
33	!	(33)	85	U	(85)	136	i	(105)	187	~	(126)	239	92	(146)
34	"	(34)	86	V	(86)	137	j	(106)	188	´	(168)	240	93	(147)
35	#	(35)	87	W	(87)	138	k	(107)	189	`	(169)	241	94	(148)
36	$	(36)	88	X	(88)	139	l	(108)	190	ˆ	(170)	242	95	(149)
37	%	(37)	89	Y	(89)	140	m	(109)	191	¨	(171)	243	96	(150)
38	&	(38)	90	Z	(90)	141	n	(110)	192	˜	(172)	244	97	(151)
39	'	(39)	91	Æ	(211)	142	o	(111)	193	₤	(175)	245	98	(152)
40	(	(40)	92	Å	(208)	143	p	(112)	194	¯	(176)	246	99	(153)
41	)	(41)	93	Á	(224)	144	q	(113)	195	°	(179)	247	9A	(154)
42	*	(42)	94	À	(161)	145	r	(114)	196	¡	(184)	248	9B	(155)
43	+	(43)	95	Â	(162)	146	s	(115)	197	¿	(185)	249	9C	(156)
44	,	(44)	96	Ä	(216)	146	ß	(222)	198	¤	(186)	250	9D	(157)
45	-	(45)	97	Ã	(225)	147	t	(116)	199	£	(187)	251	9E	(158)
46	.	(46)	98	Ç	(180)	148	u	(117)	200	¥	(188)	252	9F	(159)
47	/	(47)	99	Đ	(227)	149	v	(118)	201	§	(189)	253	K	(255)
48	0	(48)	100	É	(220)	150	w	(119)	202	ƒ	(190)			
49	1	(49)	101	È	(163)	151	x	(120)	203	¢	(191)			
50	2	(50)	102	Ê	(164)	152	y	(121)	204	–	(246)			
51	3	(51)	103	Ë	(165)	153	z	(122)	205	¼	(247)			

User-defined LEXICAL ORDER

The following program will generate the worksheet on the next page. The worksheet is handy when creating a user-defined lexical order.

```
10    DIM Lb$[1],F1$[23],F2$[23],F3$[14],F4$[20],Falt$[96],F1p$[22],F2p$[22]
20    INTEGER I
30    OUTPUT PRT;"LEXICAL ORDER TABLE WORKSHEET     |seq-num:mode-type.mode
-entry|"
40    OUTPUT PRT
50    Lb$="#"
60    F1p$=" ,DD,X,""|   :  .   ||"""
70    F1$=" ,DDD,X,""|   :  .   ||"""
80    F2p$=",X,A,X,""|   :  .   ||"""
90    F2$=",XX,A,X,""|   :  .   ||"""
100   F4$=" ,DD,X,""|   :   ||"""
110   F3$=",""Mode Length"""
120   Falt$=F1p$&F2$&F1$&F2$
130   FOR I=0 TO 63
140     SELECT I
150     CASE 0
160       OUTPUT PRT USING Lb$&Falt$&F3$;I,CHR$(I+64),I+128,CHR$(I+192)
170     CASE <32
180       OUTPUT PRT USING Lb$&Falt$&F4$;I,CHR$(I+64),I+128,CHR$(I+192),I-1
190     CASE ELSE
200       OUTPUT PRT USING Lb$&F2p$&RPT$(F2$,3)&F4$;CHR$(I),CHR$(I+64),
CHR$(I+128),CHR$(I+192),I-1
210     END SELECT
220     OUTPUT PRT
230   NEXT I
240   END
```

```
   LEXICAL ORDER TABLE WORKSHEET     |seq-num:mode-type.mode-entry|

 0 |  :  .  || @ |  :  .  ||128 |  :  .  || â |  :  .  ||Mode Length
 1 |  :  .  || A |  :  .  ||129 |  :  .  || ê |  :  .  || 0 |  :  ||
 2 |  :  .  || B |  :  .  ||130 |  :  .  || ô |  :  .  || 1 |  :  ||
 3 |  :  .  || C |  :  .  ||131 |  :  .  || û |  :  .  || 2 |  :  ||
 4 |  :  .  || D |  :  .  ||132 |  :  .  || á |  :  .  || 3 |  :  ||
 5 |  :  .  || E |  :  .  ||133 |  :  .  || é |  :  .  || 4 |  :  ||
 6 |  :  .  || F |  :  .  ||134 |  :  .  || ó |  :  .  || 5 |  :  ||
 7 |  :  .  || G |  :  .  ||135 |  :  .  || ú |  :  .  || 6 |  :  ||
 8 |  :  .  || H |  :  .  ||136 |  :  .  || à |  :  .  || 7 |  :  ||
 9 |  :  .  || I |  :  .  ||137 |  :  .  || è |  :  .  || 8 |  :  ||
10 |  :  .  || J |  :  .  ||138 |  :  .  || ò |  :  .  || 9 |  :  ||
11 |  :  .  || K |  :  .  ||139 |  :  .  || ù |  :  .  ||10 |  :  ||
12 |  :  .  || L |  :  .  ||140 |  :  .  || ä |  :  .  ||11 |  :  ||
13 |  :  .  || M |  :  .  ||141 |  :  .  || ë |  :  .  ||12 |  :  ||
14 |  :  .  || N |  :  .  ||142 |  :  .  || ö |  :  .  ||13 |  :  ||
15 |  :  .  || O |  :  .  ||143 |  :  .  || ü |  :  .  ||14 |  :  ||
16 |  :  .  || P |  :  .  ||144 |  :  .  || Å |  :  .  ||15 |  :  ||
17 |  :  .  || Q |  :  .  ||145 |  :  .  || î |  :  .  ||16 |  :  ||
18 |  :  .  || R |  :  .  ||146 |  :  .  || Ø |  :  .  ||17 |  :  ||
19 |  :  .  || S |  :  .  ||147 |  :  .  || Æ |  :  .  ||18 |  :  ||
20 |  :  .  || T |  :  .  ||148 |  :  .  || å |  :  .  ||19 |  :  ||
21 |  :  .  || U |  :  .  ||149 |  :  .  || í |  :  .  ||20 |  :  ||
22 |  :  .  || V |  :  .  ||150 |  :  .  || ø |  :  .  ||21 |  :  ||
23 |  :  .  || W |  :  .  ||151 |  :  .  || æ |  :  .  ||22 |  :  ||
24 |  :  .  || X |  :  .  ||152 |  :  .  || Ä |  :  .  ||23 |  :  ||
25 |  :  .  || Y |  :  .  ||153 |  :  .  || ì |  :  .  ||24 |  :  ||
26 |  :  .  || Z |  :  .  ||154 |  :  .  || Ö |  :  .  ||25 |  :  ||
27 |  :  .  || [ |  :  .  ||155 |  :  .  || Ü |  :  .  ||26 |  :  ||
28 |  :  .  || \ |  :  .  ||156 |  :  .  || É |  :  .  ||27 |  :  ||
29 |  :  .  || ] |  :  .  ||157 |  :  .  || ï |  :  .  ||28 |  :  ||
30 |  :  .  || ^ |  :  .  ||158 |  :  .  || ß |  :  .  ||29 |  :  ||
31 |  :  .  || _ |  :  .  ||159 |  :  .  || Ô |  :  .  ||30 |  :  ||
   |  :  .  || ` |  :  .  ||    |  :  .  || Á |  :  .  ||31 |  :  ||
 ! |  :  .  || a |  :  .  || À  |  :  .  || Ã |  :  .  ||32 |  :  ||
 " |  :  .  || b |  :  .  || Â  |  :  .  || ã |  :  .  ||33 |  :  ||
 # |  :  .  || c |  :  .  || È  |  :  .  || Đ |  :  .  ||34 |  :  ||
 $ |  :  .  || d |  :  .  || Ê  |  :  .  || đ |  :  .  ||35 |  :  ||
 % |  :  .  || e |  :  .  || Ë  |  :  .  || Í |  :  .  ||36 |  :  ||
 & |  :  .  || f |  :  .  || Î  |  :  .  || Ì |  :  .  ||37 |  :  ||
 ' |  :  .  || g |  :  .  || Ï  |  :  .  || Ó |  :  .  ||38 |  :  ||
 ( |  :  .  || h |  :  .  || ´  |  :  .  || Ò |  :  .  ||39 |  :  ||
 ) |  :  .  || i |  :  .  || `  |  :  .  || Õ |  :  .  ||40 |  :  ||
 * |  :  .  || j |  :  .  || ˆ  |  :  .  || õ |  :  .  ||41 |  :  ||
 + |  :  .  || k |  :  .  || ¨  |  :  .  || Š |  :  .  ||42 |  :  ||
 , |  :  .  || l |  :  .  || ˜  |  :  .  || š |  :  .  ||43 |  :  ||
 - |  :  .  || m |  :  .  || Ù  |  :  .  || Ú |  :  .  ||44 |  :  ||
 . |  :  .  || n |  :  .  || Û  |  :  .  || Ÿ |  :  .  ||45 |  :  ||
 / |  :  .  || o |  :  .  || £  |  :  .  || ÿ |  :  .  ||46 |  :  ||
 0 |  :  .  || p |  :  .  || ¯  |  :  .  || Þ |  :  .  ||47 |  :  ||
 1 |  :  .  || q |  :  .  ||    |  :  .  || þ |  :  .  ||48 |  :  ||
 2 |  :  .  || r |  :  .  ||    |  :  .  ||   |  :  .  ||49 |  :  ||
 3 |  :  .  || s |  :  .  || •  |  :  .  ||   |  :  .  ||50 |  :  ||
 4 |  :  .  || t |  :  .  || Ç  |  :  .  ||   |  :  .  ||51 |  :  ||
 5 |  :  .  || u |  :  .  || ç  |  :  .  ||   |  :  .  ||52 |  :  ||
 6 |  :  .  || v |  :  .  || Ñ  |  :  .  || — |  :  .  ||53 |  :  ||
 7 |  :  .  || w |  :  .  || ñ  |  :  .  || ¼ |  :  .  ||54 |  :  ||
 8 |  :  .  || x |  :  .  || ¡  |  :  .  || ½ |  :  .  ||55 |  :  ||
 9 |  :  .  || y |  :  .  || ¿  |  :  .  || ª |  :  .  ||56 |  :  ||
 : |  :  .  || z |  :  .  || ¤  |  :  .  || º |  :  .  ||57 |  :  ||
 ; |  :  .  || { |  :  .  || £  |  :  .  || « |  :  .  ||58 |  :  ||
 < |  :  .  || | |  :  .  || ¥  |  :  .  || ■ |  :  .  ||59 |  :  ||
 = |  :  .  || } |  :  .  || §  |  :  .  || » |  :  .  ||60 |  :  ||
 > |  :  .  || ~ |  :  .  || ƒ  |  :  .  || ± |  :  .  ||61 |  :  ||
 ? |  :  .  || ■ |  :  .  || ¢  |  :  .  || ■ |  :  .  ||62 |  :  ||
```

User-Defined Lexical Orders

A lexical order can be created for applications that require special collating sequences. If you can use one of the predefined lexical orders, you may wish to only skim this section.

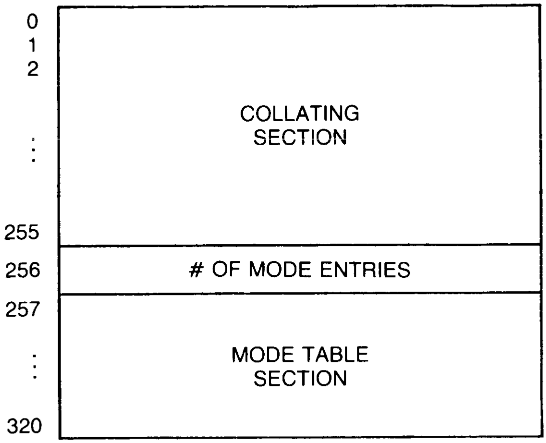
A program called LEX_AID has been supplied (on the BASIC Utilities Library disc) to simplify the creation of user-defined lexical orders. Before running the program it will be necessary to have an understanding of the terms used in this section. Using the LEX_AID program is described in the "BASIC Utilities Library" chapter of the *Installing and Maintaining the BASIC System* manual.

Basically, a 321 element (0 thru 320) INTEGER array is dimensioned, filled with sequence numbers and mode entries, and the new lexical order is established by the following statement.

```
   LEXICAL ORDER IS Table(*)
```

Where Table(*) is any valid INTEGER array name.

The following illustration shows the general construction of a user-defined lexical table created in an `INTEGER` array.



The first 256 elements (0 through 255) contain the sequence number to be used in place of the character's ASCII value. For special characters, a mode type and mode table pointer are also stored in these elements.

The next element (256) contains the number of entries in the mode table. This value can range from 0 (no mode table) thru 64 (a full mode table).

The remaining 64 elements (257 thru 320) contain the optional mode table entries assigned to special characters.

Sequence Numbers

Normally, comparing two strings results in the computer comparing the ASCII values of the characters. When the computer makes the string comparison "A"<"B", the ASCII value of "A" (65) is compared to the ASCII value of the letter "B" (66) resulting in the comparison: 65<66, which is true.

Now suppose that a new value (sequence number) could be assigned to each of the ASCII characters. We might wish to assign the letter "A" a sequence number greater than the sequence number assigned to the letter "B". If such an assignment were made, the comparison "A"<"B", would now be false.

Once a lexical order is invoked, if two strings are compared, the strings are first converted into two series of sequence numbers and the comparison is then based on the sequence numbers.

The `LEXICAL ORDER IS` statement's primary purpose is to assign a sequence number to each character. However, this is not always enough to handle certain character combinations and special cases encountered in other languages. Special characters have a mode entry included with the sequence number.

Mode Entries

Each of the first 256 array elements (0 thru 255) contains the sequence number to be used in place of the character's ASCII value. Optionally, a mode entry can be included.

Internally, an integer array element uses two bytes (16 bits) of memory. In the following diagram, the array element is divided into its upper, and lower bytes. The upper byte contains the sequence number and the lower byte is used if the character has a mode entry.

	upper byte	lower byte
array element	sequence number	optional mode entry

The lower byte is further divided into two parts. The upper-most 2-bits are used to represent one of the four mode types. The remaining 6-bits store an index (pointer) to the actual mode table entries. This method allows all the necessary information, for each character, to be stored as a single element in the INTEGER array.

	lower byte	
	mode type	mode table index

Mode Type

Any one of the following mode types can be assigned to a character.

- Don't Care Characters (Mode type: 0)
- "1 for 2" Character Replacements (Mode type: 1)
- "2 for 1" Character Replacements (Mode type: 2)
- Accent Priority (Mode type: 3)

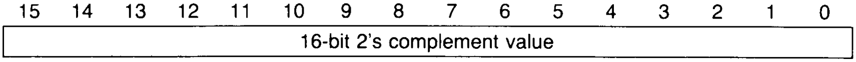
Mode Index

The mode index points to the actual mode table entry associated with the particular character. Up to 64 indexes are allowed (0 thru 63); however, some mode types use more than one table entry.

Bits, Bytes, and Mode Types

Each INTEGER array element stores a signed-integer in the range: −32768 thru 32767. Internally, the number is stored as a 16-bit 2's complement value.

Bits are usually numbered in descending order and include bit 0, so 16 bits are numbered as follows.



However, we want to store one of 256 possible sequence numbers and optionally, a mode type and mode table index. Since there are 256 characters used with the **LEXICAL ORDER IS** statement, and 8 bits are needed to store one of 256 possible values (2^8 = 256), it is convenient to think of the bits arranged as two bytes (a byte contains 8 bits).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
upper byte								lower byte							

The upper byte is used to hold the sequence number and the lower byte contains the mode entry information. The algorithm below will produce a signed 16-bit integer from two unsigned 8-bit bytes.

```
Integer = (256*Upper+Lower)-(Upper>127)*65536
```

The process can be reversed.

```
IF Integer<0 THEN Integer=Integer+65536
Upper = Integer DIV 256
Lower = Integer MOD 256
```

The lower byte is further divided into two groups. Two bits hold one of four mode types (2^2=4) and the remaining six bits are for one of 64 mode indexes (2^6=64).

	7	6	5	4	3	2	1	0
sequence number	type		index					

A "1 for 2" entry is signified by bit-6 being set. Therefore the value of the lower byte can range from 64 thru 126. (a "1 for 2" requires at least 2 entries.)

A "2 for 1" entry has bit-7 set. The value of the lower byte can range from 128 thru 191.

An "Accent priority" entry has both bit-6 and bit-7 set. The value of the lower byte ranges from 192 thru 255.

“Don’t Care” Characters

A character can be removed from the collation sequence. To mark a character as a “don’t care”, the mode type is 0 (the same as a regular character) but the mode table index is set to 1.

sequence number	type	index
any value	0	1

The mode index need not point to a valid table entry, but must be a “1” to indicate a “don’t care” character.

For example, the FRENCH lexical table lists the hyphen (-) as a “don’t care” character. Thus, the hyphen is ignored when a string comparison is being made. The entry appears:

	sequence number	type	index
(45)	45	0	1

You may wish to include “don’t care” characters in your own lexical tables. A string containing only “don’t care” characters will match the null string.

The following short program illustrates the operation of a “don’t care” character.

```
10 Return$="RESTORE"
20 Again$="RE-STORE"
30 !
40 LEXICAL ORDER IS ASCII
50 IF Restore$=Again$ THEN PRINT "True for ASCII"
60 LEXICAL ORDER IS FRENCH
70 IF Restore$=Again$ THEN PRINT "True for FRENCH"
80 END
```

Results:

```
True for FRENCH
```

“1 for 2” Character Replacement

This type of mode table entry indicates that one sequence number is to be used for two consecutive characters. It should be remembered that no characters are actually replaced by this operation, only that a single sequence number is to be used when the two characters are found adjacent to each other.

The following entry is placed in the collating section of the lexical table.

sequence number	type	mode index
normal sequence number	1	index

If a character marked as a “1 for 2” is found in a string, the next character is accessed and compared to the list of possible secondary characters in the mode table section.

(257 + index)	number of entries to check	
	second character	sequence number for this pair
	second character	sequence number for this pair

If the character does not match any of the secondary characters in the mode table, the original character's sequence number is used and processing continues. If a match is found, the sequence number for the pair is used and processing continues with the character following the secondary character.

For example, the SPANISH collating sequence has a “1 for 2” replacement for the letters “CH” or “Ch”. The letter “C” is marked as a “1 for 2” character. When the letter is encountered in a string, the next character is accessed and compared to the list of possible secondary characters (uppercase H and lowercase h). The appropriate sequence number is then used for the pair. If the character following the letter “C” is not found in the list of possible secondary characters, the sequence number for “C” is used and processing continues with the next character.

You can override a “1 for 2” character replacement by inserting a “Don't Care” character between the two characters that would otherwise be replaced by a single sequence number.

The SPANISH table entry for the character sequence "CH" is below. In the collating section, the first letter of the sequence has the following entry:

	sequence number	type	mode index
(67)	67 (C)	1	1
(68)	69 (D)	0	0

(257 + 1)	number of entries to check (2)	
(257 + 2)	second character (H)	sequence number for pair (68)
(257 + 3)	second character (h)	sequence number for pair (68)

The sequence number assigned to the two-character combination is greater than the sequence number for the letter "C" and less than the sequence number for the letter "D". Therefore, a word beginning with the characters "CH" will collate after all words starting with the letter "C" followed by any other character.

The following program shows the sorting order for the letters "CH" in the SPANISH lexical order.

```
5  DIM A$(3)[3]
10  A$(1)="CGA"
20  A$(2)="CHA"
30  A$(3)="CIA"
40  LEXICAL ORDER IS SPANISH
50  MAT SORT A$(*)
60  PRINT A$(*)
70  END
```

Produces:

```
CGA   CIA   CHA
```

It should be noted that a character may have more than one secondary character combination. This is demonstrated by having both upper and lower case entries. Other secondary characters could have been included in the same manner. The first mode table entry contains the number of secondary characters to check and must be in the range: 0 thru 63.

“2 for 1” Character Replacement

When a “2 for 1” mode entry is specified, it indicates that the character should be represented by two sequence numbers (as if there were two characters in the string). The first sequence number is stored with the character as usual. The mode index points to the mode table entry that contains the second sequence number to be used for that character.

sequence number	type	mode index
1st sequence number	2	index

The mode table entry actually contains two sequence numbers. If the original character was upper case, the next character in the string will determine whether the upper or the lower sequence number is used. If the next character in the string is uppercase, the upper sequence number is used as the second sequence number. Conversely, if the next character in the string is lowercase, the lower sequence number is used. If the original character was a lower case letter, the lower sequence number is always used.

	upper	lower
(257 + index)	2nd sequence number (UPC)	2nd sequence number (LWC)

Several “2 for 1” characters are in the GERMAN lexical order. For instance, the character “Ä” is equivalent to “AE” and has the following entry in the collating section.

	sequence number	type	mode index
(216)	65 (A)	2	index

The index points to the following entry in the mode table.

	upper	lower
(257 + index)	75 (E)	124 (e)

In some cases, such as the character “ß”, both upper and lower bytes contain the sequence number for the same character (s). This results in the same sequence numbers being generated regardless of the case of the next character.

Accent Priority

Accent Priority can be used as the final arbitrator of string comparisons. If you examine the lexical tables you will often find the same sequence number assigned to more than one character. Therefore, it is possible for two different strings to produce identical series of sequence numbers. The two strings will be considered equal unless at least one character, in each string, has been assigned different accent priorities.

Accent priority is established by assigning a value, in the range: 0 thru 63, to the character. Any character not already assigned a mode type may be assigned a priority. A priority of zero is assumed for all characters that haven't been assigned a priority.

sequence number	type	mode index
normal sequence number	3	priority

In the FRENCH lexical order, the characters: A, Á, and À have been assigned the same sequence number (64). Assume the characters were assigned the following priorities.

Character	**Priority**	
A	0	(default priority)
A	1	
A	2	

The characters can now be distinguished from one another and will collate in the following order.

A < Á < À

When two strings are compared, each string is first converted into a series of sequence numbers. The comparison is then determined (in most cases) by the greater sequence numbers or the longer series of sequence numbers.

In the event both strings produce identical series of sequence numbers, the series of priorities are checked. The string containing the characters with the higher priority is the greater string.

Subprograms and User-Defined Functions 6

Subprograms and User-Defined Functions 6

One of the most powerful constructs available in any language is the subprogram[1]. A subprogram can do everything a main program can do except that it must be invoked or "called" before it is executed, whereas a main program is executed by pressing the RUN key or executing the RUN command. In a sense, pressing the RUN key is how you "call" a main program. This chapter describes the benefits of using subprograms, and shows many of the details of using them.

Some Examples

The following program contains two subprograms and one user-defined function:

```
10    OPTION BASE 1
20    DIM Numbers(20)
30    CALL Build_array(Numbers(*),20)    ! Subprogram call.
40    CALL Sort_array(Numbers(*),20)     ! Subprogram call.
50    PRINT FNSum_array(Numbers(*),20)   ! User function call.
60    END
65    !
70    SUB Build_array(X(*),N)            ! Subprogram "Build_array".
80      ! X(*) is the array to be defined
90      ! N tells how many elements are in the array
100     ! (1 is assumed to be the lower index)
110     FOR I=1 TO N
120       DISP "ELEMENT #";I;
130       INPUT "?",X(I)
140     NEXT I
150   SUBEND
155   !
160   SUB Sort_array(A(*),N)            ! Subprogram "Sort_array".
170     ! A(*) is array to be sorted
180     ! N tells how many elements are in the array (1 is assumed
190     !  to be the lower bound)
200     ! Sort the array (elements 1-N) in increasing order
210     ! Algorithm used: Shell sort or Diminishing increment sort
220     ! Ref: Knuth, Donald E., The Art of Computer Programming,
230     !  Vol. 3 (Sorting and Searching), (Addison-Wesley 1973)
240     !  pp. 84-85
250     INTEGER T,S,H,I,J
```

[1] A user-defined function is a special form of subprogram.

```
260    REAL Temp
270    T=INT(LOG(N)/LOG(2))   ! # of diminishing increments
280    FOR S=T TO 1 STEP -1
290      H=2^(S-1)            ! ...16,8,4,2,1
300      FOR J=H+1 TO N
310        I=J-H
320        Temp=A(J)
330 Decide: IF Temp>=A(I) THEN Insert
340 Switch: A(I+H)=A(I)
350        I=I-H
360        IF I>=1 THEN Decide
370 Insert: A(I+H)=Temp
380      NEXT J
390    NEXT S
400  SUBEND
405  !
410  DEF FNSum_array(A(*),N)  !  User-defined function "Sum_array".
420    ! Add A(1)...A(N)
430    INTEGER I
440    REAL Array_total
450    FOR I=1 TO N
460      Array_total=Array_total+A(I)
470    NEXT I
480    RETURN Array_total
490  FNEND
```

Lines 10 through 60 are the main program. As you can see, it does nothing but call subprograms, which in turn do all the work. Line 70 is the header for the subprogram which asks the user to enter the values stored in his array. Notice that the main program has declared the array's name to be Numbers(*), but the subprogram uses the name X(*) to deal with the same array. The subprogram can name its variables whatever it wants without interfering with variables used outside the subprogram's context. The only variables that can be affected outside the subprogram's context are those passed through the parameter list (as shown here) or through COM (discussed later). In both cases, the matching between the subprogram and the outside world is done through the position of the variable(s) in the parameter list or COM block, not the actual name of the variable(s).

Starting at line 160 is the next subprogram which sorts the array into ascending order. The comments at the front of the subprogram serve to discuss the definition of the parameters used, and what effect the subprogram has on them. Also, the algorithm used is given, along with the proper reference material. It is an excellent idea to give a list of such pertinent details at the front of all subprograms. This makes debugging, modifying, optimizing, and re-using the subprogram much easier.

Starting at line 410 we see an example of a function subprogram. Functions are similar to SUB subprograms in concept. This particular example just adds the elements of the array together and returns the final value to the main program, which prints it.

Benefits of Subprograms

A subprogram has its own "context" or state that is distinct from a main program and all other subprograms. This means that every subprogram has its own set of variables, its own softkey definitions, its own DATA blocks, and its own line labels. There are several benefits to be realized by taking advantage of subprograms:

- The subprogram allows the programmer to take advantage of the *"top-down design" method of programming.* In this technique, the problem to be solved is broken up into a set of smaller and more easily solvable problems. These smaller problems can in turn be broken up into smaller problems yet, and so on. This technique has been shown to greatly improve the design, coding, and testing of programs, and will be discussed further at the end of the chapter.

- By *removing all the details of subtasks from the overall logic flow of the main program*, the program is much easier to read from the subprogram calls. The programmer can see at a high level what he's trying to accomplish, rather than immediately getting lost in the details of each little subtask.

- One of the most time-consuming parts of writing a program is *debugging* it, or forcing it to run correctly. The time consuming part of fixing bugs in a program is finding where the bug is in the first place. By using subprograms and *testing each one independently* of the others, it is easier to locate problems, and hence to fix them.

- Often, a programmer may want to perform the same task from several different areas of his program. For example, a set of readings may need to be taken from a voltmeter after each of four different input signals are fed through a circuit being tested. The same subprogram may be used to set up the voltmeter and take the readings, while different pieces of code would have to be used to set up the differing input conditions. Thus, subprograms can be used to *economize on the overall size of the program.*

- Finally, *libraries of commonly used subprograms can be assembled for widespread use.* Many different users doing diverse types of problems still may require some identical subprograms. For instance, an engineer may be using a subprogram to plot an array of data that he gathered from a spectrum analyzer, while the marketing person down the hall may be using the same subprogram to plot an array of data representing next year's sales forecast.

A Closer Look at Subprograms

The preceding examples only showed some of the general features of subprograms. This section shows a few of the details of using subprograms.

Calling and Executing a Subprogram

We have seen in the above examples how the two types of subprograms are called--SUBs are invoked explicitly using the CALL statement, while functions are invoked implicitly just by using the name in an expression, an output list, etc. A nuance of SUB subprograms is that the CALL keyword is optional when invoking a SUB subprogram. Thus our example of the main program which causes an array of numbers to be sorted could look like this:

```
10    OPTION BASE 1
20    DIM Numbers(20)
30    Build_array(Numbers(*),20)
40    Sort_array(Numbers(*),20)
50    PRINT FNSum_array(Numbers(*),20)
60    END
```

The omission of the CALL keyword when invoking a SUB subprogram is left solely to the discretion of the programmer; some will find it more aesthetic to omit CALL, others will prefer its inclusion. There are, however, three instances which require the use of CALL when invoking a subprogram:

CALL is required:

1. If the subprogram is called from the keyboard,
2. If the subprogram is called after the THEN keyword in an IF statement, or
3. In an ON..event..CALL statement.

Differences Between Subprograms and Subroutines

A *subroutine* and a *subprogram* are very different in HP Series 200/300 BASIC.

- The GOSUB statement transfers program execution to a subroutine. A subroutine is a segment of program lines *within the current context.* No parameters need to be passed, since it has access to all variables in the context (which is also the context in which the "calling" segment exists).
- The CALL statement transfers program execution to a subprogram, which is in a *separate context.* Subprograms can have pass parameters, and they can have their own set of local variables which are separate from all variables in all other contexts.

If you are a newcomer to HP's BASIC, be careful to distinguish between these two terms. They have been used differently in some other programming languages.

Subprogram Location

A subprogram is located after the body of the main program, following the main program's END statement. (The END statement must be the last statement in the main program except for comments.) Subprograms may not be nested within other subprograms, but are physically delimited from each other with their heading statements (SUB or DEF) and ending statements (SUBEND or FNEND).

Subprogram and User-Defined Function Names

A subprogram has a name which may be up to 15 characters long, just as with line labels and variable names. Here are some legal subprogram names:

```
Initialize
Read_dvm
Sort_2_d_array
Plot_data
```

Because up to 15 characters are allowed for naming subprograms, it is easy and convenient to name subprograms in such a way as to reflect the purpose for which the subprogram was written.

Difference Between a User-Defined Function and a Subprogram

A SUB subprogram (as opposed to a function subprogram) is invoked explicitly using the CALL statement. A function subprogram is called implicitly by using the function name in an expression. It can be used in a numeric or string expression the same way a constant would be used, or it can be invoked from the keyboard. A function's purpose is to return a single value (either a REAL number, a COMPLEX number or a string).

There are several functions that are built into the BASIC language which can be used to return values, such as SIN, SQR, EXP, etc.

```
Y=SIN(X)+Phase
Root1=(-B+SQR(B*B-4*A*C))/(2*A)
```

Using the capability of defining your own function subprograms, you can essentially extend the language if you need a feature not provided in BASIC.

```
X=FNFactorial(N)
Angle=FNAtn2(Y,X)
```

A general rule of thumb for using subprograms is that if you want to take a set of data and analyze it to generate a single value, then you probably want to implement the subprogram as a function. On the other hand, if you want to actually change the data itself, generate more than one value as a result of the subprogram, or perform any sort of I/O activity, it is better to use a SUB subprogram.

Numeric Functions and String Functions

A function is allowed to return either a REAL or COMPLEX value or a string value. Above, we saw some examples of functions returning real numbers. Let's examine one which returns a string. There are two primary differences: the first is that a $ must be added to the name of a function which is to return a string. This is used both in the definition of the function (the DEF statement) and when the function is invoked. The second difference is that the RETURN statement in the function returns a string instead of a number.

```
.
.
.
200   PRINT FNAscii_to_hex$(A$)
.
.
.
1550  DEF FNAscii_to_hex$(A$)
1560  ! Each ASCII byte consists of two hex
1570  !   digits; pretty formatting dictates that
1580  !   a space be inserted between every pair
```

```
1590  !   of hex digits.  Thus, the output string
1600  !   will be three times as long as the input
1610  !   string.
1620  !
1630  ! upper four bits     lower four bits
1640  ! UUUU LLLL           UUUU LLLL
1650  ! shift 4 bits        0000 1111 mask (15)
1660  ! 0000 UUUU           0000 LLLL final
1670  !
1680  INTEGER I,Length,Hexupper,Hexlower
1690  Length=LEN(A$)
1700  ALLOCATE Temp$[3*Length]
1710  FOR I=1 TO Length
1720     Hexupper=SHIFT(NUM(A$[I]),4)
1730     Hexlower=BINAND(NUM(A$[I],15)
1740     Temp$[3*I-2;1]=FNHex$(Hexupper)
1750     Temp$[3*I-1;1]=FNHex$(Hexlower)
1760     Temp$[3*I;1]=" "
1770  NEXT I
1780  RETURN Temp$
1790  FNEND
1800  DEF FNHex$(INTEGER X)
1810  ! Assume 0<=X<=15)
1820  ! Return ASCII representation of the
1830  !   hex digit represented by the four
1840  !   bits of X.
1850  ! If X is between 0 and 9, return
1860  !   "0"..."9"
1870  ! If X > 9, return "A"..."F"
1880  IF X<=9 THEN
1890     RETURN CHR$(48+X) ! ASCII 48 through 57
1900                       !   represent "0" - "9"
1910  ELSE
1920     RETURN CHR$(55+X) ! ASCII 65 through 70
1930                       !   represent "A" - "F"
1940  END IF
1950  FNEND
```

Lines 200, 1740, and 1750 show examples of how to call a string function. Lines 1550 and 1800 show where the two string function subprograms begin. Notice that the program could be optimized slightly by deleting lines 1720 and 1730 and modifying lines 1740 and 1750:

```
1740     Temp$[3*I-2;1]=FNHex$(SHIFT(NUM(A$[I]),4))
1750     Temp$[3*I-1;1]=FNHex$(BINAND(NUM(A$[I],15))
```

Thus it is perfectly legal to use expressions in the pass parameter list of a subprogram. (By the way, such expressions may also invoke function subprograms.)

Program/Subprogram Communication

As mentioned earlier, there are two ways for a subprogram to communicate with the main program or with other subprograms:

- By passing parameters
- By sharing blocks of common (COM) variables.

Parameter Lists

There are two places where parameter lists occur:

- The **pass parameter list** is in the CALL statement or FN call:

  ```
  30  CALL Build_array(Numbers(*),20)    ! Subprogram call.

  50  PRINT FNSum_array(Numbers(*),20)   ! User-defined function call.
  ```

  It is known as the pass parameter list because it specifies what information is to be passed to the subprogram.

- The **formal parameter list** is in the SUB or DEF FN statement that begins the subprogram's definition:

  ```
  70   SUB Build_array(X(*),N)  ! Subprogram "Build array".

  410  DEF FNSum_array(A(*),N)  !  User-defined function "Sum_array".
  ```

  This is known as the formal parameter list because it specifies the form of the information that can be passed to the subprogram.

Formal Parameter Lists

The formal parameter list is part of the subprogram's definition, just like the subprogram's name. The formal parameter list defines:

- The **number of values** that may be passed to a subprogram
- The **types of those values** (string, INTEGER, REAL or COMPLEX, and whether they are simple or array variables; or I/O path names)
- The **variable names the subprogram will use** to refer to those values. (This allows the name in the subprogram to be different from the name used in the calling context.)

The subprogram has the power to demand that the calling context match the types declared in the formal parameter list—otherwise, an error results.

Pass Parameter Lists

The calling context provides a pass parameter list which corresponds with the formal parameter list provided by the subprogram. The pass parameter list provides:

- The **actual values** for those inputs required by the subprogram.
- **Storage** for any values to be returned by the subprogram (pass by reference parameters only).

It is perfectly legal for both the formal and pass parameter lists to be null (non-existent).

Passing By Value vs. Passing By Reference

There are two ways for the calling context to pass values to a subprogram:

- Pass by value—the calling context supplies a value and nothing more.
- Pass by reference—the calling context actually gives the subprogram access to the calling context's value area (which is essentially access to the calling context's variable).

The distinction between these two methods 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 whether its parameters are passed by value or passed by reference. That is determined by the calling context's pass parameter list. For instance, in the example below, the array Numbers(*) is passed by reference, while the quantity 20 is passed by value.

```
30 CALL Build_array(Numbers(*),20) ! Subprogram call.
```

The general rules for passing parameters are as follows:

- 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** for that parameter.

Note that enclosing a variable in parentheses is sufficient to create an expression and that literals are expressions. Using pass by value, it is possible to pass an INTEGER expression to a REAL formal parameter (the INTEGER is converted to its REAL representation) without causing a type mismatch error. Likewise, it is possible to pass a REAL expression to an INTEGER formal parameter (the value of the expression is rounded to the nearest INTEGER) without causing a type mismatch error (an integer overflow error is generated if the expression is out of range for an INTEGER).

Example Pass and Corresponding Formal Parameter Lists

Here is a sample *formal* parameter list showing which types each parameter demands:

```
SUB Read_dvm(@Dvm,A(*),INTEGER Lower,Upper,Status$.Errflag)
```

`@Dvm` This is an I/O path name which may refer to either an I/O device or a mass storage file. Its name here implies that it is a voltmeter, but it is perfectly legal to redirect I/O to a file just by using a different ASSIGN with `@Dvm`.

`A(*)` This is a REAL array. Its size is declared by the calling context. Without MAT, there is no way to find the size of the array except through information supplied explicitly by the calling context; hence the parameters `Lower` and `Upper`.

`Lower`
`Upper` These are declared here to be INTEGERs. Thus, when the calling program invokes this subprogram, it must supply either INTEGER variables or INTEGER expressions, or an error will occur.

`Status$` This is a simple string which presumably could be used to return the status of the voltmeter to the main program. The length of the string is defined by the calling context.

`Errflag` This is a REAL number. The declaration of the string `Status$` has limited the scope of the INTEGER keyword which caused `Lower` and `Upper` to require INTEGER pass parameters.

Let's look at our previous example from the calling side (which shows the *pass* parameter list):

```
CALL Read_dvm(@Voltmeter,Readings(*),1,400,Status$,Errflag)
```

@Voltmeter — This 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(*) — This matches the array A(*) in the subprogram's formal parameter list. Arrays, too, are always passed by reference.

1, 400 — These are the values passed to the formal parameters Lower and Upper. Since constants are classified as expressions rather than variables, these parameters have been passed by value. Thus, if the subprogram used either Lower or Upper on the left-hand side of an assignment operator, no change would take place in the calling context's value area.

Status$ — This is passed by reference here. If it were enclosed in parentheses, it would be passed by value. Notice that if it were passed by value, it would be totally useless as a method for returning the status of the voltmeter to the calling context.

Errflag — This is passed by reference.

OPTIONAL Parameters

Another important feature of formal parameter lists is the OPTIONAL keyword. Any formal parameter list (the one defining the subprogram) may contain the keyword OPTIONAL somewhere, although it isn't required to. The OPTIONAL keyword indicates that any parameters that follow it are not required in the pass parameter list of a calling context—they are optional. 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 will be generated. The rules requiring matching of parameter types apply to OPTIONAL parameters as well as to ordinary parameters. There is a standard function called NPAR which can be used inside the subprogram to find out how many pass parameters the calling context actually did use. (NPAR will return 0 if used inside the main program, or if no parameters were passed to a subprogram.)

The OPTIONAL/NPAR combination is very effectively used in situations requiring external instrument setups. Most instruments have several different ranges, modes, settings, etc., which can be used depending upon the requirements of the user. Often, the user doesn't require the entire flexibility the instrument has to offer, and would rather use some reasonable defaults.

Consider the HP 3437A Digital Voltmeter. Among other things, this device has two data formats (packed and ASCII), three trigger modes (internal, external, and hold/manual), three voltage ranges (0.1V, 1V, and 10V), and also has programmable values for delay between readings and number of readings taken. Naturally, the values used for the various settings will depend entirely upon the application for which the voltmeter is being used, but let's make some assumptions:

- The values for delay and number of readings are going to be changed frequently, so they will not be OPTIONAL parameters.
- Of the remaining OPTIONAL parameters, the range is most likely to be altered.

A reasonable setup routine for the voltmeter might look like this:

```
2010 SUB Setup_dvm(@Dvm,INTEGER Readings,REAL Delay, OPTIONAL INTEGER Prange,
Ptrigger,Pformat)
2020 SELECT NPAR
2030 CASE 3
2040    Format=1                               ! Default ASCII format
2050    Trigger=1                              ! Default internal trigger
2060    Range=2                                ! Default 1 volt range
2070 CASE 4
2080    Format=1
2090    Trigger=1
2100    Range=Prange
2110 CASE 5
2120    Format=1
2130    Trigger=Ptrigger
2140    Range=Prange
2150 CASE 6
2160    Format=Pformat
2170    Trigger=Ptrigger
2180    Range=Prange
2190 END SELECT
2200 OUTPUT @Dvm;"N"&VAL$(Readings)&"SD"&VAL$(Delay)&"SR"&VAL$(Range)&"T"&
VAL$(Trigger)&"F"&VAL$(Format)
2210 SUBEND
```

Legal invocations of the Setup_dvm subprogram are:

```
570  Setup_dvm(@Dvm,100,.001)       ! Default Range,Trigger,Format
630  Setup_dvm(@Dvm,500,.05,3)      ! Default Trigger,Format
850  Setup_dvm(@Dvm,50,.005,1,2)    ! Default Format
1010 Setup_dvm(@Dvm,70,.075,2,1,2)  ! Explicitly declare all values
```

Notice in the example above that local variables are used instead of the formal parameters. This is because it is illegal to use an OPTIONAL parameter variable if that variable was not passed from the calling context.

Other applications of the OPTIONAL/NPAR feature are limited only by the imagination, but here are a few ideas:

- Write a subprogram which sorts an array in ascending order unless an OPTIONAL parameter tells it to sort in descending order.
- Write a rootfinder routine which has an acceptance tolerance of $\pm 10^{-6}$ unless overridden with an OPTIONAL parameter.
- Write a program which keeps track of departmental expenses, including the account billed, the item or service purchased, the person incurring the expense, and optionally, the person authorizing the expense.

COM Blocks

Since we've discussed parameter lists in detail, let's turn now to the other method a subprogram has of communicating with the main program or with other subprograms, the COM block.

There are two types of COM (or common) blocks: blank and labeled. Blank COM is simply a special case of labeled COM (it is the COM whose name is nothing) with the exception that blank COM must be declared in the main program, while labeled COM blocks don't have to be declared in the main program. Both types of COM blocks simply declare blocks of data which are accessible to any context having matching COM declarations.

A blank COM block might look like this:

```
10   OPTION BASE 1
20   COM Conditions(15),INTEGER,Cmin,Cmax,@Nuclear_pile,Pile_status$[20],
Tolerance
```

A labeled COM might look like this:

```
   30 COM /Valve/ Main(10),Subvalves(10,15),@Valve_ctrl
```

A COM block's name, if it has one, will immediately follow the COM keyword, and will be set off with slashes, as shown above. The same rules used for naming variables and subprograms are used for naming COM blocks.

Any context need only declare those COM blocks which it needs to have access to. If there are 150 variables declared in 10 COM blocks, it isn't necessary for every context to declare the entire set—only those blocks that are necessary to each context need to be declared. COM blocks with matching names must have matching definitions. As in parameter lists, matching COM blocks is done by position and type, not by name.

COM vs. Pass Parameters

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 are set to undefined after pushing the RUN key, or upon entering a subprogram. This is true of COM the first time the RUN key is pressed, but after COM block variables are defined, they retain their values until:

    - SCRATCH A or SCRATCH C is executed,
    - A statement declaring a COM block is modified by the user, or
    - A new program is brought into memory using the GET or LOAD commands which doesn't match the declaration of a given COM block, or which doesn't declare a given COM block at all.

- COM blocks can be arbitrarily large. 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, the invocation or subprogram header, and possibly (in the case of a function) a string or numeric expression. Depending upon the situation, this can impose a restriction on the size of your parameter lists.

    COM blocks can take as many statements as necessary. COM statements can be interwoven with other statements (though this is considered a poor practice). All COM statements within a context which have the same name will be part of the definition of that COM block.

- COM blocks can be used for communicating 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. For instance, one routine might be responsible for setting the voltage range on a voltmeter, while another routine may need to know what the current voltage range is in order to set up the scale on a graph properly.

- 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.

- 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. A machine which tests capacitors in an incoming inspection department may require calibration after every 100 tests are performed. If the subprogram which does the testing has a way to count how many tests it has already performed (using a labeled COM block), then this task can be left to the testing routine, simplifying the rest of the system.

- COM blocks allow subprograms to share data without the intervention of the main program. Subprogram libraries may consist of elaborate relationships of both programs and data structures. In many cases, a major portion of the data structures are only used for support of the task being performed, rather than being integral to the task itself. Thus the main program does not need to declare the supportive data structures.

  Examples of this situation might include data base management libraries (hashing tables may need to be maintained for accessing data quickly) or three-dimensional graphics libraries (window, viewport, and clip information need to be kept, as well as object definitions and related transformations).

Hints for Using COM Blocks

Any COM blocks needed by your program must be resident in memory at prerun time (prerun is caused by pressing [RUN], executing a RUN command, executing LOAD or GET from the program, or executing a LOAD or GET from the keyboard and specifying a run line.) Thus if you want to create libraries of subprograms which share their own labeled COM blocks, it is wise to collect all the COM declarations together in one subprogram to make it easy to append them to the rest of the program for inclusion at prerun time. (The subprogram need not contain anything but the COM declarations.)

COM can be used to communicate between programs which overlay each other using LOAD or GET statements, if you remember a few rules:

1. COM blocks which match each other exactly between the two programs will be preserved intact. "Matching" requires that the COM blocks are named identically (except blank COM), and that corresponding blocks have exactly the same number of variables declared, and that the types and sizes of these variables match.
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 destroyed.
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. All values stored in that COM block under the old program are destroyed.
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 are set to undefined.

The first occurrence in memory of a COM block is used to define or set up the block. Subsequent occurrences of the COM block must match the defining block, both in the number of items, and the types of the items. In the case of strings and arrays, the actual sizes need be specified only in the defining COM blocks. Subsequent occurrences of the COM blocks may either explicitly match the size specifications by re-declaring the same size, or they may implicitly match the size specifications. In the case of strings, this is done by not declaring any size, just declaring the string name. In the case of arrays, this is done by using the (*) specifier for the dimensions of the array instead of explicitly re-declaring the dimensions.

Consider the following COM block definition:

```
10   COM /Dvm_state/ INTEGER Range,Format,N,REAL Delay,Lastdata(1:40),Status$[20]
```

The following occurrence of the same COM block within a subprogram matches the COM block explicitly and is legal:

```
2000 COM /Dvm_state/ INTEGER Range,Format,N,REAL Delay,Lastdata(1:40),Status$[20]
```

The following block within a different subprogram uses implicit matching and is also legal:

```
4010 COM /Dvm_state/ INTEGER Range,Format,N,REAL Delay,Lastdata(*),Status$
```

The following declaration is illegal, since it uses explicit size specifications for the array and string which do not match the original definition from line 10.

```
5020 COM /Dvm_state/ INTEGER Range,Format,N,REAL Delay,Lastdata(1:30),Status$[15]
```

The following declaration is also illegal, since it violates the types set forth by the defining block.

```
6010 COM /Dvm_state/ Range,Format,N,REAL Delay,Lastdata(*),Status$
```

In general, the implicit size matching on arrays and strings is preferable to the explicit matching because it makes programs easier to modify. If it becomes necessary to change the size of an array or string in a COM block, it only needs to be changed in one statement, the one which defines the COM block. If all other occurrences of the COM block use the (*) specifier for arrays, and omit the length field in strings, none of those statements will have to be changed as a result of changing an array or string size.

Context Switching

As mentioned in the introduction to this chapter, a subprogram has its own **context** or state which is distinct from a main program and all other subprograms. In between the time that a CALL statement is executed (or an FN name is used) and the time that the first statement in the subprogram gets executed, the computer performs a "prerun" on the subprogram. This "entry" phase is what defines the context of the subprogram. The actions performed at subprogram entry are similar, but not identical, to the actual prerun performed at the beginning of a program. Here is a summary:

- The calling context has a DATA pointer which points to the next item in the current DATA block which will be used the next time a READ is executed (assuming of course that a DATA block even exists in the calling program). This pointer is saved away whenever a subprogram is called, and then the DATA pointer is reset to the first DATA statement in the new subprogram context.

- The RETURN stack for any GOSUBs in the current context is saved and set to the empty stack in the new context.

- The system priority of the current context is saved, and the called subprogram inherits this value. Any change to the system priority which takes place within the subprogram (or any of the subprograms which it calls in turn) is purely local, since the system priority is restored to its original value upon subprogram exit. This is an important consideration: if the subprogram is called as a result of an event-initiated GOSUB/CALL statement, any ON <event> GOTO/GOSUB/CALL/RECOVER condition set up in the called subprogram must have a higher priority assigned to it than the event responsible for the subprogram's invocation. Otherwise, the event is guaranteed **not** to cause an end of line branch. See the "EVENTS" chapter of *BASIC Interfacing Techniques* for a description of system priority.

- Any event-initiated GOTO/GOSUB statements are disabled for the duration of the subprogram. If any of the specified events occur, this will be logged, but no action will be taken. (The fact that an event did occur will be logged, but only once—multiple occurrences of the same event will not be serviced.) Upon exiting the subprogram, these event-initiated conditions will be restored to active status, and if any of these events occurred while the subprogram was being executed, the proper branches will be taken.

- Any event-initiated CALL/RECOVER statements are saved away upon entering a subprogram, but the subprogram still inherits these ON conditions since CALL/RECOVER are global in scope. However, it is legal for the subprogram to redefine these conditions, in which case the original definitions are restored upon subprogram exit.

- The current value of OPTION BASE is saved, and the value for the subprogram (0 or 1, explicitly declared or defaulted) is used.
- The current DEG or RAD mode for trigonometric operations and graphics rotations is stored away. The subprogram will inherit the current DEG or RAD setting, but if it gets changed within the subprogram, the original setting will be restored when the subprogram is exited.

Variable Initialization

Space for all arrays and variables declared is set aside, whether they are declared explicitly with DIM, REAL, INTEGER, or COMPLEX, or implicitly just by using the variable. The entire value area is initialized as part of the subprogram's prerun. All numeric values are set to zero, all strings are set to the null string, and all I/O path names are set to undefined.

Subprograms and Softkeys

ON KEYs are a special case of the event-initiated conditions that are part of context switching. They are special because they are the only event conditions which give visible evidence of their existence to the user through the softkey labels at the bottom of the CRT. These key labels are saved away just as the event conditions are, and the labels get restored to their original state when the subprogram is exited, regardless of any changes the subprogram made in the softkey definitions. This means the programmer doesn't have to make any special allowances for re-enabling his keys and their associated labels after calling a subprogram which changes them—the language system handles this automatically.

It is important to remember that the called subprogram inherits the softkey labels. All the keys are still active in some sense; ON KEY...CALL/RECOVER will cause their original program branches to take place immediately if the proper key is pressed, and ON KEY...GOTO/GOSUB will log the fact that a key is pressed until the subprogram is exited, at which time the proper branch will occur. This latter case may cause some consternation on the part of the user if he presses a softkey expecting immediate action and nothing happens since the key was temporarily disabled due to a called subprogram. If the called subprogram is expected to take a noticeably long time to execute, it might be a good idea to explicitly remove the labels from the disabled softkeys using the OFF KEY statement. Thus, the user won't expect anything to happen as a result of pressing a softkey. This technique is also useful for guaranteeing that a given subprogram is **not** interrupted prematurely. (The DISABLE statement is useful for preventing program branches as a result of an event-initiated happening, although it will not turn off the softkey labels.)

Subprograms and the RECOVER Statement

The event-initiated RECOVER statement allows the programmer to cause the program to resume execution at any given place in the context defining the ON...RECOVER as a result of a specified event occurring, regardless of subprogram nesting.

Thus, if a main program executes an ON...RECOVER statement (for example a softkey or an external interrupt from the SRQ line on an HP-IB), and then calls a subprogram, which calls a subprogram, which calls a subprogram, etc., program execution can be caused to immediately resume within the main program as a result of the specified event happening.

By way of illustration, consider the following example. Suppose you are performing an exhaustive component test on a circuit board. The program may be designed like so:

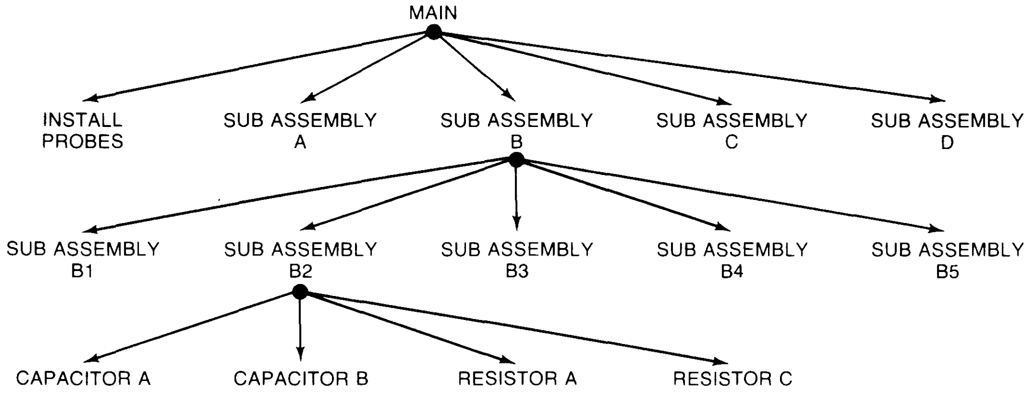


Figure 6-1. Program Design

When lunch break comes around, you may want to halt the current test so you can use the computer to play chess, or your boss might wander by and want to see the results of the rest of the tests performed this week. In either case, if the test program is nested three or four levels deep in subprograms, it might take a while for the test to complete. By defining a softkey to RECOVER to the main program, you can instantly terminate the test at any time, and make the computer available for something else. The RECOVER will discard anything being done in any of the subprograms between the context declaring the event-initiated RECOVER, and the subprogram being executed when the specified event occurs.

Again, the DISABLE statement can be used within any subprograms in which it is critical not to allow interruptions.

Calling Subprograms from the Keyboard

Functions and subprograms can be called by using CALL and FN at the keyboard. There are some restrictions:

- Since variables cannot be created by the user from the keyboard (variables can only be defined by the program), it is legal to use only parameters that already exist in the current context.
- Constants may be used in the pass parameter list.
- When calling a SUB subprogram from the keyboard, the CALL keyword must be used.

The "MEM_UTILS" utility on the *BASIC Utilities* disc has examples of calling subprograms from the keyboard. The program is explained in the "Utilities" chapter of *Installing and Maintaining the BASIC System.*

Speed Considerations

In some programs, speed is of the essence. In these cases, programmers will be reluctant to incur any unnecessary overhead in executing their task. There is a certain amount of overhead incurred in calling subprograms, although the overhead is fairly small, and shouldn't be an impediment to the use of subprograms. ("Overhead" is loosely defined to be the time it takes to perform those activities which aren't explicitly asked for by the user's program, but which are still necessary to keep the user's program running in a correct manner. The tasks discussed earlier under context switching are an excellent example of such overhead.)

Let's look at how much time it takes just to get in and out of the subprogram regardless of the task being performed by the subprogram. (The times in this discussion are approximate and apply to Series 200 computers with an 8 MHz MC68000 processor, without an HP 98635A floating-point math card or MC68881 co-processor.)

The time it takes to enter a subprogram depends upon the number of parameters being passed, the types of parameters being passed, and the number of variables declared local to the subprogram itself. To get in and out of a subprogram which has no parameters and which does nothing (in other words, a SUB followed by a SUBEND) takes 572 microseconds, meaning if you call it 1748 times, you'll use up about 1 second. (By way of comparison, 572 microseconds is about what it takes to perform four floating-point additions. To perform four floating-point additions and store the result from each one in a variable will take about 1080 microseconds, or just over a millisecond.)

Table 6-1. Subprogram Entry Execution Speed

Entry Conditions	Approximate Execution Speed[1]
No parameters	572 μsec.
1 simple numeric	+ 105 μsec.
1 simple string	+ 128 μsec.
1 numeric array	+ 141 μsec.
1 string array	+ 141 μsec.
1 I/O path name	+ 123 μsec.
OPTION BASE in sub	+ 31 μsec.
REAL or INTEGER in sub	+ 32 μsec.
1st numeric array declaration	+ 18 μsec.
other numeric array declarations	+ 11 μsec.
1st string array declaration	+ 21 μsec.
other string array declarations	+ 12 μsec.

As you can see from the table, subprograms are a bargain in terms of speed. The relatively small amount of overhead required for invoking a subprogram is more than made up for by the benefits to be derived.

[1] The times in this table are approximate and apply to Series 200 computers with an 8 MHz MC68000 processor, without an HP 98635A floating-point math card or MC68881 co-processor.

Using Subprogram Libraries

This section shows some of the tasks involved in using and managing subprogram libraries.

Why Use Subprogram Libraries?

As mentioned earlier, subprograms are also convenient for use in creating and distributing libraries of commonly used feature sets. They are also handy when you have a large program, along with sizable data arrays, which could potentially require more memory than is currently installed in your computer. You can break the program up into subprograms, each of which may be programmatically loaded, called, and then deleted in order to conserve memory. This section outlines some of the operations you will need to perform in creating, using, and maintaining subprogram libraries.

Listing the Subprograms in a PROG File

You can determine which subprograms are in a PROG file by performing a CAT on the file:

```
CAT "ProgFile"
```

The system returns a list of the subprograms and user-defined functions in the file, along with other information, such as the amount of memory required for each subprogram. (See the *BASIC Language Reference* description of CAT for details.)

Loading Subprograms

If you already have subprograms stored in PROG file(s), there are several options to choose from in loading them into memory:

- If you want to load a specific subprogram from a PROG file, you would use something like this:

  ```
  LOADSUB Sub_name FROM "File"
  ```

- If you want to load all the subprograms from a specific PROG file, you would use the LOADSUB ALL FROM statement.

  ```
  LOADSUB ALL FROM "File"
  ```

- And, if you wanted to see which subprograms are still missing or load all those still needed, you would use something like this:

  ```
  LOADSUB FROM "File"
  ```

  (Note that this is statement is **not** programmable; that is, it cannot appear in a program line.)

Loading Subprograms One at a Time

Suppose your program has several options to select from, and each one needs many subprograms and much data. All the options, however, are mutually exclusive; that is, whichever option you choose, it does not need anything that the other options use. This means that you can clean up everything you've used when you are finished with that option.

If all of your subprograms can be put into one file, you can selectively retrieve them as needed with this sort of statement:

```
LOADSUB Subprog_1 FROM "SUBFILE"
LOADSUB Subprog_2 FROM "SUBFILE"
LOADSUB FNNumeric_fn FROM "SUBFILE"
LOADSUB FNString_function$ FROM "SUBFILE"
```

Note that only one subprogram per line can be loaded with this form of LOADSUB. If, for any program option, you need so many subprograms that this method would be cumbersome, you could use the following form of the command.

Loading Several Subprograms at Once

For this method, you store **all** the subprograms needed for each option in its own file. Then, when the program's user selects Program Option 1, you could have this line of code execute:

```
LOADSUB ALL FROM "OPT1SUBFL"
```

and if the user selects Option 2,

```
LOADSUB ALL FROM "OP2SUBFL"
```

and so forth.

There is one other form of LOADSUB, but it cannot be used programmatically. This is covered next.

Loading Subprograms Prior to Execution

In the LOADSUB FROM form, for which you need PDEV, neither ALL nor a subprogram name is specified in the command. This is used prior to program execution. It looks through the program in memory, notes which subprograms are needed (referenced) but not loaded, goes to the specified file and attempts to load all such subprograms. If the subprograms are found in the file, they are loaded into memory; if they are not, an error message is displayed and a list of the subprograms still needed but not found in the file is printed.

This can be handy in two ways. The first and obvious way is that subprograms can be loaded quickly. The other way is this: suppose that you are developing a program and as you are coding, you realize you need a subprogram that does such-and-such. But your train of thought is chugging along so smoothly, you do not want to interrupt your coding of the routine you are working on to do the other little subprogram. But when the big one is done, you have forgotten all about coding the little one. If you suspect you've done this, the LOADSUB FROM command is very useful. Type a LOADSUB FROM command where the file name is a file in which you **know** there are none of the subprograms you need (perhaps a null PROG file). Of course, no subprograms will be loaded, but *a list of those yet undefined will be printed.* These are the ones you still need to code. Naturally, if you have already coded them and stored them somewhere, go get them. But if you haven't, this is a simple way of listing those still to be entered.

Any COM blocks declared in subprograms brought into memory with a LOADSUB by a running program must already have been declared. LOADSUB does not allow new COM blocks to be added to the ones already in memory. Furthermore, any COM blocks in the subprograms brought in must match a COM block in memory in both the number and type of the variables. Otherwise, an error occurs.

Note

If a main program is in a file referenced by a LOADSUB, it will **not be loaded**; only subprograms can be loaded with LOADSUB. Main programs are loaded with the LOAD command.

With all this talk of loading subprograms from files, one question arises: How do you get the subprograms *in* the file? Easily: type in the subprograms you want to be in one file, and then STORE them with the desired file name. You must use STORE and not SAVE, because the LOADSUB looks for a PROG-type file. If you can't type in your subprograms error-free the first time (and who can?), what you can do is this: type in your program with all the subprograms it needs and debug them. **After storing *everything* in a file for safekeeping**, delete what you do **not** want in the file, and STORE everything else in the subprogram file from which you will later do a LOADSUB. In this way, you know the subprograms will work when you load them.

Deleting Subprograms

The utility of the LOADSUB commands would be greatly reduced if one could not delete subprograms from memory at will. So, there is a way to delete subprograms during execution of a program: DELSUB. If you want to delete only selected ones, you could type something like this (you can also execute these statements in programs):

```
DELSUB Sort_data,Print_report,FNPoly_solve
```

If you are sure of the positioning of the subprograms in memory, here is a method of deleting whole groups of subprograms:

```
DELSUB Print_report TO END
```

You can combine these methods:

```
DELSUB Sort_data,Print_report,FNGet_name$ TO END
```

The subprograms to be deleted do not have to be contiguous in memory, nor does the order in which you specify the subprograms in a DELSUB statement have to be the order in which they occur in memory. The computer deletes each subprogram before moving on to the next name.

If there are any comments after an FNEND or SUBEND, but before the next SUB or DEF FN, these will be deleted as well as the rest of the subprogram body.

If the computer attempts to delete a non-existent subprogram, an error occurs, and the DELSUB statement is terminated. This means that subprograms whose names are listed after the error-causing name will not be deleted.

A subprogram can be deleted only if it is not currently active and if it is not referenced by a currently active ON RECOVER/CALL statement. This means:

1. A subprogram cannot delete itself.
2. A subprogram can not delete the subprogram that called it, either directly or indirectly. (Otherwise it wouldn't have anywhere to return to when finished!)

Between the time that a subprogram is entered and the time it is exited, the computer keeps track of an *activation record* for that subprogram. Thus, if a subprogram calls a subprogram that calls a subprogram, etc., none of the subsequently-called subprograms can delete the original one or any of the ones in between because the system knows from the activation record that control will eventually need to return to the original calling context. A similar situation exists with active event-initiated CALL/RECOVER statements. As long as the possibility of the specified event occurring exists, the system will not let the subprogram be deleted. In essence, the system will not let you execute two mutually-exclusive, contradictory commands simultaneously.

Editing Subprograms

Inserting Subprograms

There are some rules to remember when inserting SUB and DEF FN statement in the middle of the program. All DEF FN and SUB statements must be appended to the **end** of the program. If you want to insert a subprogram in the middle of your program because your prefer to see it listed in a given order, you must perform the following sequence:

1. STORE the program.
2. Delete all lines **above** the point where you want to insert your subprogram (refer to the DEL statement).
3. STORE the remaining segment of the program in a new file.
4. LOAD the original program stored in step 1.
5. Delete all lines **below** the point where you want to insert your subprogram.
6. Type in the new subprogram.
7. Do a LOADSUB ALL from the new file created in step 3.

With the PDEV binary, the job is much easier:

1. Write your new subprogram *at the end* of the program.
2. Perform a MOVELINES command where:
   a. The Starting Line in the MOVELINES command is the line which you want to immediately follow your new subprogram,
   b. The Ending Line in the MOVELINES command is the line immediately prior to the SUB or DEF FN of the new subprogram, and
   c. The Destination Line is any line number greater than the highest line number currently in memory.

In either case there is an optional final step. It is not *required* that you do a REN to renumber the program at this point, but often it is desirable to close up the void left in the program line numbering which resulted from the block of subprograms being moved to the end of memory.

Deleting Subprograms

It is not possible to delete either DEF FN or SUB statements with the DEL LN or Delete line key unless you first delete all the other lines in the subprogram. This includes any comments after the SUBEND or FNEND. Another way to delete DEF FN and SUB statements is to delete the entire subprogram, up to, but **not** including, the next SUB or DEF FN line (if any). This can be done either with the DEL command, or with the DELSUB command.

Merging Subprograms

If you want to merge two subprograms together, first examine the two subprograms carefully to insure that you don't introduce conflicts with variable usage and logic flow. If you've convinced yourself that merging the two subprograms is really necessary, here's how you go about it:

1. SAVE everything in your program **after** the SUB or DEF FN statement you want to delete.
2. Delete everything in your program from the unwanted SUB statement to the end.
3. GET the program segment you saved in step 1 back into memory, taking care to number the segment in such a way as not to overlay the part of the program already in memory.

Once again, with PDEV, your job is greatly simplified:

Execute a MOVELINES command in which you move everything from one subprogram—**excluding the SUB/DEF FN and SUBEND/FNEND statements**—into the desired position in the other subprogram. If there are any declarative statements in the moved code, you will probably want to move those up next to the declarative statements in the receiving code. Don't forget to go back to the place where the code came from and delete the SUB/DEF FN statement and the SUBEND/FNEND statements.

SUBEND and FNEND

The SUBEND and FNEND statements must be the last statements in a SUB or function subprogram, respectively. These statements don't ever have to be executed; SUBEXIT and RETURN are sufficient for exiting the subprogram. (If SUBEND is executed, it will behave like a SUBEXIT. If FNEND is executed, it will cause an error.) Rather, SUBEND and FNEND are delimiter statements that indicate to the language system the boundaries between subprograms. The only exception to this rule is the comment statements (either REM or !), which are allowed after SUBEND and FNEND.

Recursion

Both function subprograms and SUB subprograms are allowed to call themselves. This is known as recursion. Recursion is a useful technique in several applications.

The simplest example of recursion is the computation of the factorial function. The factorial of a number N is denoted by N! and is defined to be N × (N−1)! where 0!=1 by definition. Thus N! is simply the product of all the whole numbers from 1 through N inclusive. A recursive function which computes N factorial is:

```
DEF FNFactorial(INTEGER N)
IF N=0 THEN RETURN 1
RETURN N*FNFactorial(N-1)
FNEND
```

Consider also the example of nested multiplication when evaluating a polynomial. A polynomial has the form:

$$A_N X^N + A_{N-1} X^{N-1} + \ldots + A_2 X^2 + A_1 X + A_0$$

One way to evaluate a polynomial is to use the technique of nested multiplication:

$$A_0 + X \times (A_1 + X \times (A_2 + X \times (\ldots\ldots(A_{N-1} + X \times (A_N))\ldots)))$$

If the polynomial is evaluated the way it is written, there are N multiplications, N additions, and N-1 exponentiations performed. Using the nested multiplication technique, there are still N multiplications and N additions, but **no** exponentiations.

The following function implements the nested multiplication recursively:

```
1000  DEF FNPoly_evaluate(A(*),N,X)
1010  ! A(*) is the coefficient array,
1020  !    with N the order of the polynomial.
1030  ! X is the value at which the polynomial
1040  !    is being evaluated.
1050  RETURN FNPoly(A(*),0,N,X)
1060  FNEND

1120  DEF FNPoly(A(*),M,N,X)
1130  ! A(*) is the coefficient array of order N
1140  ! M is the outside coefficient
1150  ! X is the value at which the polynomial
1160  !    is being evaluated.
1170  IF M=N THEN RETURN X*A(N)
1180  RETURN A(M)+X*FNPoly(A(*),M+1,N,X)
1190  FNEND
```

The above examples are cited because they are easily understood, not because they are elegant ways to compute factorials or evaluate polynomials (both are performed much faster in a FOR/NEXT loop). We'll consider a more useful application of recursion in the following section on Top-Down Design.

Top-Down Design

A major problem that every programmer faces is designing programs that can be easily implemented and tested. A lot has been written on this subject over the past 15 years or so, and several references are cited at the end of the chapter. A method of program design that has become widely recommended is Top-Down Design, also known as Stepwise Refinement.

The general approach is to consider a problem at its highest level, and break it down into a small number of identifiable subtasks. Each subtask is in turn considered as a large problem which is to be broken down into smaller problems, and so on until the "smaller problems" which have to be solved turn out to be lines of code, which the computer knows how to solve! At the higher levels of this process, the various subtasks are implemented as subprogram calls. It is best to define exactly what each subprogram is supposed to do long before the subprogram is actually written. Furthermore, this should be done at each level of refinement. By considering what each subprogram requires as input and what it returns as output from the topmost levels, the most serious problems of programming (namely defining your data structures and the communications paths between subprograms) are attacked at the beginning of the problem solving process, rather than at the end when all the small pieces are trying to jumble together. It is best to tackle these questions at the beginning because then you have the most flexibility—no code has been written and it's not necessary to try and save any investment in programming time.

Let's look at a simple example and apply these techniques.

The Problem

In a certain production department in a large manufacturing facility, there are eighty people who build and test widgets. The manager of this department has asked you to write a program to keep track of the total number of widgets each person builds each week. Furthermore, it is necessary to track failure rates during the production process for each person. The manager wants to be able to ask for reports sorted either by employee name, number of units built, or failure rate.

A Data Structure

Before proceeding any further, we need to come up with a data structure which will support the stated requirements.

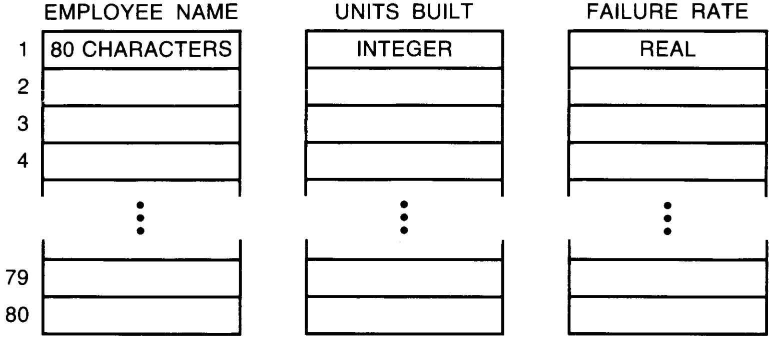


Figure 6-2. Widget Example Data Structure

This above structure is simple and holds all the necessary information. The Jth entry in the Units Built array tells how many units were built by the employee whose name is given by the Jth entry in the Employee Name array, and the Jth entry in the Failure Rate array gives the failure rate that the Jth employee experienced in building the given number of units.

The only problem unsolved by the given data structure is that of ordering. The manager wants to be able to see a report sorted by any one of the three arrays. One way to solve this problem is to provide a sort subprogram as part of the package, but you would have to remember to carry along the other two fields associated with the one on which you are sorting whenever you switch the elements in the array. An alternate way is simply to leave all the data in place and construct a pointer array associated with whichever array you elect to do the report with. A very handy way to construct this pointer array in such a way as to be conducive to printing sorted results is to construct a binary tree.

The binary tree is a simple data structure used for a variety of applications from data management to parsing computer languages. Knuth[4] defines a binary tree as "a finite set of nodes which either is empty, or consists of a root and two disjoint binary trees called the left and right subtrees of the root." Note that this definition is recursive -- it uses the term being defined (binary tree) in its own definition. Thus, a binary tree either consists of two subtrees (which in turn can have two subtrees, etc.), or it is empty. Consider the following illustration of a binary tree:

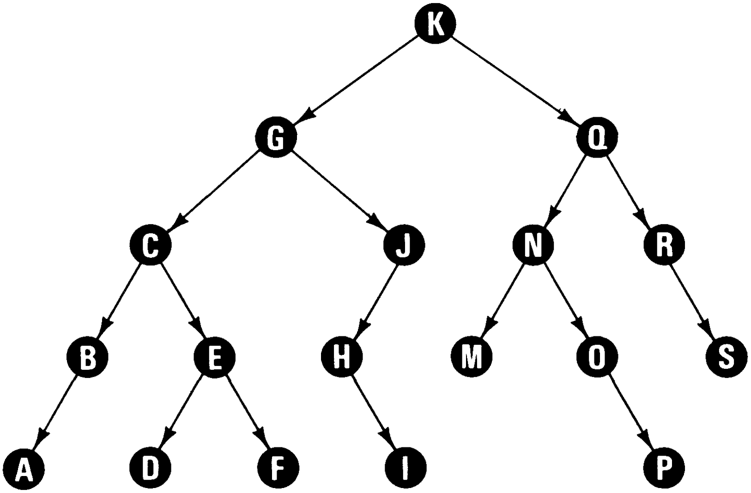


Figure 6-3. Binary Tree

Every node (represented here by a letter) has at most two subtrees. The subtrees are ordered on a lexical basis. Every letter belonging to any node's left subtree will be lexically "less" than the node itself, which in turn will be lexically "less" than the letters in that node's right subtree. Because the binary tree is defined recursively, this relationship will hold true at all levels of the tree. Furthermore, there are extremely simple recursive algorithms for traversing (or in our case printing) a binary tree which is organized lexically in sorted order.

Graphically, the tree is easy to understand. You have a piece of data (or a "node") and you have a couple of little arrows which point to the next nodes. Inside a computer, these little arrows are called "pointers" because they point to a location in memory where the next node is to be found.

Our binary tree is going to be implemented as an 80 by 2 integer array. Any element of this array A(I,1) will be a pointer to the left subtree, while A(I,2) will be a pointer to the right subtree. I is simply the location within the other three arrays of the pertinent data. The first item in each array is defined to be the root of the tree.

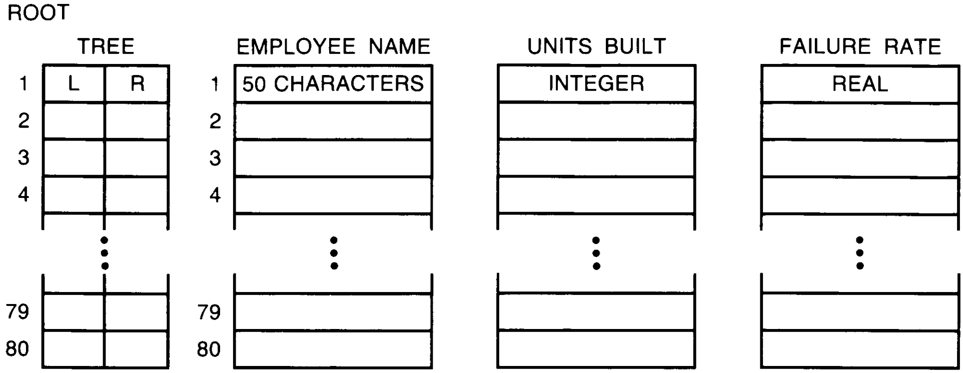


Figure 6-4. Revised Data Structure

The manager can choose which field to sort on when the report is printed, and the `Pointers(*)` array will be constructed accordingly.

The amount of detail we've spent studying and understanding the data structure emphasizes the importance of this phase of the design.

Let's proceed now with designing our program. At the highest level, what we would really like to have is a command which does everything at one fell swoop:

```
10   Do_it
20   END
```

Top-Down design calls for breaking this massive task down into a set of smaller problems. First we'll declare the data structure and define what actions we want to take on the data. Note that in an actual application, there would be some sort of menu to let the user choose the action he desired. The human interface has been left off this example for the sake of simplicity.

```
10   OPTION BASE 1
20   DIM Name$(80)[80],Failure_rate(80)
30   INTEGER Units_built(80),Max,Howmany
40   Max=80
50   Input_data(Name$(*),Units_built(*),Failure_rate(*),Max,Howmany)
60   Store_data(Name$(*),Units_built(*),Failure_rate(*),Max,Howmany)
70   Report(Name$(*),Units_built(*),Failure_rate(*),Max,Howmany)
80   END
90   SUB Input_data(Name$(*),INTEGER Units(*),REAL Failures(*),INTEGER
Max,Howmany)
100  SUBEND
110  SUB Store_data(Name$(*),INTEGER Units(*),REAL Failures(*),INTEGER
Max,Howmany)
120  SUBEND
130  SUB Report(Name$(*),INTEGER Units(*),REAL Failures(*),INTEGER Max,
Howmany)
140  SUBEND
```

Notice that we haven't worried about the tree structure yet. This is because the tree is only used to provide ordering information to be used in printing out the report. Since the tree is not necessary except for the report, we'll let the report subprogram worry about it. The variables `Max` and `Howmany` are introduced for the sake of flexibility. It is possible that the department may have to hire more people at some time in the future, or that some people may leave the company or accept jobs in other departments. In this case, the program will have to be changed to allow for a different number of people. By making the maximum number of people, and the actual number of people, variables instead of constants, modifying the program becomes very easy.

Also, notice that each of the subprograms has been "stubbed in." The reason for doing this is that you can immediately run the program to test the communications between modules. So far, the program will not do anything, but it does allow you to make sure that your pass parameter lists match the formal parameter lists in the number and types of parameters. Furthermore, this process can be repeated every step of the way. As each subprogram is designed, the modules called by it can be "stubbed in" in a similar fashion, insuring that the parameter lists and communications paths are well defined and properly implemented at every level of your design. The most difficult part of testing your program is done as the program is being designed.

Let's step down to the next level of the design and consider each of the subprograms mentioned above:

```
90   SUB Input_data(Name$(*),INTEGER Units(*),REAL Failures(*),INTEGER
Max,Howmany)
91   DIM Which$[3]
92   INPUT "New Data or Old?",Which$
93   IF Which$="New" THEN
94      Enter_new(Name$(*),Units(*),Failures(*),Max,Howmany)
95   ELSE
96      Edit_old(Name$(*),Units(*),Failures(*),Max Howmany)
97   END IF
100  SUBEND
101  !
110  SUB Store_data(Name$(*),INTEGER Units(*),REAL Failures(*),INTEGER
Max,Howmany)
111  Setup_file(@File)
112  OUTPUT @File;Name$(*),Units(*),Failures(*)
113  ASSIGN @File TO *
120  SUBEND
121  !
130  SUB Report(Name$(*),INTEGER Units(*),REAL Failures(*),INTEGER
Max,Howmany)
132  OPTION BASE 1
133  INTEGER Root,I,Whichfield
134  ALLOCATE INTEGER Tree(Howmany,2)
135  Init_tree(Root,Tree(*))
136  Ask: INPUT "Which field (1=Name,2=Units,3=Failures)?",Whichfield
137  IF Whichfield<1 OR Whichfield>3 THEN Ask
138  FOR I=2 TO Howmany
139     SELECT Which_field
140     CASE 1
141        Buildstring(Root,Tree(*),I,Name$(*))
142     CASE 2
143        Buildnum(Root,Tree(*),I,Units(*))
144     CASE 3
145        Buildnum(Root,Tree(*),I,Failures(*))
146     END SELECT
148  NEXT I
149  Inorder(Root,Tree(*),Name$(*),Units(*),Failures(*))
150  SUBEND
```

Here we haven't gone through the exercise of providing the dummy subprograms, though in actual practice we would. In lines 94 and 96 of the data entry program, we see two more subprograms that need to be designed. The module for entering new data from the keyboard will be straightforward and need not be considered in further detail for this example. The module for editing old data will involve loading a set of data from the diskette and then allowing the user to modify those values. This will involve a little more detail and perhaps another level of subprograms, but the techniques to be used are still straightforward enough not to demand further attention here.

Line 111 calls for a module to setup a data file to store the data on, and passes an I/O path name back out that's ready for use. This means that the subprogram must:

1. Ask the user for a file
2. Create the file if necessary
3. ASSIGN it for use

The Report subprogram is by far the most interesting one in this example, since it deals with the initialization, construction, and traversal of a binary tree, as discussed above. The **Init_tree** subprogram called in line 135 simply initializes the root node's (first element, remember) subtrees to be empty. Subsequently, the **Buildstring** subprogram called in line 140 simply enters the Ith string in the **Name$(*)** array into the structure of **Tree(*)**, assuming that the user asked for the report to be sorted by **Name$(*)**. Similarly, if the user wanted either **Units(*)** or **Failures(*)** to be the sort key, then the **Buildnum** subprogram (called in lines 143 and 145) would be used to construct the tree.

Finally, the **Inorder** subprogram traverses the structure in "inorder" once the tree has been built. "Inorder" simply means that every node is printed in between that node's subtrees. This traversal mechanism, as you will see, is quite short, and is a truly elegant expression of the task being performed.

Here are the Init_tree, Buildstring, and Inorder subprograms (Buildnum is so similar to Buildstring that it isn't necessary to list it too):

```
200  SUB Init_tree(INTEGER Root,Tree(*))
210  COM /Tree/INTEGER Nil,Left,Right
220  Nil=0
230  Left=1
240  Right=2
250  Root=1
260  Tree(Root,Left)=Nil
270  Tree(Root,Right)=Nil
280  SUBEND
281  !
290  SUB Buildstring(INTEGER Root,Tree(*),Index,A$(*))
300  COM /Tree/INTEGER Nil,Left,Right
310  IF A$(Index)<=A$(Root) THEN     ! Search the left subtree
320     IF Tree(Root,Left)=Nil THEN  ! Once a leaf is found (link is
330        Tree(Root,Left)=Index     !  nil) point to the new node
340        Tree(Index,Left)=Nil      !  (Index) with the leaf's left
350        Tree(Index,Right)=Nil     !  pointer and set up the new
351                                  !  node as a leaf.
360     ELSE
370        Buildstring(A(Root,Left),Tree(*),Index,A$(*))
380     END IF
390  ELSE                            ! Search the right subtree
400     IF Tree(Root,Right)=Nil THEN ! Once a leaf is found (link is
410        Tree(Root,Right)=Index    !  nil) point to the new node
420        Tree(Index,Left)=Nil      !  from the right pointer instead
430        Tree(Index,Right)=Nil     !  of the left.
440     ELSE
450        Buildstring(A(Root,Right),Tree(*),Index,A$(*))
460     END IF
470  END IF
480  SUBEND
481  !
490  SUB Inorder(INTEGER Root,Tree(*),Name$(*),INTEGER Units(*),REAL
Failures(*))
500  COM /Tree/INTEGER Nil,Left,Right
510  IF Root<>Nil THEN
520     Inorder(A(Root,Left),Tree(*),Name$(*),Units(*),Failures(*))
530     PRINT Name$(Root),Units(Root),Failures(Root)
540     Inorder(A(Root,Right),Tree(*),Name$(*),Units(*),Failures(*))
550  END IF
560  SUBEND
```

Let's step through a sample input stream and see how the tree is constructed using the `Buildstring` subprogram:

Contents of the `Name$(*)` array:

1. Perriwinkle
2. Jones
3. Smith
4. Snodgrass
5. Figby
6. Brown
7. Thompson
8. Richards
9. Hughes
10. Davenport

Tree structure after `Init_tree` is executed:

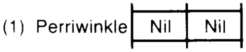


Figure 6-5. Tree Structure After Init_tree Execution

Tree structure after subsequent insertions into the tree by the **Buildstring** subprogram:

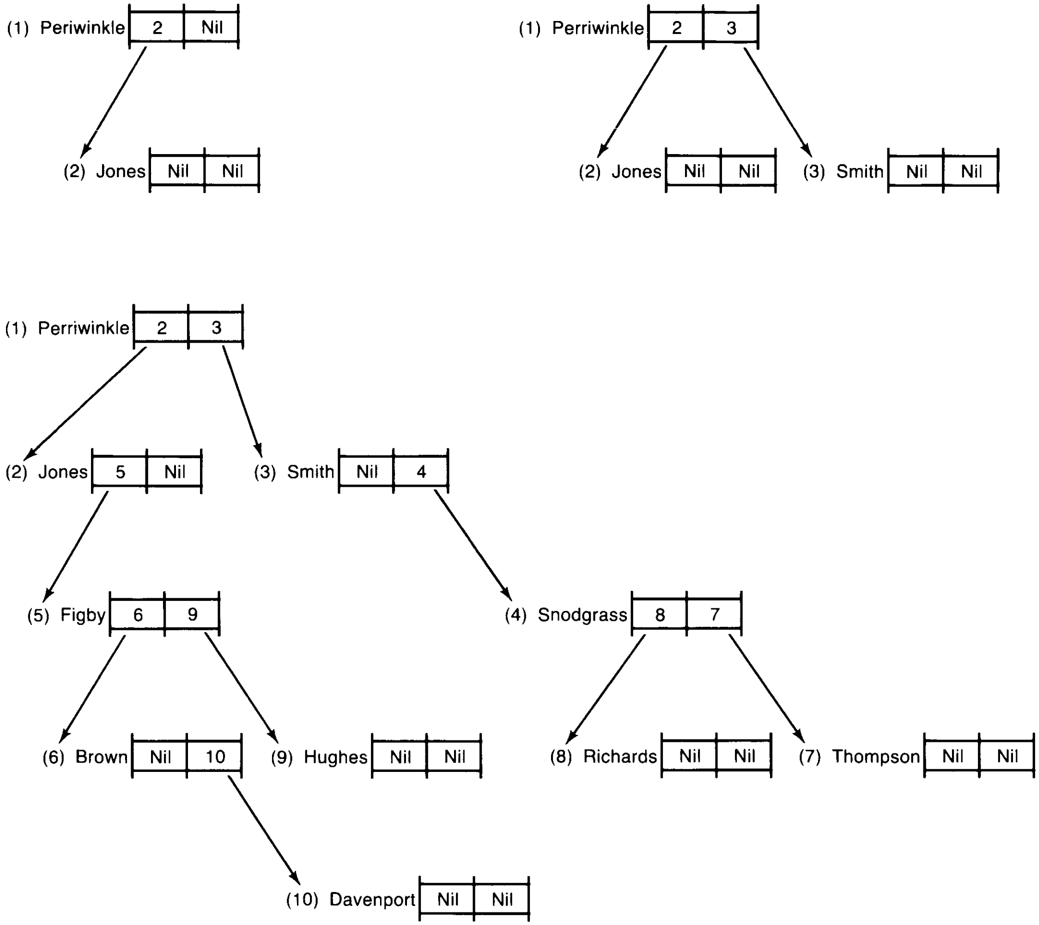


Figure 6-6. Tree Structure After Buildstring Execution

These three subprograms illustrate several points that were discussed in this chapter:

- They share a labelled COM block which is not declared in the main program, nor in the Report program. The information in the COM block was only relevant to the the three subprograms, yet the programs never called each other—they were all called from Report.
- Both the `Inorder` and `Buildstring` subprograms are recursive—they call themselves. This technique was an appealing way to solve the problem because of the recursive nature of the data structure. (Many types of data structures are recursively defined.)
- The use of subprograms to build and traverse the data structure turned out to execute faster than a sort subprogram which physically moved the items in the three fields into a given order based on sorting one of the arrays. (The difference was about 40% using Donald Shell's algorithm[5].)
- The method of Top-Down Design led to the orderly design, creation, and testing of each subprogram, module by module, layer by layer. Communication paths and data structures/types were forced to be clearly defined at each step of the way.

[1] Wirth, Niklaus, "Program Development by Stepwise Refinement", *Communications of the ACM*, April 1971, Vol. 14, No. 4, pp. 221-227

[2] Yourdan, Edward, *Techniques of Program Structure and Design*, (Prentice-Hall, Englewood Cliffs, NJ, 1975)

[3] Dahl, Dijkstra, & Hoare, *Structured Programming* (Academic Press, New York, 1972)

[4] Knuth, Donald E., *The Art of Computer Programming*, Vol. 1, Fundamental Algorithms (Addison-Wesley, Reading, Mass, 1973), pp. 308-309,316-317

[5] Knuth, Donald E., *The Art of Computer Programming*, Vol. 3, Sorting and Searching (Addison-Wesley, Reading, Mass, 1973), pp. 84-85

Data Storage and Retrieval 7

Data Storage and Retrieval 7

This chapter describes some useful techniques for storing and retrieving data.

- First we describe how to store and retrieve **data that is part of the BASIC program**. With this method, **DATA statements** specify data to be stored in the memory area used by BASIC programs; thus, the data is always kept with the program, even when the program is stored in a mass storage file. The data items can be retrieved by using READ statements to assign the values to variables. This is a particularly effective technique for small amounts of data that you want to maintain in a program file.
- For larger amounts of data, and for data that will be generated or modified by a program, **mass storage files** are more appropriate. Files provide means of storing data on mass storage devices. The three types of data files available with Series 200/300 BASIC system computers are described in this chapter.
  - ASCII—used for general text and numeric data storage. (These are the interchange method with many other HP systems.)
  - BDAT—provide the most compact and flexible data storage mechanism.
  - HP-UX—similar to BDAT files in format and flexibility, but can also be interchanged with other systems like ASCII files.

More details about these files, including how to choose a file type and how to access each, are described in this chapter.

Storing Data in Programs

This section describes a number of ways to store values in memory. In general, these techniques involve using program variables to store data. The data are kept with the program when it is stored on a mass storage device (with STORE and SAVE). These techniques allow extremely fast access of the data. They provide good use of the computer's memory for storing relatively small amounts of data.

Storing Data in Variables

Probably the simplest method of storing data is to use a simple assignment, such as the following LET statements:

```
100   LET Cm_per_inch=2.54
110   Inch_per_cm=1/Cm_per_inch
```

The data stored in each variable can then be retrieved simply by specifying the variable's name. This technique works well when there are only a relatively few items to be stored or when several data values are to be computed from the value of a few items. The program will execute faster when variables are used than when expressions containing constants are used; for instance, using the variable `Inch_per_cm` in the preceding example would be faster than using the constant expression `1/2.54`. In addition, it is easier to modify the value of an item when it appears in only one place (i.e., in the LET statement).

Data Input by the User

You also can assign values to variables at run-time with the INPUT and LINPUT statements as shown in the following examples.

```
100   INPUT "Type in the value of X, please.",Id
  .
  .
  .
200   DISP "Enter the value of X, Y, and Z.";
210   LINPUT "",Response$
```

Note that with this type of storage, the values assigned to the corresponding variables are *not* kept with the program when it is stored; they must be entered each time the program is run. This type of data storage can be used when the data are to be checked or modified by the user each time the program is run. As with the preceding example, the data stored in each variable can then be retrieved simply by specifying the variable's name.

Using DATA and READ Statements

The DATA and READ statements provide another technique for storing and retrieving data from the computer's read/write (R/W) memory. The DATA statement allows you to store a stream of data items in memory, and the READ statement allows you retrieve data items from the stream.

You can have any number of READ and DATA statements in a program in any order you want. When you RUN a program, the system concatenates all DATA statements in the same context into a single "data stream." Each subprogram has its own data stream. The following DATA statements distributed in a program would produce the following data stream.

```
100 DATA 1,A,50
      .
      .
      .
200 DATA "BB",20,45
      .
      .
      .
300 DATA X,Y,77
```

DATA STREAM:

1	A	50	BB	20	45	X	Y	77

As you can see from the example above, a data stream can contain both numeric and string data items; however, each item is stored as if it were a string.

Each data item must be separated by a comma and can be enclosed in optional quotes. Strings that contain a comma, exclamation mark, or quote mark must be enclosed in quotes. In addition, you must enter two quote marks for every one you want in the string. For example, to enter the string QUOTE"QUO"TE into a data stream, you would write:

```
100 DATA "QUOTE""QUO""TE"
```

To retrieve a data item, assign it to a variable with the READ statement. Syntactically, READ is analogous to DATA; but instead of a data list, you use a variable list. For instance, the statement:

```
100 READ X,Y,Z$
```

would read three data items from the data stream into the three variables. Note that the first two items are numeric and the third is a string variable.

Numeric data items can be READ into either numeric or string variables. If the numeric data item is of a different type than the numeric variable, the item is converted (i.e., REALs are converted to INTEGERs, and INTEGERs to REALs). If the conversion cannot be made, an error is returned. A READ into a COMPLEX variable is satisfied with two REAL DATA values. Strings that contain non-numeric characters must be READ into string variables. If the string variable has not been dimensioned to a size large enough to hold the entire data item, the data item is truncated.

The system keeps track of which data item to READ next by using a "data pointer." Every data stream has its own data pointer which points to the next data item to be assigned to the next variable in a READ statement. When you run a program segment, the data pointer is placed initially at the first item of the data stream. Every time you READ an item from the stream, the pointer is moved to the next data item. If a subprogram is called by a context, the position of the data pointer is recorded and then restored when you return to the calling context.

Starting from the position of the data pointer, data items are assigned to variables one by one until all variables in a READ statement have been given values. The exception is when a COMPLEX variable is read   two numeric data items are consumed. If there are more variables than data items, the system returns an error, and the data pointer is moved back to the position it occupied before the READ statement was executed. .index READ statement

Examples

The following example shows how data is stored in a data stream and then retrieved. Note that DATA statements can come after READ statements even though they contain the data being READ. This is because DATA statements are linked during program pre-run, whereas READ statements aren't executed until the program actually runs.

```
10  DATA November,26
20  READ Month$,Day,Year$
30  DATA 1981,"The date is"
40  READ Str$
50  Print Str$;Month$,Day,Year$
60  END
```

```
  The date is November 26 1981
```

Storage and Retrieval of Arrays

In addition to using READ to assign values to string and numeric variables, you can also READ data into arrays. The system will match data items with variables one at a time until it has filled a row. The next data item then becomes the first element in the next row. You must have enough data items to fill the array or you will get an error. In the examples below, we show how REAL and COMPLEX DATA values can be assigned to elements of a 3-by-3 numeric array. Note that two COMPLEX DATA values have to be assigned to each element of the 3-by-3 numeric array. The first COMPLEX DATA value is the real part of the complex number and the second COMPLEX DATA value is the imaginary part.

```
10    OPTION BASE 1
20    DIM Example1(3,3)
30    DATA 1,2,3,4,5,6,7,8,9,10,11
40    READ Example1(*)
50    PRINT USING "3(K,X),/";Example1(*)
60    END

 1 2 3
 4 5 6
 7 8 9
```

```
10    OPTION BASE 1
20    COMPLEX Example2(3,3)
30    DATA 23,-2,-1,10,-6,-7,4,5,-8,10,1,1,34,2,9,17,-12,-14
40    READ Example2(*)
50    PRINT USING "3(3D,X,3D,3X),/";Example2(*)
60    END

23  -2     -1  10     -6  -7
 4   5     -8  10      1   1
34   2      9  17    -12 -14
```

In the first example, the data pointer is left at item 10; thus, items 10 and 11 are saved for the next READ statement. In the second example, there are just enough items to fill each element of the complex array.

Moving the Data Pointer

In some programs, you will want to assign the same data items to different variables. To do this, you have to move the data pointer so that it is pointing at the desired data item. You can accomplish this with the RESTORE statement. If you don't specify a line number or label, RESTORE returns the data pointer to the first data item in the data stream. If you do include a line identifier in the RESTORE statement, the data pointer is moved to the first data item in the first DATA statement at or after the identified line. The example below illustrates how to use the RESTORE statement.

```
100   DIM Array1(1:3)    ! Dimensions a 3-element array.
110   DIM Array2(0:4)    ! Dimensions a 5-element array.
120   DATA 1,2,3,4       ! Places 4 items in stream.
130   DATA 5,6,7         ! Places 3 items in stream.
140   READ A,B,C         ! Reads first 3 items in stream.
150   READ Array2(*)     ! Reads next 5 items in stream.
160   DATA 8,9           ! Places 2 items in stream.
170                      !
180   RESTORE            ! Re-positions pointer to 1st item.
190   READ Array1(*)     ! Reads first 3 items in stream.
200   RESTORE 140        ! Moves data pointer to item "8".
210   READ D             ! Reads "8".
220                      !
230   PRINT "Array1 contains:";Array1(*);" "
240   PRINT "Array2 contains:";Array2(*);" "
250   PRINT "A,B,C,D equal:";A;B;C;D
260   END
```

```
Array1 contains: 1 2 3
Array2 contains: 4 5 6 7 8
A,B,C,D equal: 1 2 3 8
```

File Input and Output (I/O)

The rest of this chapter describes the second general class of data storage and retrieval—that of using mass storage files. It presents BASIC programming techniques used for accessing files.

- The first section gives a brief introduction to the *general* steps you might take to:
    - Choose a file type.
    - Store data in any file.
- Subsequent sections describe *details* of these steps with ASCII, BDAT, and HP-UX files.

If you feel that you need additional background information about files or mass storage organization while reading this material, refer to the "Mass Storage Concepts" chapter of *Installing and Maintaining the BASIC System.*

Brief Comparison of Available File Types

With the Series 200/300 BASIC system, there are three different types of files in which you can store and retrieve data. Understanding the characteristics of each file type will help you choose the one best suited for your specific application.

- **ASCII**—used for general text and numeric data storage. Here are the **advantages** of this type of file:
    - There is less chance of reading the contents into the wrong data type (which is possible with BDAT and HP-UX files). Thus, it is the easiest file to read when you don't know how it was written.
    - The file format provides fairly compact storage for string data.
    - ASCII files are compatible with other HP computers that support this file type[1].
    - ASCII files containing BASIC program lines can be read with GET and written with SAVE.

[1] The full name of ASCII files is "LIF ASCII." LIF stands for Logical Interchange Format, a directory and data storage format that is used by many HP computer divisions.

The main **disadvantages** of ASCII files are that:

  - They can be accessed *serially* but not *randomly.*
  - They can be written in *only default ASCII format* (no formatting is possible[1], and the data cannot be stored in internal representation).

- **BDAT**—provide the most compact and flexible data storage mechanism. These files have several **advantages**:

  - They can be *randomly or serially* accessed.
  - More *flexibility* in data formats and access methods.
  - *Faster* transfer rates.
  - Generally more *space-efficient* than ASCII files (except for string data items).
  - They allow data to be stored in ASCII format, internal format, or in a "custom" format (which you can define with IMAGE specifiers).

  The **disadvantages** are that:

  - You *must* know how the data items were written (as INTEGERs, REALs, COMPLEX values, strings, etc.) in order to correctly read the data back.
  - These data files cannot be *interchanged* with as many other systems as can ASCII files (for instance, the Series 200/300 Pascal Workstation system cannot read BDAT files).

[1] It is possible, however, to format data to be sent to an ASCII file by first sending it to a string variable (with OUTPUT..USING), and then OUTPUT this string's contents to the file. See the subsequent section called "Formatted OUTPUT with ASCII Files" for examples.

- **HP-UX**—similar to BDAT files in structure, but also have some of the advantages of ASCII files:
  - Like BDAT files, they can also be accessed randomly or serially, and they can use ASCII, internal, or custom data representations.
  - Like ASCII files, they are useful for data-file interchange; however, the set of computers with which they can be interchanged is slightly different than LIF ASCII files. HP-UX files can be interchanged with any other system that uses the Hierarchical File System (HFS) format for mass storage volumes (such as HP-UX systems, and HP Series 200/300 Pascal systems beginning with version 3.2). See the "Porting and Sharing Files" chapter for a list of operating systems and languages that support HP-UX file access.
  - HP-UX files containing BASIC program lines can be read with GET and written with RE-SAVE.

If in doubt about the type of file to use, choose a BDAT file because of its speed and compact data storage.

Overview of File I/O

Storing data in files requires a few simple steps. The following program segment shows a simplistic example of placing several items in a data file.

```
100  REAL Real_array1(1:50,1:25),Real_array2(1:50,1:25)
110  INTEGER Integer_var
120  DIM String$[100]
       .
       .
390  ! Specify default mass storage.
400  MASS STORAGE IS ":,700,1"
410  !
420  ! Create BDAT data file with ten (256-byte) records
430  ! on the specified mass storage device (:,700,1).
440  CREATE BDAT "File_1",10
450  !
460  ! Assign (open) an I/O path name to the file.
470  ASSIGN @Path_1 TO "File_1"
480  !
490  ! Write various data items into the file.
500  OUTPUT @Path_1;"Literal"        ! String literal.
510  OUTPUT @Path_1;Real_array1(*)   ! REAL array.
520  OUTPUT @Path_1;255              ! Single INTEGER.
530  !
540  ! Close the I/O path.
550  ASSIGN @Path_1 TO *
       .
```

```
790  ! Open another I/O path to the file (assume same default drive).
800  ASSIGN @F_1 TO "File_1"
810  !
820  ! Read data into another array (same size and type).
830  ENTER @F_1;String_var$         ! Must be same data types
840  ENTER @F_1;Real_array2(*)      ! used to write the file.
850  ENTER @F_1;Integer_var         ! "Read it like you wrote it."
860  !
870  ! Close I/O path.
880  ASSIGN @F_1 TO *
```

Line 400 specifies the *default mass storage device,* which is to be used whenever a mass storage device is *not explicitly specified* during subsequent mass storage operations. The term *mass storage volume specifier (msvs)* describes the string expression used to uniquely identify which device is to be the mass storage. In this case, ":,700,1" is the msvs. (For a complete discussion of mass storage volume specifiers, see the "Mass Storage Concepts" chapter of *Installing and Maintaining the BASIC System* manual.)

In order to store data in mass storage, a data file must be created (or already exist) on the mass storage media. In this case, line 440 creates a BDAT file[1]; the file created contains 10 defined records of 256 bytes each. (Defined records and record size are discussed later in this chapter.)

The term *file specifier* describes the string expression used to uniquely identify the file. In this example, the file specifier is simply `File_1`, which is the file's name. If the file is to be created (or already exists) on a mass storage device *other than the default mass storage*, the appropriate msus must be appended to the file name. If that device has a hierarchical directory format (such as HFS or SRM discs), then you may also have to specify a directory path (such as /USERS/MARK/PROJECT_1).

Then, in order to store data in (or retrieve data from) the file, you must assign an I/O path name to the file. Line 470 shows an example of assigning an I/O path name to the file (also called opening an I/O path to the file). Lines 500 through 520 show data items of various types being written into the file through the I/O path name.

The I/O path name is closed after all data have been sent to the file. In this instance, closing the I/O path may have been optional, because a *different* I/O path name is assigned to the file later in the program. (All I/O path names are automatically closed by the system at the end of the program.) Closing an I/O path to a file updates the file pointers.

[1] Later sections describe using HP-UX and ASCII files.

Since these data items are to be retrieved from the file, another ASSIGN statement is executed to open the file (line 800). Notice that a different I/O path name was arbitrarily chosen. Opening this I/O path name to the file sets the file pointer to the beginning of the file. (Re-opening the I/O path name `@File_1` would have also reset the file pointer.)

Notice also that the msvs is *not* included with the file name. This shows that the current default mass storage device, here ":,700,1", is assumed when a mass storage device is not specified.

The subsequent ENTER statements read the data items into variables; *with BDAT and HP-UX files*[1], the *data type of each variable must match the data type type of each data item.* With ASCII files, for instance, you can read INTEGER items into REAL variables and not have problems.

This is a fairly simple example; however, it shows the general steps you must take to access files.

A Closer Look at General File Access

Before you can access a data file, you must assign an I/O path name to the file. Assigning an I/O path name to the file sets up a table in computer memory that contains various information describing the file, such as its type, which mass storage device it is stored on, and its location on the media. The I/O path name is then used in I/O statements (OUTPUT, ENTER, and TRANSFER) which move the data to and from the file. I/O path names are also used to transfer data to and from devices. *BASIC Interfacing Techniques* explains data transfers with devices and provides several relevant insights into data representations. However, in this chapter we deal mostly with I/O paths to files.

[1] When using the BASIC internal (FORMAT OFF) data representation. This topic is discussed in the section called "A Closer Look at BDAT and HP-UX Files".

Every I/O path to a file maintains the following information:

Validity Flag	Tells whether the path is currently opened (assigned) or closed (not assigned).
Type of Resource	Holds the file type: ASCII, BDAT, or HP-UX.
Device Selector	Stores the device selector of the drive. (I/O paths can also be associated with devices and buffers. See *BASIC Interfacing Techniques* for further details.)
Attributes	Such as FORMAT OFF and FORMAT ON, BYTE, and PARITY ODD.
File Pointer	There is a file pointer that points to the place in the file where the next data item will be read or written. The file pointer is updated whenever the file is accessed.
End-Of-File Pointer	An I/O path has an EOF pointer that points to the byte that follows the last byte of the file.

Opening an I/O Path

I/O path names are similar to other variable names, except that I/O path names are preceded by the "@" character. When an I/O path name is used in a statement, the system looks up the contents of the I/O path name and uses them as required by the situation.

To open an I/O path to a file (to set the validity flag to Open), assign the I/O path name to a file specifier by using an ASSIGN statement. For example, executing the following statement:

```
ASSIGN @Path1 TO "Example"
```

assigns an I/O path name called `@Path1` to the file `Example`. The file that you open must already exist and must be a data file. If the file does not satisfy one of these requirements, the system will return an error. If you do not use an msus in the file specifier, the system will look for the file on the current MASS STORAGE IS device. If you want to access a different device, use the msus syntax described earlier. For instance, the statement:

```
ASSIGN @Path2 TO "Example:HP9122,700"
```

opens an I/O path to the file `Example` on an HP 9122 disc drive, interface select code 7 and primary address 0. You must include the protect code or password, if the LIF or SRM file has one, respectively.

ASSIGNing an I/O path name to a file has the following effect on the I/O path table:

- If the I/O path is currently open, the system *closes* the I/O path and then *re-opens* it. If the I/O path is not currently open, it is opened. In both cases, the system sets the validity flag to Open.
- The file's type (ASCII, BDAT, or HP-UX) is set.
- The file's directory path (if in a hierarchical directory structure) and msus are recorded.
- The specified attributes are assigned to the I/O path name. If an attribute is not specified, the appropriate default attribute is assigned (such as FORMAT OFF with BDAT and HP-UX files, and FORMAT ON with ASCII files).
- The file pointer is positioned to the beginning of the file.
- If the I/O path name is associated with a BDAT or HP-UX file, the physical EOF pointer (read from the volume on which the file resides) is copied to the I/O path table.

Once an I/O path has been opened to a file, you always use the path name to access the file. An I/O path name is only valid in the context in which it is opened, unless you pass it as a parameter or put it in the COM area. To place a path name in the COM area, simply specify the path name in a COM statement before you ASSIGN it. For instance the two statements below would declare an I/O path name in an unnamed COM area and then open it:

```
100   COM @Path3
110   ASSIGN @Path3 TO "File1"
```

Assigning Attributes

When you open an I/O path, certain attributes are assigned to it which define the way data is to be read and written. There are two attributes which control how data items are represented: FORMAT ON and FORMAT OFF.

- With FORMAT ON, ASCII data representations are used.
- With FORMAT OFF, the BASIC system's internal data representations are used.

Additional attributes are available, which provide control of such functions as parity generation and checking, converting characters, and changing end-of-line (EOL) sequences. See ASSIGN in the *BASIC Language Reference*, or "I/O Path Attributes" in the *BASIC Interfacing Techniques* for further details.

As mentioned in the tutorial section, BDAT files can use either data representation; however, ASCII files permit only ASCII-data format. Therefore, if you specify FORMAT OFF for an I/O path to an ASCII file, the system ignores it. The following ASSIGN statement specifies a FORMAT attribute:

```
ASSIGN @Path1 TO "File1";FORMAT OFF
```

If `File1` is a BDAT or HP-UX file, the FORMAT OFF attribute specifies that the internal data formats are to be used when sending and receiving data through the I/O path. If the file is of type ASCII, the attribute will be ignored. *Note that FORMAT OFF is the default FORMAT attribute for BDAT and HP-UX files.*

Executing the following statement directs the system to use the ASCII data representation when sending and receiving data through the I/O path:

```
ASSIGN @Path2 TO "File2";FORMAT ON
```

If `File2` is a BDAT or HP-UX file, data will be written using ASCII format, and data read from it will be interpreted as being in ASCII format. For an ASCII file, this attribute is redundant since ASCII-data format is the only data representation allowed anyway.

If you want to change the attribute of an I/O path, you can do so by specifying the I/O path name and attribute in an ASSIGN statement while excluding the file specifier. For instance, if you wanted to change the attribute of `@Path2` to FORMAT OFF, you could execute:

```
ASSIGN @Path2;FORMAT OFF
```

Alternatively, you could re-enter the entire statement:

```
ASSIGN @Path2 TO "File2";FORMAT OFF
```

These two statements, however, are not identical. The first one only changes the FORMAT attribute. The second statement resets the entire I/O path table (e.g., resets the file pointer to the beginning of the file).

It is important to note that once a file is written, changing the FORMAT attribute of an I/O path to the file should only be attempted by experienced programmers. **In general, data should always be read in the same manner as it was written.** For instance, data written to a BDAT or HP-UX file with FORMAT OFF should also be read with FORMAT OFF, and vice versa. In addition, the same data types should be used to write the file as to read the file. For instance, if data items were written as INTEGERs, they should also be read as INTEGERs (this is mandatory with FORMAT OFF, but not always necessary with FORMAT ON).

In theory, there is no limit to the number of I/O paths you can ASSIGN to the same file. Each I/O path, however, has its own file pointer and EOF pointer, so that in practice it can become exceedingly difficult to keep track of where you are in a file if you use more than one I/O path. **We recommend that you use only one I/O path at any one time for each file.**

Closing I/O Paths

I/O path names not in the COM area are closed whenever the system moves into a stopped state (e.g., STOP, END, SCRATCH, EDIT, etc.). I/O path names local to a context are closed when control is returned to the calling context. Re-ASSIGNing an I/O path name will also cancel its previous association.

You can also explicitly cancel an I/O path by ASSIGNing the path name to an * (asterisk). For instance, the statement:

```
ASSIGN @Path2 TO *
```

closes `@Path2` (sets the validity flag to Closed). `@Path2` cannot be used again until it is re-assigned. You can re-assign a path name to the same file or to a different file.

A Closer Look at Using ASCII Files

You have already been introduced to general file I/O techniques in the example of writing and reading a BDAT file in the preceding section. This section gives you a closer look at ASCII file I/O techniques.

Example of ASCII File I/O

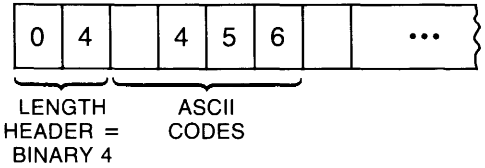
Storing data in ASCII files requires a few simple steps. The following program segment shows a simplistic example of placing several items in an ASCII data file. Note that it is *nearly identical* to the first example in the preceding "Overview of File I/O" section, except for changes to the CREATE statement and file name.

```
100  REAL Real_array1(1:50,1:25),Real_array2(1:50,1:25)
110  INTEGER Integer_var
120  DIM String$[100]
       .
       .
390  ! Specify "default" mass storage device.
400  MASS STORAGE IS ":,700,1"
410  !
420  ! Create ASCII data file with 10 sectors
430  ! on the "default" mass storage device.
440  CREATE ASCII "File_2",10
450  !
460  ! Assign (open) an I/O path name to the file.
470  ASSIGN @Path_1 TO "File_2"
480  !
490  ! Write various data items into the file.
500  OUTPUT @Path_1;"Literal"       ! String literal.
510  OUTPUT @Path_1;Real_array1(*)  ! REAL array.
520  OUTPUT @Path_1;255             ! Single INTEGER.
530  !
540  ! Close the I/O path.
550  ASSIGN @Path_1 TO *
       .
       .
790  ! Open another I/O path to the file (assume same default drive).
800  ASSIGN @F_1 TO "File_2"
810  !
820  ! Read data into another array (same size and type).
830  ENTER @F_1;String_var          ! Must be same data types.
840  ENTER @F_1;Real_array2(*)
850  ENTER @F_1;Integer_var
860  !
```

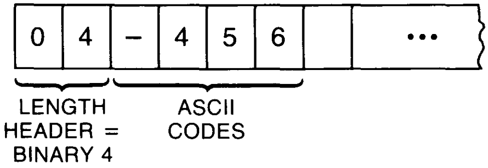
```
870  ! Close I/O path.
880  ASSIGN @F_1 TO *
          .
          .
```

Data Representations in ASCII Files

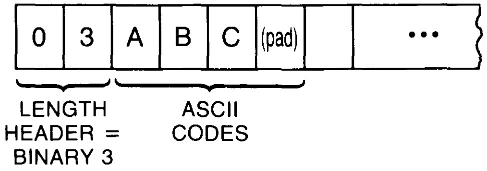
In an ASCII file, every data item, whether string or numeric, is represented by ASCII characters; one byte represents one ASCII character. Each data item is preceded by a two-byte length header which indicates how many ASCII characters are in the item. However, there is no "type" field for each item; data items contain no indication (in the file) as to whether the item was stored as string or numeric data. For instance, the number 456 would be stored as follows in an ASCII file:



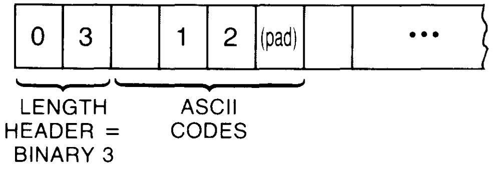
Note that there is a space at the beginning of the data item. This signifies that the number is positive. If a number is negative, a minus sign precedes the number. For instance, the number −456, would be stored as follows:



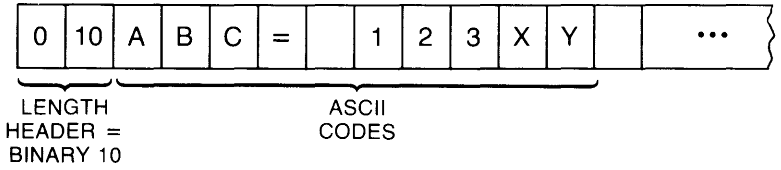
If the length of the data item is an odd number, the system "pads" the item with a space to make it come out even. The string "ABC", for example, would be stored as follows:



There is often a relatively large amount of overhead for numeric data items. For instance, to store the integer 12 in an ASCII file requires the following six bytes:



Similarly, reading numeric data from an ASCII file can be a complex and relatively slow operation. The numeric characters in an item must be entered and evaluated individually by the system's "number builder" routine, which derives the number's internal representation. (Keep in mind that this routine is called automatically when data are entered into a numeric variable.) For example, suppose that the following item is stored in an ASCII file:



Although it may seem obvious that this is not a numeric data item, the system has no way of knowing this since *there is no type-field stored with the item.* Therefore, if you attempt to enter this item into a numeric variable, the system uses the number-builder routine to strip away all non-numeric characters and spaces and assign the value 123 to the numeric variable. When you add to this the intricacies of real numbers and exponential notation, the situation becomes more complex. For more information about how the number builder works, see the chapter called "Entering Data" in *BASIC Interfacing Techniques.*

Because ASCII files require so much overhead (for storage of "small" items), and because retrieving numeric data from ASCII files is sometimes a complex process, they are not the preferred file type for numeric data when compactness is an important criteria. However, as we mentioned before, ASCII files are interchangeable with many other HP products.

In this chapter, we refer to the data representation described above as ASCII-data format. As mentioned earlier, you can also store data in BDAT files in ASCII format (by using the FORMAT ON attribute). Be careful not to confuse the ASCII-*file type* with the ASCII-*data format.* The ASCII format used in BDAT files when FORMAT ON is specified differs from the format used in ASCII files in several respects. Each item output to an ASCII file has its own length header; there are no length headers in a FORMAT ON

BDAT file. At the end of each OUTPUT statement an end-of-line sequence is written to a FORMAT ON BDAT file unless surpressed by an IMAGE or EOL OFF. No end-of-line sequence is written to an ASCII file at the end of an OUTPUT statement.

In general, you should only use ASCII files when you want to transport data between HP Series 200/300 computers and other HP machines. There may be other instances where you will want to use ASCII files, but you should be aware that they cause a *noticeable transfer rate degradation* compared to BDAT and HP-UX files (especially for numeric data items).

Formatted OUTPUT with ASCII Files

As mentioned in the "Brief Comparison of File Types," you cannot format items sent to ASCII files; that is, you **cannot** use the following statement with an ASCII file:

```
OUTPUT @Ascii_file USING "#,DD.D,4X,5A";Number,String$
```

You can, however, direct the output to a string variable first, and then OUTPUT this formatted string to an ASCII file:

```
   OUTPUT String_var$ USING "#,DD.D,4X,5A";Number,String$
   OUTPUT @Ascii_file;String_var$
```

When a string variable is specified as the destination of data in an OUTPUT statement, source items are evaluated individually and placed into the variable according to the free-field rules or the specified image, depending on which type of OUTPUT statement is used. Thus, item terminators may or may not be placed into the variable. The ASCII data representation is always used during outputs to string variables; in fact, **data output to string variables is exactly like that sent to devices through I/O paths with the FORMAT ON attribute**.

When using OUTPUT to a string, characters are always placed into the variable beginning at the first position; no other position can be specified as the beginning position at which data will be placed. Thus, *random access of the information in string variables is not allowed* from OUTPUT and ENTER statements; all data must be accessed serially. For instance, if the characters "1234" are output to a string variable by one OUTPUT statement, and a subsequent OUTPUT statement outputs the characters "5678" to the same variable, the second output **does not** begin where the first one left off (i.e., at string position five). The second OUTPUT statement begins placing characters in position one, just as the first OUTPUT statement did, overwriting the data initially output to the variable by the first OUTPUT statement.

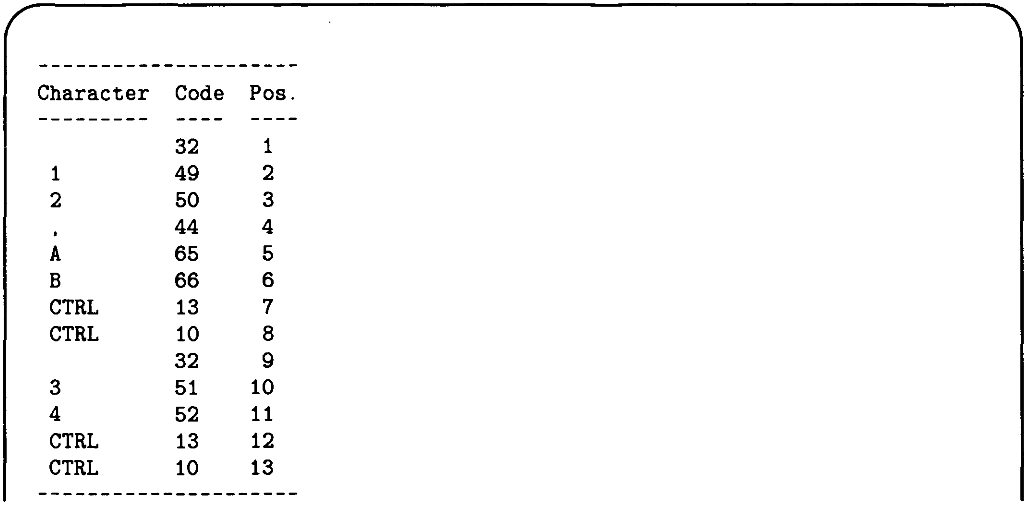
The string variable's length header (2 bytes) is updated and compared to the dimensioned length of the string as characters are output to the variable. If the string is filled before all items have been output, an error is reported; however, the string contains the first *n* characters output (where *n* is the dimensioned length of the string).

Example

The following program outputs string and numeric data items to a string variable and then calls a subprogram which displays each character, its decimal code, and its position within the variable. Even though this program does not write to an ASCII file it shows a character representation of what would appear in an ASCII file.

```
100   ASSIGN @Crt TO 1  ! CRT is disp. device.
110   !
120   OUTPUT Str_var$;12,"AB",34
130   !
140   CALL Read_string(@Crt,Str_var$)
150   !
160   END
170   !
180   !
190 SUB Read_string(@Disp,Str_var$)
200     !
210     ! Table heading.
220     OUTPUT @Disp;"---------------------"
230     OUTPUT @Disp;"Character  Code  Pos."
240     OUTPUT @Disp;"---------  ----  ----"
250     Dsp_img$="2X,4A,5X,3D,2X,3D"
260     !
270     ! Now read the string's contents.
280   FOR Str_pos=1 TO LEN(Str_var$)
290         Code=NUM(Str_var$[Str_pos;1])
300       IF Code<32 THEN ! Don't disp. CTRL chars.
310           Char$="CTRL"
320       ELSE
330           Char$=Str_var$[Str_pos;1] ! Disp. char.
340       END IF
350       !
360       OUTPUT @Disp USING Dsp_img$;Char$,Code,Str_pos
370   NEXT Str_pos
380   !
390   ! Finish table.
400   OUTPUT @Disp;"---------------------"
410   OUTPUT @Disp ! Blank line.
420   !
430   SUBEND
```

Final Display



```
---------------------
Character  Code  Pos.
---------  ----  ----
            32    1
 1          49    2
 2          50    3
 ,          44    4
 A          65    5
 B          66    6
 CTRL       13    7
 CTRL       10    8
            32    9
 3          51   10
 4          52   11
 CTRL       13   12
 CTRL       10   13
---------------------
```

Outputting data to a string and then examining the string's contents is usually a more convenient method of examining output data streams than using a mass storage file. The preceding subprogram may facilitate the search for control characters. They are not displayed, because they might cause the printer or CRT to perform control actions.

The following example program shows how outputs to string variables can be used to reduce the overhead required in ASCII data files. To do this, the program compares two possible methods for storing data in an ASCII data file. The first method stores 64 two-byte items in a file one at a time. Each two-byte item is preceded by a two-byte length header. The second method stores 64 two-byte items in a string array which is output to a string variable. The string variable is then output to an ASCII data file with only one two-byte length header being used. Since the second method used only one two-byte length header to store 64 two-byte items, it can easily be seen that the second method required less overhead. Note that the second method is also the *only way to format data sent to ASCII data files.*

```
100   PRINTER IS CRT
110   !
120   ! Create a file 1 record long (=256 bytes).
130   ON ERROR GOTO File_exists
140   CREATE ASCII "TABLE",1
150 File_exists:   OFF ERROR
160                !
170                !
180   ! First method outputs 64 items individually..
190   ASSIGN @Ascii TO "TABLE"
200   FOR Item=1 TO 64  ! Store 64 2-byte items.
210       OUTPUT @Ascii;CHR$(Item+31)&CHR$(64+RND*32)
220       STATUS @Ascii,5;Rec,Byte
230       DISP USING Image_1;Item,Rec,Byte
240   NEXT Item
250 Image_1: IMAGE "Item ",DD,"  Record ",D,"  Byte ",3D
260   DISP
270   Bytes_used=256*(Rec-1)+Byte-1
280   PRINT Bytes_used;" bytes used with 1st method."
290   PRINT
300   PRINT
310   !
320   !
330   ! Second method consolidates items.
340   DIM Array$(1:64)[2],String$[128]
350   ASSIGN @Ascii TO "TABLE"
360   !
370   FOR Item=1 TO 64
380       Array$(Item)=CHR$(Item+31)&CHR$(64+RND*32)
390   NEXT Item
400   !
410   OUTPUT String$;Array$(*); ! Consolidate in string variable.
420   OUTPUT @Ascii;String$     ! OUTPUT to file as 1 item.
430   !
440   STATUS @Ascii,5;Rec,Byte
450   Bytes_used=256*(Rec-1)+Byte-1
460   PRINT Bytes_used;" bytes used with 2nd method."
470   !
480   END
```

The program shows many of the features of using ASCII files and string variables. The first method of outputting the data items shows how the file pointer varies as data are sent to the file. Note that the file pointer points to the **next** file position at which a subsequent byte will be placed. In this case, it is incremented by four by every OUTPUT statement (since each item is a two-byte quantity preceded by a two-byte length header).

The program could have used a BDAT file, which would have resulted in using slightly less disc-media space; however, using BDAT files usually saves much more disc space than would be saved in this example. The program does not show that **ASCII files cannot be accessed randomly**; this is one of the major differences between using ASCII and BDAT (and HP-UX) files.

Example

The VAL$ function (or a user-defined function subprogram) and outputs made to string variables can be used to generate the string representation of a number. The advantage of the latter method is you can explicitly specify the number's image. The following program compares a string generated by the VAL$ function to that generated by outputting a number to a string variable.

```
100   X=12345678
110   !
120   PRINT VAL$(X)
130   !
140   OUTPUT Val$ USING "#,3D.E";X
150   PRINT Val$
160   !
170   END
```

Printed Results

```
1.2345678E+7
123.E+05
```

Formatted ENTER with ASCII Files

Data is entered from string variables in much the same manner as output to the variable. For example,

```
ENTER @File;String$
ENTER String$;Var1, Var2$
```

All ENTER statements that use string variables as the data source interpret the data according to the FORMAT ON attribute. Data is read from the variable beginning at the first string position; if a subsequent ENTER statement reads characters from the variable, the read also begins at the first position. If more data is to be entered from the string than is contained in the string, an error is reported; however, all data entered into the destination variable(s) before the end of the string was encountered remain in the variable(s) after the error occurs.

When entering data from a string variable, the computer keeps track of the number of characters taken from the variable and compares it to the string length. Thus, **statement-termination** conditions are **not** required; the ENTER statement automatically terminates when the last character is read from the variable. However, **item** terminators are still required **if** the items are to be separated **and** the lengths of the items are not known. If the length of each item is known, an image can be used to separate the items.

Example

The following program shows an example of the need for **either** item terminators **or** length of each item. The first item was not properly terminated and caused the second item to not be recognized.

```
100   OUTPUT String$;"ABC123";   ! OUTPUT w/o CR/LF.
110   !
120   ! Now enter the data.
130   ON ERROR GOTO Try_again
140   !
150 First_try: !
160   ENTER String$;Str$,Num
170   OUTPUT CRT;"First try results:"
180   OUTPUT CRT;"Str$= ";Str$,"Num=";Num
190   BEEP     ! Report getting this far.
200   STOP
210   !
220 Try_again: OUTPUT CRT;"Error";ERRN;" on 1st try"
230            OUTPUT CRT;"STR$=";Str$,"Num=";Num
240            OUTPUT CRT
250            OFF ERROR  ! The next one will work.
260            !
270   ENTER String$ USING "3A,3D";Str$,Num
```

```
280    OUTPUT CRT;"Second try results:"
290    OUTPUT CRT;"Str$= ";Str$,"Num=";Num
300    !
310    END
```

Executing the above program produces the following results:

```
Error 153 on 1st try
Str$=ABC123
Num= 0

Second try results:
Str$= ABC
Num= 123
```

This technique is convenient when attempting to enter an unknown amount of data or when numeric and string items within incoming data are not terminated. The data can be entered into a string variable and then searched by using images.

Example

ENTERs from string variables can also be used to generate a number from ASCII numeric characters (a recognizable collection of decimal digits, decimal point, and exponent information), rather than using the VAL function. As with outputs to string variables, images can be used to interpret the data being entered.

```
30     Number$="Value= 43.5879E-13"
40     !
50     ENTER Number$;Value
60     PRINT "VALUE=";Value
70     END
```

Example

An ASCII file can always be read as strings even if the data is numeric. The following program reads any ASCII file that has lines that are 80 characters or less in length.

```
10     DIM Buf$[80]
20     ASSIGN @Str_file TO "File"
30     ON END @Str_file GOTO Ending
40     LOOP
50      ENTER @Str_file;Buf$
60      PRINT Buf$
70     END LOOP
80 Ending: !
90     END
```

A Closer Look at BDAT and HP-UX Files

As mentioned earlier, BDAT and HP-UX files are designed for flexibility (random and serial access, choice of data representations), storage-space efficiency, and speed. This chapter provides several examples of using these types of files. (The "Porting and Sharing Files" chapter also contains several examples of using HP-UX files from BASIC as well as from HP-UX languages.)

Data Representations Available

The data representations available are:

- BASIC internal formats (allow the fastest data rates and are generally the most space-efficient)
- ASCII format (the most interchangeable)
- Custom formats (design your own data representations using IMAGE specifiers)

More details of each type of representation are described in the remainder of this section.

Random vs. Serial Access

Random access means that you can directly read from and write to any record within the file, while serial access only permits you to access the file in order, from the beginning. That is, you must read records 1, 2, ..., $n-1$ before you can read record n. Serial access can waste a lot of time if you're trying to access data at the end of a file. On the other hand, if you want to access the entire file sequentially, you are better off using serial access than random access, because it generally requires less programming effort and often uses less file space. BDAT and HP-UX files can be accessed both randomly and serially, while ASCII files can be accessed only serially.

Data Representations Used in BDAT Files

BDAT files allow you to store and retrieve data using internal format, ASCII format, or user-defined formats.

- With internal format (FORMAT OFF), items are represented with the same format the system uses to store data in internal computer memory. (This is the default FORMAT for BDAT and HP-UX files.)
- With ASCII format (FORMAT ON), items are represented by ASCII characters.
- User-defined formats are implemented with programs that employ OUTPUT and ENTER statements that reference IMAGE specifiers (items are represented by ASCII characters).

Complete descriptions of ASCII and user-defined formats are given in *BASIC Interfacing Techniques.* This section shows the details of internal (FORMAT OFF) representations of numeric and string data.

BDAT Internal Representations (FORMAT OFF)

In most applications, you will use internal format for BDAT files. Unless we specify otherwise, you can assume that when we talk about retrieving and storing data in BDAT files, we are also talking about internal format. This format is synonymous with the FORMAT OFF attribute, which is described later in this chapter.

Because FORMAT OFF assigned to BDAT files uses almost the same format as internal memory, very little interpretation is needed to transfer data between the computer and a FORMAT OFF file. FORMAT OFF files, therefore, not only save space but also save time.

Data stored in internal format in BDAT files require the following number of bytes per item:

Data Type	Internal Representation
INTEGER	2 bytes
REAL	8 bytes
COMPLEX	16 bytes (same as 2 REALs)
String	4-byte length header; 1 byte per character (plus 1 pad byte if string length is an odd number)

INTEGER values are represented in BDAT files which have the FORMAT OFF attribute by using a 16-bit, two's-complement notation, which provides a range −32 768 thru 32 767. If bit 15 (the MSB) is 0, the number is positive. If bit 15 equals 1, the number is negative; the value of the negative number is obtained by changing all ones to zeros, and all zeros to ones, and then adding one to the resulting value.

Examples

Binary Representation	Decimal Equivalent
00000000 00010111	23
11111111 11101000	−24
10000000 00000000	−32768
01111111 11111111	32767
11111111 11111111	−1
00000000 00000001	1
00100011 01000111	9031
11011100 10111001	−9031

REAL values are stored in BDAT files by using their internal format (when FORMAT OFF is in effect): the IEEE-standard, 64-bit, floating-point notation. Each REAL number is comprised of two parts: an exponent (11 bits), and a mantissa (53 bits). The mantissa uses a sign-and-magnitude notation. The sign bit for the mantissa is not contiguous with the rest of the mantissa bits; it is the most significant bit (MSB) of the entire eight bytes. The 11-bit exponent is offset by 1 023 and occupies the 2nd through the 12th MSB's. Every REAL number is internally represented by the following equation. (Note that the mantissa is in binary notation):

$$-1^{\text{mantissa sign}} \times 2^{\text{exponent} - 1023} \times 1._{\text{mantissa}}$$

The figure below shows how the real number "1/3" would be stored in a BDAT file.

Byte	1	2	3	4	...	8
Decimal value of character	63	213	85	85	...	85
Binary value of characters	00111111	11010101	01010101	01010101	...	01010101

$$\overset{\uparrow}{0}\,\underbrace{0111111\ 11010101}_{\text{exponent}}\ \underbrace{01010101\ 01010101\ \ldots\ 01010101}_{\text{mantissa}}$$

mantissa sign

COMPLEX values are always stored as two REAL values.

String data are stored in FORMAT OFF BDAT files in their internal format.

- A 4-byte length header contains a value that specifies the length of the string (the 2 leading bytes of length header are always 0 for Series 200/300 computers).
- Every character in a string is represented by one byte which contains the character's ASCII code. If the length of the string is odd, a pad character is appended to the string to get an even number of characters; however, the length header does not include this pad character.

Examples

If stored as a string value, the number "45" would be:

$$\underbrace{00000000\ 00000000\ 00000000\ 00000010}_{\text{Length = 0002 (binary)}}\ \underbrace{00110100}_{\text{ACSII 52}}\ \underbrace{00110101}_{\text{ASCII 53}}$$

The string "A" would be stored:

$$\underbrace{00000000\ 00000000\ 00000000\ 00000001}_{\text{Length = 0001 (binary)}}\ \underbrace{01000001}_{\text{ASCII 65}}\ \underbrace{00100000}_{\text{ASCII 32}}$$

In this case, the space character (ASCII code 32) is used as the pad character; however, not all operations use the space as the pad character.

ASCII and Custom Data Representations

When using the ASCII data format for BDAT files, all data items are represented with ASCII characters. With user-defined formats, the image specifiers referenced by the OUTPUT or ENTER statement are used to determine the data representation (which is ASCII characters).

```
OUTPUT @File USING "SDD.DD,XX,B,#";Number,Binary_value
ENTER  @File USING "B,B,40A,%";Bin_val1,Bin_val2,String$
```

Using both of these formats with BDAT files produce results identical to using them with devices. The entire subject is described fully in *BASIC Interfacing Techniques*. The topic of advanced transfer techniques for BDAT files is described in the same manual.

Data Representations with HP-UX Files

HP-UX files are **very similar to BDAT files**. The **only differences** between the two are:

- The internal representation (FORMAT OFF) of strings is slightly different:
  - HP-UX FORMAT OFF strings have no length header; instead, they are terminated by a null character, CHR$(0).
  - BDAT FORMAT OFF strings have a 4-byte length header;
- HP-UX files have a **fixed record length of 1**. (BDAT files allow user-definable record lengths.)
- HP-UX files have **no system sector** like BDAT files do (see the next section for details).

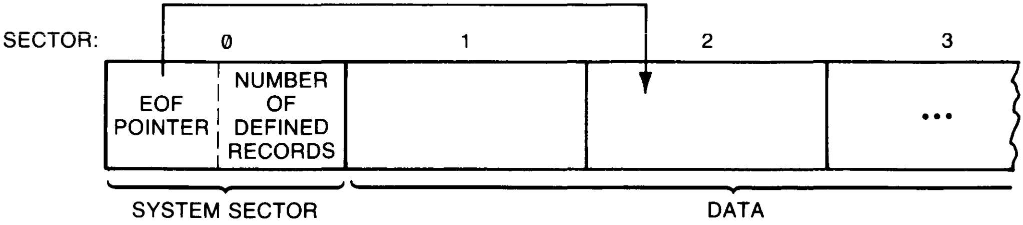
The FORMAT ON representations for HP-UX files are the same as for devices. The entire subject is described fully in *BASIC Interfacing Techniques*. The topic of advanced transfer techniques for HP-UX files is described in the same manual.

Note

Throughout this section, you should be able to assume that unless otherwise stated the techniques shown will apply to both BDAT and HP-UX files.

BDAT File System Sector

On the disc, every BDAT file is preceded by a system sector that contains an end-of-file (EOF) pointer and the number of defined records in the file. All data is placed in succeeding sectors. You cannot directly access the system sector. However, as you shall see later, it is possible to indirectly change the value of an EOF pointer.



EOF Pointer:
- number of sectors from beginning of file (32-bit binary number)
- number of bytes from beginning of sector (32-bit binary number)

Number of defined records: See description below (32-bit binary number)

Defined Records

To access a BDAT or HP-UX file randomly, you specify a particular defined record. Records are the smallest units in a file directly addressable by a random OUTPUT or ENTER.

- With BDAT files, defined records can be anywhere from 1 through 65 534 bytes long.
- With HP-UX files, defined records are always 1 byte long.

Specifying Record Size (BDAT Files Only)

Both the length of the file and the length of the defined records in it are specified when you create a BDAT file. This section shows how to specify the record length of a BDAT file. (The next section talks about how to choose the record length.)

For example, the following statement would create a file called `Example` with 7 defined records, each record being 128 bytes long:

```
CREATE BDAT "Example",7,128
```

If you don't specify a record length in the CREATE BDAT statement, the system will set each record to the default length of 256 bytes.

Both the record length and the number of records are rounded to the nearest integer. Further, the record length is rounded up to the nearest even integer. For example, the statement:

```
CREATE BDAT "Odd",3.5,28.7
```

would create a file with 4 records, each 30 bytes long. On the other hand, the statement:

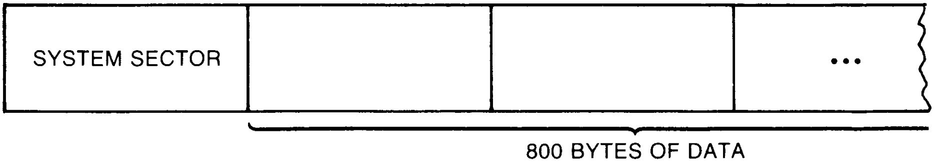
```
CREATE "Odder",3.49,28.3
```

would create a file with 3 records, each 28 bytes long.

Once a file is created, you cannot change its length, or the length of its records. You must therefore calculate the record size and file size required *before* you create a file.

Choosing A Record Length (BDAT Files Only)

Record length is important only for random OUTPUTs and ENTERs. It is not important for serial access. The most important consideration in selecting of a proper record length is the type of data being stored and the way you want to retrieve it. Suppose, for instance, that you want to store 100 real numbers in a file, and be able to access each number individually. Since each REAL number uses 8 bytes, the data itself will take up 800 bytes of storage.



The question is how to divide this data into records. If you define the record length to be 8 bytes, then each REAL number will fill a record. To access the 15th number, you would specify the 15th record. If the data is organized so that you are always accessing two data items at a time, you would want to set the record length to 16 bytes.

The worst thing you can do with data of this type is to define a record length that is not evenly divisible by eight. If, for example, you set the record length to four, you would only be able to randomly access half of each real number at a time. In fact, the system will return an End-Of-Record condition if you try to randomly read data into REAL variables from records that are less than 8 bytes long.

So far, we have been talking about a file that contains only REAL numbers. For files that contain only INTEGERs, you would want to define the record length to be a multiple of two. To access each INTEGER individually, you would use a record length of two; to access two INTEGERs at a time, you would use a record length of four, and so on.

Files that contain string data present a slightly more difficult situation since strings can be of variable length. If you have three strings in a row that are 5, 12, and 18 bytes long, respectively, there is no record length less than 22 that will permit you to randomly access each string. If you select a record length of 10, for instance, you will be able to randomly access the first string but not the second and third.

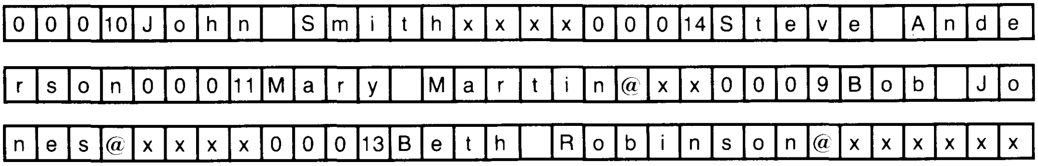
If you want to access strings randomly, therefore, you should make your records long enough to hold the largest string. Once you've done this, there are two ways to write string data to a BDAT file. The first, and easiest, is to output each string in random mode. In other words, select a record length that will hold the longest string and then write each string into its own record. Suppose, for example, that you wanted to OUTPUT the following 5 names into a BDAT file and be able to access each one individually by specifying a record number.

John Smith
Steve Anderson
Mary Martin
Bob Jones
Beth Robinson

The longest name, "Steve Anderson", is 14 characters. To store it in a BDAT file would require 18 bytes (four bytes for the length header). So you could create a file with record length of 18 and then OUTPUT each item into a different record:

```
100   CREATE BDAT "Names",5,18          ! Create a file.
110   ASSIGN @File TO "Names"           ! Open the file (FORMAT OFF).
120   OUTPUT @File,1;"John Smith"       ! Write names to
130   OUTPUT @File,2;"Steve Anderson"   !  successive records
140   OUTPUT @File,3;"Mary Martin"      !  in file.
150   OUTPUT @File,4;"Bob Jones"
160   OUTPUT @File,5;"Beth Robinson"
```

On the disc, the file Names would look like the figure below. The four-byte length headers show the decimal value of the bytes in the header. The data are shown in ASCII characters.



0	0	0	10	J	o	h	n		S	m	i	t	h	x	x	x	x	0	0	0	14	S	t	e	v	e		A	n	d	e
r	s	o	n	0	0	0	11	M	a	r	y		M	a	r	t	i	n	@	x	x	0	0	0	9	B	o	b		J	o
n	e	s	@	x	x	x	x	0	0	0	13	B	e	t	h		R	o	b	i	n	s	o	n	@	x	x	x	x	x	x

1 = length header
x = whatever data previously resided in that space
@ = pad character

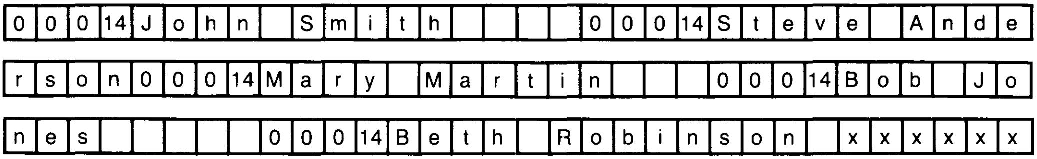
The unused portions of each record contain whatever data previously occupied that physical space on the disc.

The other method for writing strings to a BDAT or HP-UX file is to pad each entry so that they are all of uniform length. While this method involves more programming, it allows you to pad the unused portions of each record with whatever characters you choose. It also permits you to read and write the data serially as well as randomly. The program below shows how you might enter the five names into a file by padding each name with spaces.

```
100   CREATE BDAT "Names",5,18      ! Create file.
110   ASSIGN @Path1 TO "Names"      ! Open the file (FORMAT OFF).
120   FOR Entry=1 TO 5
130      LINPUT Name$[1;14]         ! Get names from keyboard.
140      OUTPUT @Path1;Name$        ! Write name to file.
150   NEXT Entry
```

In this program, we input each name from the keyboard and then pad the name with spaces so that its length is 14 bytes. With the four-byte length header, each entry is 18 bytes, or one record. In line 140, we write the name serially to the file. Since every data item is 18 bytes, there is no need to write randomly, although we could have if we wanted to. Since the LINPUT statement is limited to 14 bytes, any names that are longer than 14 characters are automatically truncated.

If we had used the second program to enter the names, file **Names** would look like the figure below:



0	0	0	14	J	o	h	n		S	m	i	t	h					0	0	0	14	S	t	e	v	e		A	n	d	e
r	s	o	n	0	0	0	14	M	a	r	y		M	a	r	t	i	n				0	0	0	14	B	o	b		J	o
n	e	s						0	0	0	14	B	e	t	h		R	o	b	i	n	s	o	n		x	x	x	x	x	x

EOF Pointers

There are two types of End-Of-File pointers associated with BDAT and HP-UX files:

- Logical EOF pointer in the I/O path table—maintained in the table of the I/O path currently assigned to the file.
- Logical EOF pointer on the volume—resides on the physical volume that contains the file:
  - With BDAT files, it is in the system sector.
  - With HP-UX files, the EOF pointer is stored in one of two places:
    - On HFS volumes, it is read from the size stored in the file's inode.
    - On LIF volumes, it is read from a long word stored in the directory.

The two pointers are always updated at the same time so that they always agree with one another. (This may not be true if you use more than one I/O path to OUTPUT data to one file.) The two pointers are updated when either of the two conditions below occur.

- If, after an OUTPUT statement has been executed, the file pointer value is greater than the EOF pointers, the EOF pointers are moved to the file pointer position.
- If an OUTPUT statement contains the "END" secondary word, the EOF pointers are moved to the file pointer position regardless of their current values.

The function of EOF pointers is to mark the logical end of a data file. Every file also has a physical EOF on a volume--the last byte reserved for the file when you create it. The EOF pointers cannot point beyond the physical EOF. The EOF pointer marks the point at which no more data can be read. Also, you cannot randomly write data more than one record past the current EOF position.

If you have a 100-record file, and the EOF pointers point to the 50th record, records 50 through 100 cannot be read. If you attempt to read data beyond an EOF pointer, an EOF condition occurs. EOF conditions can be trapped with an ON END statement. If you do not trap it, an EOF condition will cause Error 59. Attempting to read or write beyond the physical EOF on a volume will also result in an EOF condition. EOF conditions are described in more detail later in this chapter. Note that files on SRM and HFS discs are extensible. Thus the file is extended rather than getting a physical EOF condition.

Moving EOF Pointers

When you first create a file, the logical EOF on a volume has a pointer which points to the first byte in the file. When you ASSIGN an I/O path to a file, the logical EOF pointer on a volume is copied to the I/O path table. As you OUTPUT data items to the file, both EOF pointers are moved so that they point to the next byte. This is also where the file pointer is positioned.

If you overwrite a file, however, the EOF pointers will not necessarily agree with the file pointer. For example, suppose you write 100 bytes to a file, and then re-ASSIGN the I/O path. By re-ASSIGNing, you move the file pointer back to the first byte in the file. The EOF pointers, though, still point to the 101st byte. They will not be changed until the file pointer value is greater than 101, or until you specify an "END" in an OUTPUT statement.

The secondary word "END" is used to move the EOF pointers backwards. It forces the EOF pointers to be re-positioned to the file pointer byte even if it is earlier in the file than their current position. In effect, this shrinks the file, causing data that lies past the new EOF position to become inaccessible.

Writing Data

Data is always written to a file with an OUTPUT statement via an I/O path. You can OUTPUT numeric and string variables, numeric and string expressions, and numeric and string arrays. When you OUTPUT data with the FORMAT OFF, data items are written to the file in internal format (described earlier).

There is no limit to the number of data items you can write in a single OUTPUT statement, except that program statements are limited to two CRT lines. Also, if you try to OUTPUT more data than the file can hold, or the record can hold (if you are using random access), the system will return an EOF or EOR condition. If an EOF or EOR condition occurs, the file retains any data output before the end condition occurred.

There is also no restriction on mixing different types of data in a single OUTPUT statement. The system decides which data type each item is before it writes the item to the disc. Any item enclosed in quotes is a string. Numeric variables and expressions are OUTPUT according to their type (16 bytes for COMPLEX values, 8 bytes for REAL values, and 2 bytes for INTEGER values). Arrays are written to the file in row-major order (right-most subscript varies quickest).

Each data item in an OUTPUT statement should be separated by either a comma or semi-colon (there is no operational difference between the two separators with FORMAT OFF). Punctuation at the end of an OUTPUT statement is ignored with FORMAT OFF.

Serial OUTPUT

Data is written serially to BDAT and HP-UX files whenever you do not specify a record number in an OUTPUT statement. When writing data serially, each data item is stored immediately after the previous item (with FORMAT OFF in effect, there are no separators between items). Sector and record boundaries are ignored. Data items are written to the file one by one, starting at the current position of the file pointer. As each item is written, the file pointer is moved to the byte following the last byte of the preceding item. After all of the data items have been OUTPUT, the file pointer points to the byte following the last byte just written.

There are a number of circumstances where it is faster and easier to use serial access instead of random access. The most obvious case is when you want to access the entire file sequentially. If, for example, you have a list of data items that you want to store in a file and you know that you will never want to read any of the items individually, you should write the data serially. The fastest way to write data serially is to place the data in an array and then OUTPUT the entire array at once.

Another situation where you might want to use serial access is if the file is so small that it can fit entirely into internal memory at once. In this case, even if you want to change individual items, it might be easier to treat the entire file as one or more arrays, manipulate as desired, and then write the entire array(s) back to the file.

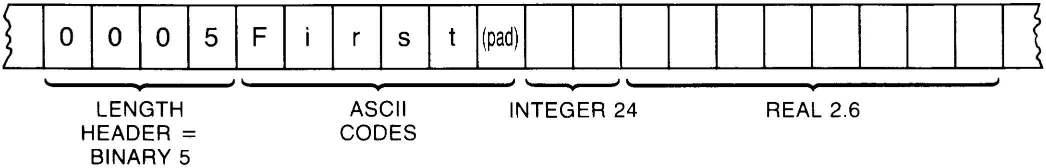
The examples below illustrate how data is stored serially in a BDAT file[1]. Assume that the following statement was used to open the file (and assign the FORMAT OFF attribute to the I/O path):

```
ASSIGN @Path1 TO "BDATorHPUX"; FORMAT OFF
```

The statement:

```
OUTPUT @Path1;"First",24;2.6,
```

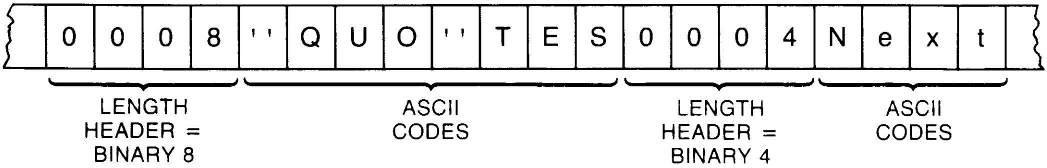
would result in the following storage format:



Note that quotation marks around a string are not written to the file. To write quote marks to a file, enter two quote marks for every one you want to OUTPUT. Note also that separators are not written to the file. To write a comma or semi-colon to a file, you must enclose it in quotes. For instance, the statement:

```
OUTPUT @Path1; """QUO""TE,","Next"
```

would be stored:

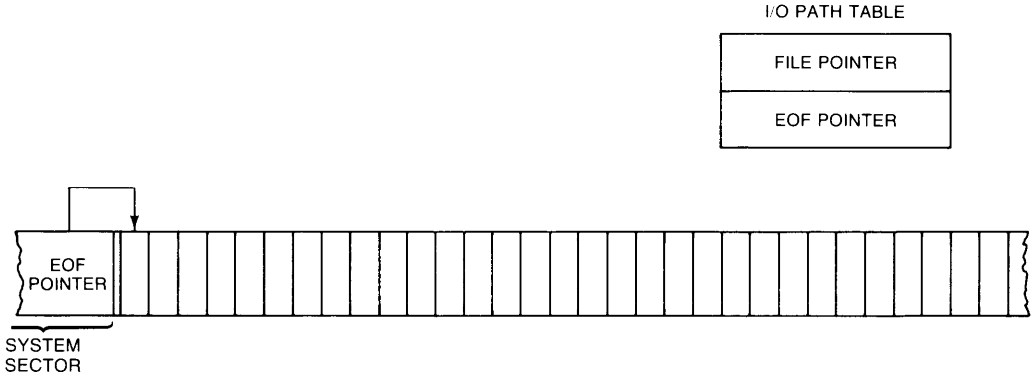


[1] Most of the details of the subsequent examples also pertain to how data items are written to HP-UX files. The main difference is that HP-UX files always have a defined record length of 1 byte.

The following sequence of serial OUTPUT statements show how data is written to a BDAT file and how the file pointer and EOF pointers are updated.

The following statement creates a BDAT file with four 128-byte records.

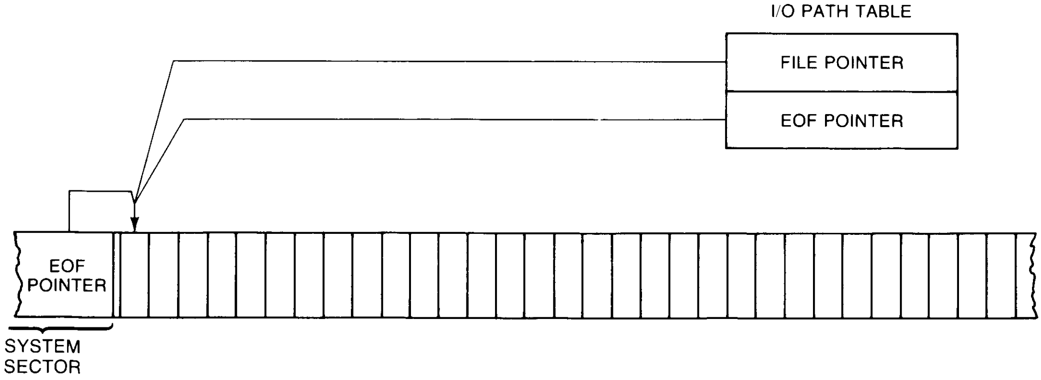
```
CREATE BDAT "Example",4,128
```



When the file is initially created, the logical EOF pointer on the volume points to the first byte in the file.

The following statement opens an I/O path to the file named **Example**.

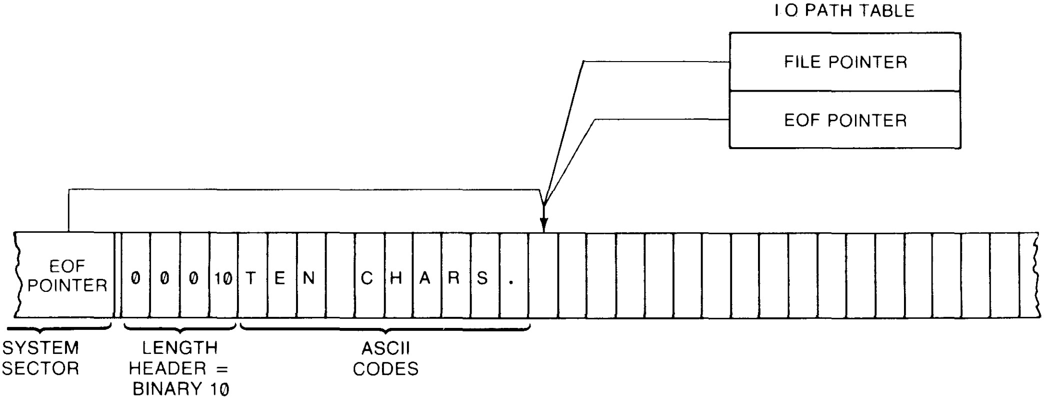
```
ASSIGN @Path1 TO "Example"
```



The logical EOF pointer on the volume is copied from the volume into the I/O path table. The file pointer is positioned to the beginning of the file.

Fourteen bytes are written to the file with this statement.

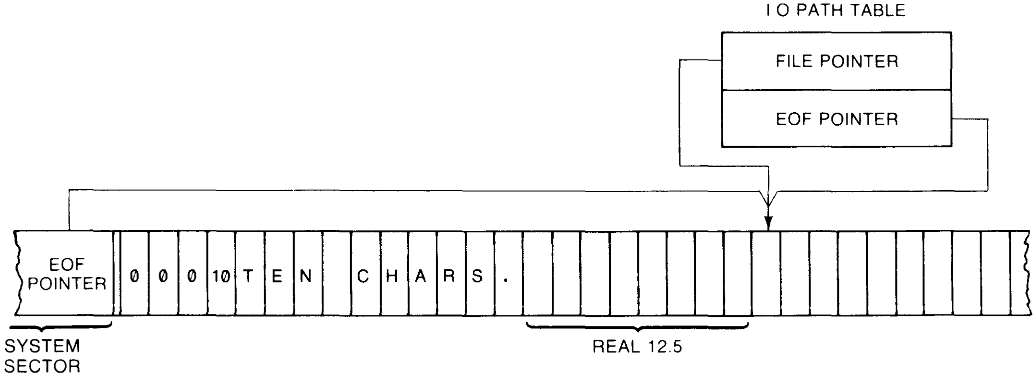
```
OUTPUT @Path1;"TEN CHARS."
```



The EOF pointers are moved to the 15th byte. The file pointer also points to the 15th byte.

This statement writes eight more bytes to the file.

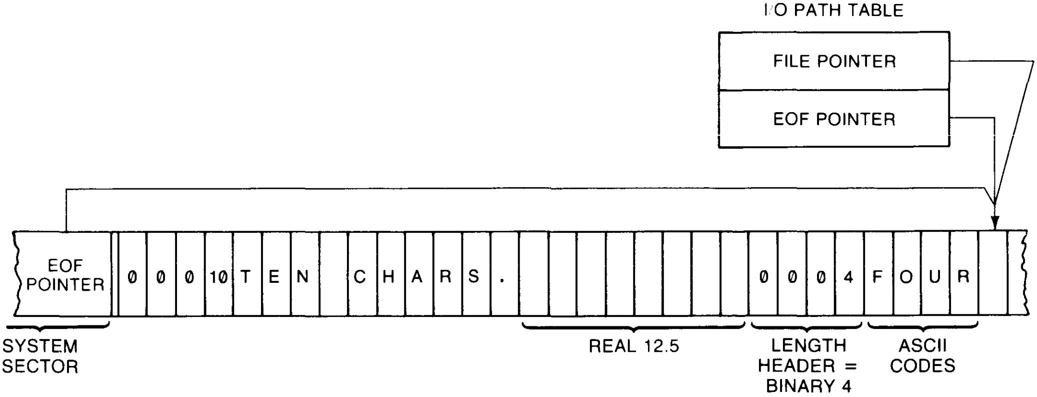
```
   OUTPUT @Path1;12.5,END
```



The file pointer now points to the 23rd byte. Both the logical EOF pointer in the I/O path table and the logical EOF pointer on the volume are updated to 23.

This statement writes eight more bytes to the file.

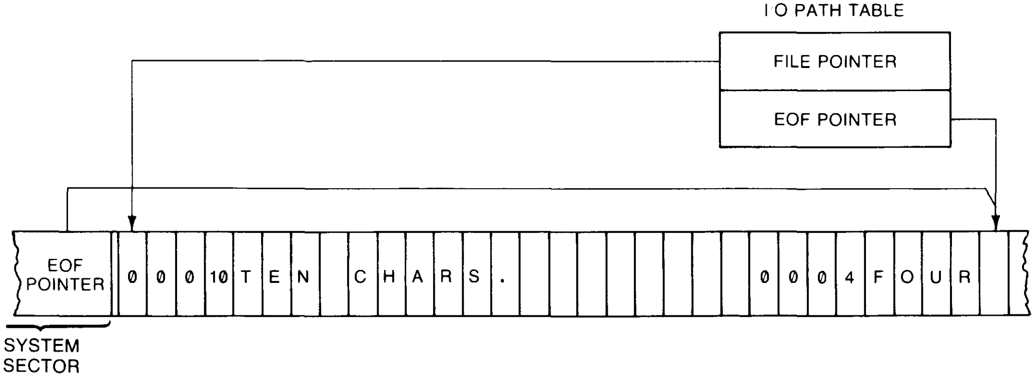
```
OUTPUT @Path1;"FOUR"
```



The file pointer now points to the 31st byte. The EOF pointers are updated to 31 because 31 is greater than 23, the current EOF value.

This statement re-assigns the I/O path name, which re-opens the file.

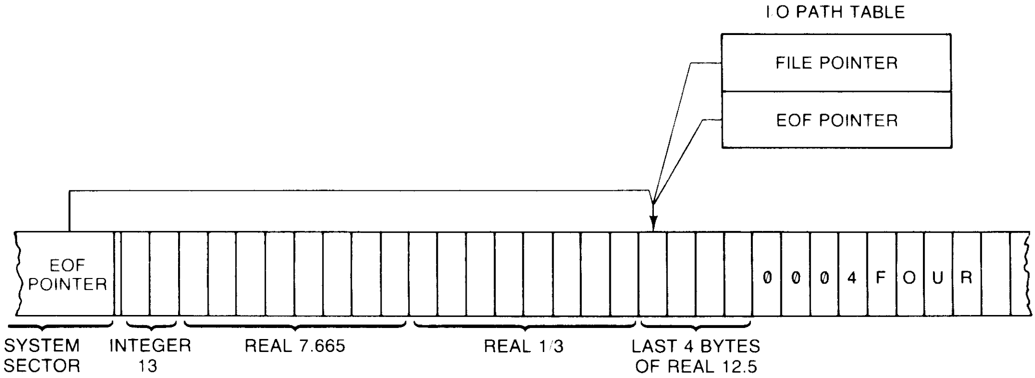
```
ASSIGN @Path1 TO "Example"
```



The file pointer is positioned back to the beginning of the file. The value of logical EOF pointer on a volume is copied into the logical EOF pointer in the I/O path table.

In this example, eighteen bytes (one INTEGER and two REALs) are OUTPUT, starting at the beginning of the file.

```
OUTPUT @Path1;13,7.665,1/3,END
```



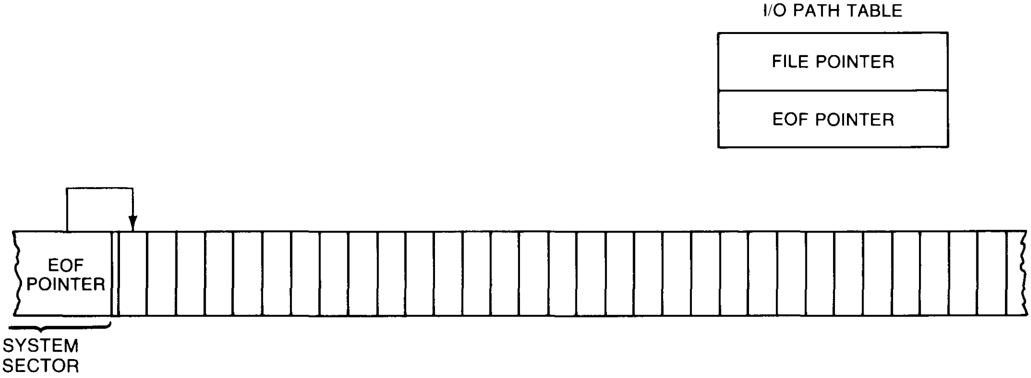
The original data, therefore, is overwritten. The file pointer points to the 19th byte. The EOF pointers are also positioned to 19 because the statement contains the "END" secondary word.

Random OUTPUT

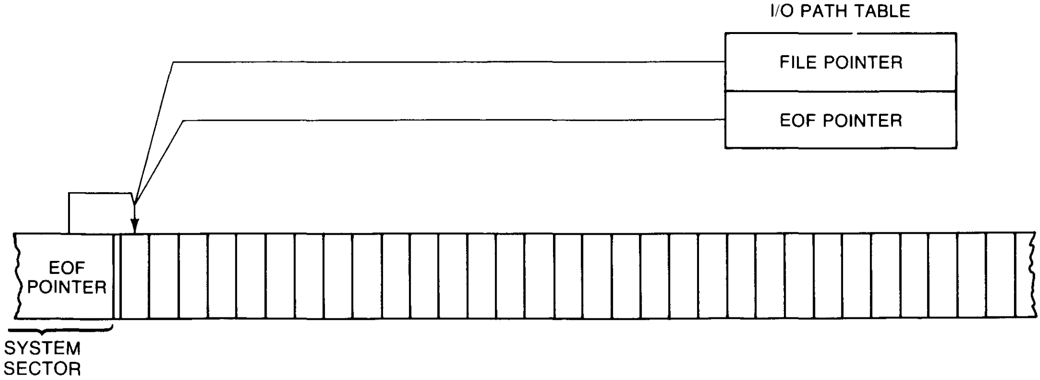
Random OUTPUT allows you to write to one record at a time. As with serial OUTPUT, there are EOF and file pointers that are updated after every OUTPUT. The EOF pointers follow the same rules as in serial access. The file pointer positioning is also the same, except that it is moved to the beginning of the specified record before the data is OUTPUT. If you wish to write randomly to a newly created file, either use a CONTROL statement to position the EOF in the last record, or start at the beginning of the file and write some "dummy" data into every record.

If you attempt to write more data to a record than the record will hold, the system will report an End-Of-Record (EOR) condition. An EOF condition will result if you try to write data more than one record past the EOF position. EOR conditions are treated by the system just like EOF conditions, except that they return Error 60 instead of 59 if they are not trapped by ON END. Data already written to the file before an EOR condition arises will remain intact. The examples below illustrate how data is stored randomly in a BDAT file. (HP-UX files can also be accessed randomly; however, you may recall that HP-UX files always have record lengths of 1 byte. Examples of accessing 1-byte records are shown in the subsequent section.)

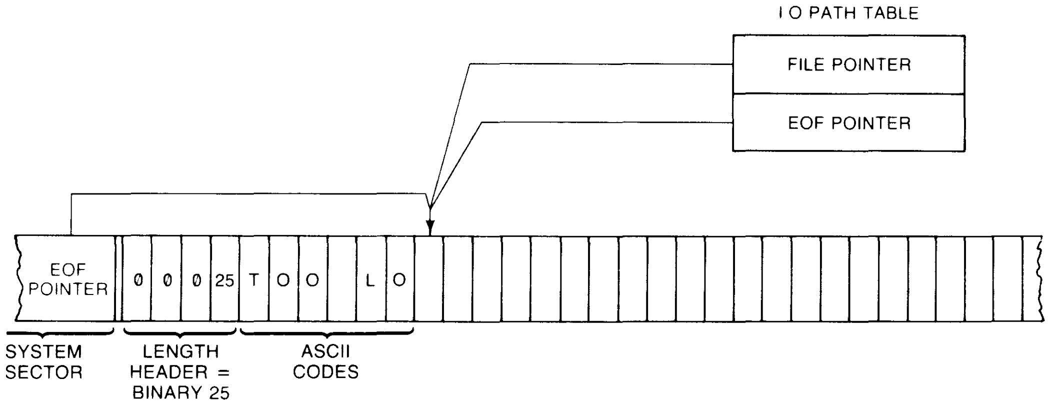
```
CREATE BDAT "Random",10,10
```



```
ASSIGN @Path2 TO "Random"
```

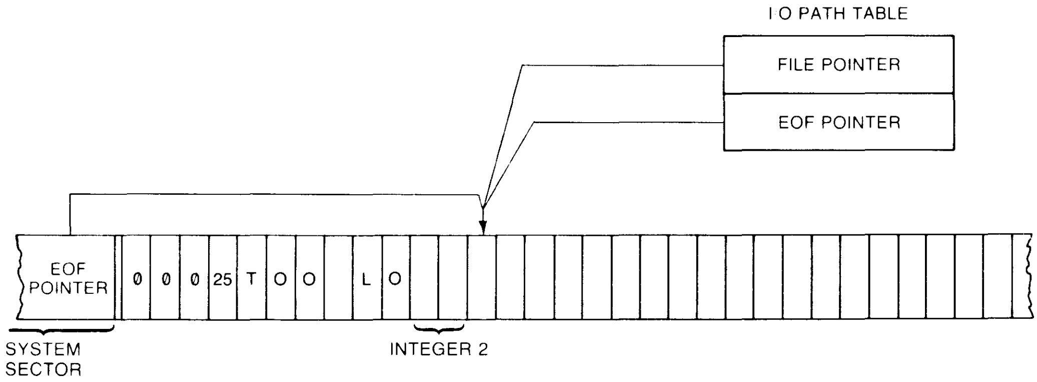


```
OUTPUT @Path2,1;"TOO LONG TO FIT IN RECORD"
```

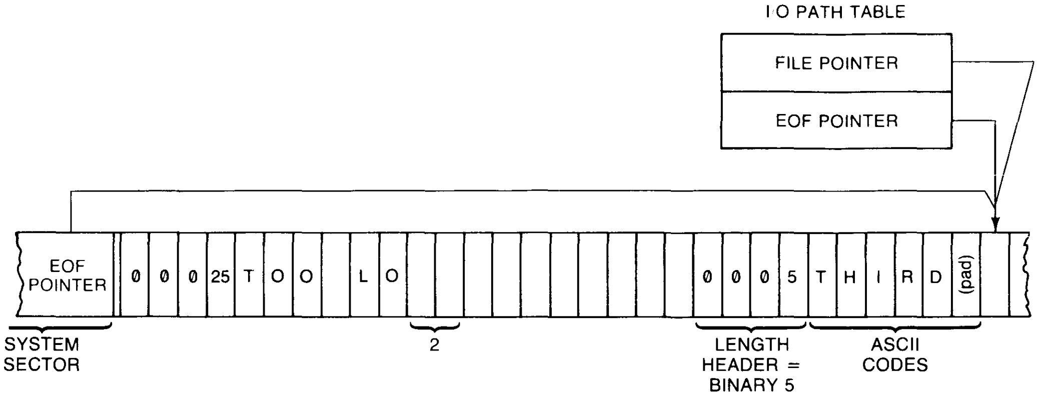


Even though this statement produces an EOR condition, the EOF pointers and file pointer are still updated. The ON END statement can be used to trap their error. Also, the length header represents the length of the string characters sent to the file, since the whole string is not written out.

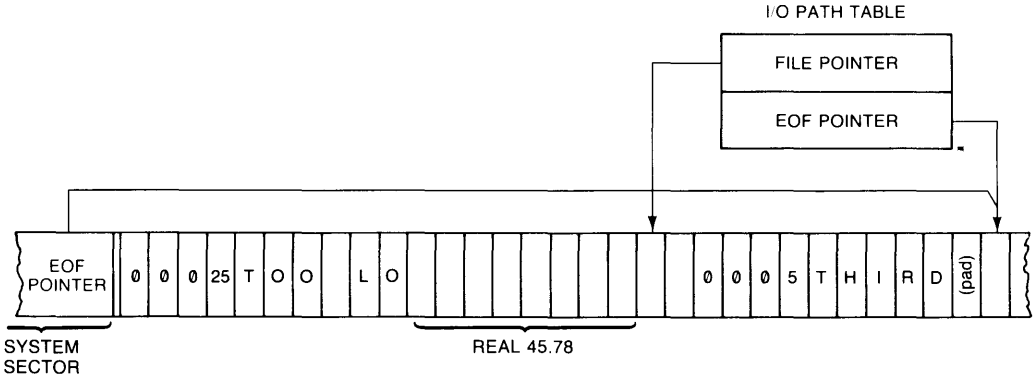
```
OUTPUT @Path2,2;2
```



```
OUTPUT @Path2,3;"THIRD"
```



```
OUTPUT @Path1,2;45.78
```



Reading Data From BDAT and HP-UX Files

Data is read from files with the ENTER statement. As with OUTPUT, data is passed along an I/O path. You can use the same I/O path you used to OUTPUT the data or you can use a different I/O path.

You can have several variables in a single ENTER statement. Each variable must be separated from the other variables by either a comma or semi-colon. It is extremely important to make sure that your variable types agree with the data types in the file. If you wrote a REAL number to a file, you should ENTER it into a REAL variable; INTEGERs should be entered into INTEGER variables; and strings into string variables. The rule to remember is:

Read it the way you wrote it.

That is the *only* technique that is always guaranteed to work.

In addition to making sure that data types agree, it is also advisable to make sure that access modes agree. If you wrote data serially, you should read it serially; and if you wrote it randomly, you should read it randomly. There are a few exceptions to this rule which we discuss later. However, you should be aware that mixing access modes can lead to erroneous results unless you are aware of the precise mechanics of the file system.

Reading String Data From a File

When reading string data from a file, you must enter it into a string variable. How the system does this depends on file type and FORMAT attribute assigned to the file:

- With FORMAT OFF assigned to a BDAT file, the system reads and interprets the first four bytes after the file pointer as a length header. It will then try to ENTER as many characters as the length header indicates. If the string has been padded by the system to make its length even, the pad character is not read into the variable.
- With FORMAT OFF assigned to an HP-UX file, strings have no length header. Instead, they are assumed to be null-terminated; that is, entry into the string terminates when a null character, CHR$(0), is encountered.
- With FORMAT ON assigned to either type of file, the system reads and interprets the bytes as ASCII characters. The rules for item and ENTER-statement termination match those for devices (see the "Entering Data" chapter of *BASIC Interfacing Techniques* for details.)

After an ENTER statement has been executed, the file pointer is positioned to the next unread byte. If the last data item was a padded string (written to a BDAT file when using FORMAT OFF), the file pointer is positioned after the pad. If you use the same I/O path name to read and write data to a file, the file pointer will be updated after every ENTER and OUTPUT statement. If you use different I/O path names, each will have its own file pointer which is independent of the other. However, be aware that each also has its own EOF pointer and that these pointers may not match, which can cause problems.

Entering data does not affect the EOF pointers. If you attempt to read past an EOF pointer, the system will report an EOF condition.

Serial ENTER

When you read data serially, the system enters data into variables starting at the current position of the file pointer and proceeds, byte by byte, until all of the variables in the ENTER statement have been filled. If there is not enough data in the file to fill all of the variables, the system returns an EOF condition. All variables that have already taken values before the condition occurs retain their values.

The following program creates a BDAT file, assigns an I/O path name to the file (with default FORMAT OFF attribute), writes five data items serially, and then retrieves the data items.

```
10   CREATE BDAT "STORAGE",1    ! Could also be an HP-UX file.
20   ASSIGN @Path TO "STORAGE"
30   INTEGER Num,First,Fourth
40   Num=5
60   OUTPUT @Path;Num,"squared"," equals",Num*Num,"."
70   ASSIGN @Path TO "STORAGE"
80   ENTER @Path;First,Second$,Third$,Fourth,Fifth$
90   PRINT First;Second$;Third$,Fourth,Fifth$
100  END
```

```
5 squared equals 25.
```

Note that we re-ASSIGNed the I/O path in line 70. This was done to re-position the file pointer to the beginning of the file. If we had omitted this statement, the ENTER would have produced an EOF condition.

Random ENTER

When you ENTER data in random mode, the system starts reading data at the beginning of the specified record and continues reading until either all of the variables are filled or the system reaches the EOR or EOF. If the system comes to the end of the record before it has filled all of the variables, an EOR condition is returned.

In the following example, we randomly OUTPUT data to 5 successive records, and then ENTER the data into an array in reverse order.

```
10      CREATE BDAT "SQ_ROOTS",5,2*8
20      ASSIGN @Path TO "SQ_ROOTS"  ! Default is FORMAT OFF.
30      FOR Inc=1 to 5
40         OUTPUT @Path,Inc;Inc,SQR(Inc)  ! Inc*8 is HP-UX record number.
50      NEXT Inc
60      FOR Inc=5 TO 1 STEP -1
70         ENTER @Path,Inc;Num(Inc),Sqroot(Inc)
80      NEXT Inc
90      PRINT "Number","Square Root"
100     FOR Inc=1 TO 5
110        PRINT Num(Inc),Sqroot(Inc)
120     NEXT Inc
130     END
```

```
Number     Square Root
 1          1
 2          1.41421356237
 3          1.73205080757
 4          2
 5          2.2360679775
```

In this example, there was no need to re-ASSIGN the I/O path because the random ENTER automatically re-positions the file pointer.

The comment on line **40** of the above program states how to be correctly positioned for an HP-UX file. For example, the output statement would look like this:

```
OUTPUT @Path,Inc*8;Inc,SQR(Inc)
```

Executing a random ENTER without a variable list has the effect of moving the file pointer to the beginning of the specified record. This is useful if you want to serially access some data in the middle of a file. Suppose, for instance, that you have a BDAT file containing 100 8-byte records, and each record has a REAL number in it. If you want to read the last 50 data items, you can position the file pointer to the 51st record and then serially read the remainder of the file into an array.

```
           :
100    REAL Array(50)
110    ENTER @Realpath,51;     ! 51*8 is HP-UX record number.
120    ENTER @Realpath;Array(*)
           :
```

Accessing Files with Single-Byte Records

With BDAT files, you can define records to be just one byte long (defined records in HP-UX files are always 1 byte long). In this case, it doesn't make sense to read or write one record at a time since even the shortest data type requires two bytes to store a number.

Random access to one-byte records, therefore, has its own set of rules. When you access a one-byte record, the file pointer is positioned to the specified byte. From there, the access proceeds in serial mode. Random OUTPUTs write as many bytes as the data item requires, and random ENTERs read enough bytes to fill the variable.

The example below illustrates how you can read and write randomly to one-byte records.

```
10     INTEGER Int
20     CREATE BDAT "BYTE",100,1
30     ASSIGN @Bytepath TO "BYTE"
40     OUTPUT @Bytepath,1;3.67
50     OUTPUT @Bytepath,9;3
60     OUTPUT @Bytepath,11;"string"
70     ENTER @Bytepath,9;Int
80     ENTER @Bytepath,1;Real
90     ENTER @Bytepath,11;Str$
100    PRINT Real
110    PRINT Int
120    PRINT Str$
130    END
```

```
 3.67
 3
string
```

Note that we had to declare the variable `Int` as an INTEGER. If we hadn't, the system would have given it the default type of REAL and would therefore have required 8 bytes.

Trapping EOF and EOR Conditions

An EOF condition exists whenever the system attempts to read data at, or beyond, the byte marked by the EOF pointers. The EOR condition will arise if you attempt to randomly read or write beyond the particular record specified. If, for example, you try to randomly OUTPUT a 20-character string into a 10-byte record, an EOR condition will occur. EOF conditions will also result whenever you try to read or write beyond the physical end-of-file.

EOF and EOR conditions can be trapped with an ON END statement. ON END is similar to ON ERROR except that it only traps EOF/EOR conditions and is only applicable to the specified I/O path. If you do not have an ON END statement in a program, the EOF/EOR condition will produce an error that is trappable by the ON ERROR statement. Encountering a logical or physical end of file will produce Error 59. Encountering an end of record in random mode produces Error 60.

You can have any number of ON END statements in a program context. ON END statements that refer to different I/O paths will not interfere with each other, even if the paths go to the same file. If you have more than one ON END to the same I/O path, the system will use whichever one it most recently executes during program flow.

An ON END is cancelled by the OFF END statement. OFF END only cancels the ON END branch for the specified I/O path. Re-ASSIGNing an I/O path will also cancel any existing ON END branch for the particular path.

The example below illustrates some of the more common situations that cause an EOF condition.

```
100   CREATE BDAT "ONEND",10,8
110   ASSIGN @Endpath TO "ONEND"
120   ON END @Endpath GOTO Eof1
130   FOR Inc=1 TO 20
140     OUTPUT @Endpath,Inc;SQR(Inc)
150   NEXT Inc
160 Eof1:  !
170   PRINT "EOF: attempt to randomly write beyond physical EOF."
180   PRINT
190   !
200   ON END @Endpath GOTO Eof2
210   OUTPUT @Endpath,5;"THIS IS A STRING."
220 Eof2:  !
230   PRINT "EOR: attempt to randomly write item longer than record."
240   PRINT
250   !
260   ON END @Endpath GOTO Eof3
270   ENTER @Endpath,5;Str$
280 Eof3:  !
290   PRINT "EOR: attempt to read item longer than record."
300   PRINT
310   !
320   ASSIGN @Endpath TO "ONEND"
330   ON END @Endpath GOTO Eof4
340   FOR Inc=1 TO 100
350     OUTPUT @Endpath;"A"
360   NEXT Inc
370 Eof4:  !
380   PRINT "EOF: attempt to serially write beyond physical EOF."
390   PRINT
400   !
410   ASSIGN @Endpath TO "ONEND"
420   ON END @ENDPATH GOTO Eof5
430   FOR Inc=1 TO 100
440     ENTER @Endpath;Str$
450   NEXT Inc
460 Eof5:  !
470   PRINT "EOF: attempt to serially read beyond physical EOF."
480   PRINT
490   !
500   ON END @Endpath GOTO Eof6
510   OUTPUT @Endpath,5;5,END
520   ENTER @Endpath,6;X
530 Eof6:  !
540   PRINT "EOF: attempt to randomly read beyond logical EOF."
550   !
560   END
```

```
EOF: attempt to randomly write beyond physical EOF.

EOR: attempt to randomly write item longer than record.

EOR: attempt to read item longer than record.

EOF: attempt to serially write beyond physical EOF.

EOF: attempt to serially read beyond physical EOF.

EOF: attempt to randomly read beyond logical EOF.
```

This example highlights a number of interesting points. First, in line 210 we try to randomly write a 17-byte string into an 8-byte record. The system returns an EOR condition. The length header for the string, however, is still 17. So when we try to read the string in line 270, we again receive an EOR condition.

In line 320 we re-ASSIGN the I/O path name in order to position the file pointer to byte 1. Then we redefine the ON END branch. These two statements must appear in this order since re-ASSIGNing an I/O path has the effect of canceling any ON END branch previously associated with the path.

In line 510, we shrink the file by moving the EOF pointer to the end of record 5 with the "END" secondary word. When we try to read record 6 in line 520 we get an EOF condition.

Extended Access of Directories

The CAT statement has the following additional capabilities:

- Catalog an individual PROG-type file
- Send the directory to a string array
- Select files to be cataloged by name or by beginning letter(s) of the file name
- Count the number of selected file entries
- Skip a specific number file entries before sending entries to the destination
- Suppress the catalog header
- Use the CRT format when when sending the directory to a string array.
- A listing of only the names of the files in the current working directory of the current default volume.

Cataloging Individual PROG Files

A catalog of a PROG file yields the following information:

- A list of the binary programs contained in the program file and the size of each (in bytes)
- The size of the main program (in bytes).
- A list of contexts (SUB and FN subprograms) and their sizes (in bytes)

The following catalog listing is an example of a CAT performed on an individual PROG file. Note that this catalog format only requires 45 columns.

```
NEWPAGER_A
NAME                    SIZE TYPE
====================== ===== ================
MAIN                   62002 BASIC
FNBar$                  3680 BASIC
FNRoman$                 656 BASIC
Killkeys                 426 BASIC
FNTrim$                  414 BASIC
FNUpc$                   344 BASIC
FNLwc$                   416 BASIC
Table_formatter         6810 BASIC
Strip                   1260 BASIC
  AVAILABLE ENTRIES = 0
```

The **AVAILABLE ENTRIES** table entry is not currently used.

The following listing shows a program which was stored while a BIN[1] program was resident in the computer. If the currently loaded BASIC system version is **different** from the binary program version, a warning and the version codes of both BASIC system and binary program are included in the catalog information. The following example shows the format of the message returned.

```
NEWPAGER_B
NAME                    SIZE TYPE
====================== ===== ================
PHYREC 1.0              1734 BASIC BINARY
*** WARNING:  System level 5.  Bin level 1.
MAIN                   56394 BASIC
FNBar$                  3218 BASIC
FNRoman$                 656 BASIC
Killkeys                 426 BASIC
FNTrim$                  414 BASIC
FNUpc$                   344 BASIC
FNLwc$                   374 BASIC
Table_formatter         7622 BASIC
  AVAILABLE ENTRIES = 0
```

[1] This "BIN" program was different than the "BIN" files currently available with BASIC systems (3.0 and later revisions). This type of binary program was automatically stored with a program when STORE or RE-STORE was executed.

Cataloging to a String Array

The following example program segment shows an example of directing the catalog of mass storage file entries to the CRT and then to a string array.

```
100   PRINT "          CAT to CRT."
110   PRINT "-----------------------------------"
120   CAT TO #CRT;COUNT Files_and_headr  ! Includes 5-line header.
130   PRINT "Number of files=";Files_and_headr-5
140   PRINT
150   !
160   PRINT "     CAT to a string array."
170   PRINT "-----------------------------------"
180   Array_size=Files_and_headr+2  ! Allow for 7-line header.
190   ALLOCATE Catalog$(1:Array_size)[80]
200   CAT TO Catalog$(*)
210   FOR Entry=1 TO Array_size
220     PRINT Catalog$(Entry)
230   NEXT Entry
240   PRINT "Number of files=";Array_size-7
250   PRINT
260   !
270   END
```

The program produces the following output.

```
          CAT to CRT.
-----------------------------------
:INTERNAL
VOLUME LABEL: B9836
FILE NAME PRO TYPE  REC/FILE BYTE/REC   ADDRESS    DATE     TIME
Data1         ASCII        3      256        16  12-Jan-87  12:30
Chap1         BDAT         3      256        20  13-Jan-87   8:00
Prog1         PROG         2      256        23  14-Jan-87   9:10
Chap2         BDAT         7      256        26  14-Jan-87  10:15
Prog2         PROG         2      256        33  14-Jan-87  12:30
Data2         ASCII        9      256        35   3-Mar-87   6:45
Chap3         BDAT         6      256        45   3-Mar-87   7:15
Data3         ASCII        5      256        51   3-Mar-87   9:00
BCD_INTR      ASCII        3      256        56  25-Mar-87   7:00
BCD_CONFIG    ASCII        9      256        59  25-Mar-87   8:15
BCD_ENT1      ASCII        2      256        68  31-Mar-87   9:12
BCD_OUT1      ASCII        1      256        70   1-Apr-87  10:11
BCD_ENTBIN    ASCII        2      256        71  21-Apr-87  12:11
BCD_ENTFMT    ASCII       10      256        73   9-Jun-87   9:00
Number of files= 14

     CAT to a string array.
 ----------------------------------
:INTERNAL, 4
LABEL:  B9826
FORMAT: LIF
AVAILABLE SPACE:   892
                           SYS  FILE   NUMBER   RECORD     MODIFIED     PUB OPEN
FILE NAME             LEV TYPE  TYPE  RECORDS   LENGTH DATE        TIME ACC STAT
===================== === ==== ===== ======== ======== ================ === ====
Data1                   1      ASCII        3      256 12-Jan-87  12:30 MRW
Chap1                   1 98X6 BDAT         3      256 13-Jan-87   8:00 MRW
Prog1                   1 98X6 PROG         2      256 14-Jan-87   9:10 MRW
Chap2                   1 98X6 BDAT         7      256 14-Jan-87  10:15 MRW
Prog2                   1 98X6 PROG         2      256 14-Jan-87  12:30 MRW
Data2                   1      ASCII        9      256  3-Mar-87   6:45 MRW
Chap3                   1 98X6 BDAT         6      256  3-Mar-87   7:15 MRW
Data3                   1      ASCII        5      256  3-Mar-87   9:00 MRW
BCD_INTR                1      ASCII        3      256 25-Mar-87   7:00 MRW
BCD_CONFIG              1      ASCII        9      256 25-Mar-87   8:15 MRW
BCD_ENT1                1      ASCII        2      256 31-Mar-87   9:12 MRW
BCD_OUT1                1      ASCII        1      256  1-Apr-87  10:11 MRW
BCD_ENTBIN              1      ASCII        2      256 21-Apr-87  12:11 MRW
BCD_ENTFMT              1      ASCII       10      256  9-Jun-87   9:00 MRW
Number of files= 14
```

You may have noticed that the format for catalogs sent to string arrays (the second catalog listing) is different from catalogs sent to the PRINTER IS device. The format for catalogs sent to string arrays is the SRM catalog format, which requires that each array element must be dimensioned to hold at least 80 characters with this type of CAT operation. Again, the header contains 7 lines, not 5 as with catalogs sent to devices.

Getting an "Extended" Catalog of a LIF or HFS Disc

When you are cataloging an HFS or LIF directory to a string array, the catalog format is normally that of an SRM catalog. However, you can also specify an HFS- or LIF-format catalog listing with the following syntax:

```
100   CAT TO String_array$(*); EXTEND
```

For an explanation of the HFS catalog listing, see the *BASIC Language Reference* description of CAT.

Getting a Count of Files Cataloged

Including the keyword COUNT followed by a numeric variable returns the total number of file entries plus header lines to that variable; in the preceding example program, the variable `Files_and_headr` is used:

```
120   CAT TO #CRT;COUNT Files_and_headr ! Includes 5-line header.
```

In the above example, line **180** adds 2 to the variable `Files_and_headr` to compensate for the 7-line header which is sent instead of the usual 5-line header (the next section shows how to suppress the header) and stores the result in `Array_size`. `Array_size` is then used to direct the computer to ALLOCATE just enough space in a string-array variable to hold the directory listing. The program can then search the directory listing for further information, if desired.

If the CAT operation would not have filled the string array, the unused array elements would have been set to the null string (i.e., strings of length 0). If there are more catalog lines than string-array elements, the operation stops when the array is filled. No indication of the "overflow" is reported; the count returned is equal to the number of array elements.

Suppressing the Catalog Header

To suppress the catalog header, use the following syntax:

```
CAT;NO HEADER
CAT TO String_array$(*);NO HEADER
CAT "Prog_2";NO HEADER
```

Using NO HEADER suppresses the 5-line heading of a LIF catalog format or 7-line heading of an SRM or HFS catalog format. The catalog listing of a PROG file would be 4 lines shorter. The first line of each catalog listing contains the first directory entry, the second element contains the second entry, and so forth.

If the COUNT option is included, the count returned is the total number of selected files.

Cataloging Selected Files

The directory entries of files that begin with certain characters can be obtained by using the secondary keyword SELECT. For this example, assume that the directory contains the following entries:

```
:INTERNAL
VOLUME LABEL: B9836
FILE NAME PRO TYPE  REC/FILE BYTE/REC   ADDRESS    DATE     TIME
Data1         ASCII        3      256        16  12-Jan-87  12:30
Chap1         BDAT         3      256        20  13-Jan-87   8:00
Prog1         PROG         2      256        23  14-Jan-87   9:10
Chap2         BDAT         7      256        26  14-Jan-87  10:15
Prog2         PROG         2      256        33  14-Jan-87  12:30
Data2         ASCII        9      256        35   3-Mar-87   6:45
Chap3         BDAT         6      256        45   3-Mar-87   7:15
Data3         ASCII        5      256        51   3-Mar-87   9:00
BCD_INTR      ASCII        3      256        56  25-Mar-87   7:00
BCD_CONFIG    ASCII        9      256        59  25-Mar-87   8:15
BCD_ENT1      ASCII        2      256        68  31-Mar-87   9:12
BCD_OUT1      ASCII        1      256        70   1-Apr-87  10:11
BCD_ENTBIN    ASCII        2      256        71  21-Apr-87  12:11
BCD_ENTFMT    ASCII       10      256        73   9-Jun-87   9:00
Number of files= 14
```

Suppose that you want to catalog only files beginning with the letters "Prog". The following examples show how this may be accomplished. Notice that this is **not** the same operation as getting a catalog of a PROG file.

```
Beginning_chars$="Prog"
CAT;SELECT Beginning_chars$

CAT;SELECT "Prog",COUNT Files_and_headr
```

The directory entries of the three files beginning with the letters "Prog" are sent to the PRINTER IS device. In the second CAT statement above, the variable `Files_and_headr` is filled with the number of selected files found on the current default mass storage device plus 5 or 7 header lines. Both CAT statements above go to the PRINTER IS device (or file).

The following result would be sent to the system printing device.

```
:INTERNAL, 4
VOLUME LABEL: B9836
FILE NAME PRO TYPE  REC/FILE BYTE/REC   ADDRESS    DATE     TIME
Prog1         PROG         2      256        23  12-Jan-87  12:30
Prog2         PROG         2      256        33  13-Jan-87   8:00
Prog3         PROG         2      256       533  14-Jan-87   9:10
```

SELECT may also be used to get the catalog of an individual file entry by selecting the entire file name, as shown in the following statement:

```
CAT;SELECT "Chap3"
```

Note that if any other files begin with the letters "Chap3", they will also be listed.

Getting a Count of Selected Files

It is often desirable to determine the total number of files on a disc or the number that begin with a certain character or group of characters. The COUNT option directs the computer to return the number of selected files in the variable that follows the COUNT keyword.

```
CAT;COUNT Files_and_headr
CAT;SELECT "Data",COUNT Selected_files,NO HEADER
```

The first CAT operation returns a count of all files in the directory plus 5 or 7 header lines, since not including SELECT defaults to "select all files". The second operation returns a count of the specifically selected files. Keep in mind that the number of selected files includes the number of files sent to the destination plus the number of files skipped, if any.

Catalogs sent to external devices in the LIF format have a five-line header; in SRM and HFS formats they have seven-line headers. Catalogs to string arrays are SRM format unless EXTEND is added. Catalogs of individual PROG files have a three-line header and a one-line trailer. If an "overflow" of a string array occurs, the count is set to the number of string-array elements plus the number of files skipped. If no entries are sent to the destination (because the directory is empty, or because no entries were selected, or because all selected entries were skipped), the value returned depends on whether there is a header. If there is no header, then zero (0) is returned. If there is a header, then the value returned is the size of the header plus the number following the skip option (the number requested to be skipped).

If an option is given more than once, only the last instance is used.

Skipping Selected Files

If there are many files that begin with the same characters, it is often useful to be able to skip some of the directory entries so that the catalog is not as long. This may be especially useful when using a drive such as an HP 7912, which has the capability of storing more than 10 000 files.

The following statement shows an example of skipping file entries before sending selected entries to the destination.

```
   CAT;SELECT "BCD",SKIP 5
```

```
:INTERNAL, 4
VOLUME LABEL: B9836
FILE NAME PRO TYPE  REC/FILE BYTE/REC   ADDRESS    DATE     TIME
BCD_ENTFMT    ASCII       10      256        73  13-Jan-87   8:00
```

The first five "selected" files (that begin with the specified characters) are "skipped" (i.e., not sent with the rest of the catalog information).

Including COUNT in the previous CAT operation (as shown below) returns a count of the selected files (plus header lines), not just the catalog lines sent to the destination. Remember that selected files includes all files skipped, if any. In this case, a value of 11 is returned, not 1 (or 6) as might be expected.

```
   CAT·SELECT "BCD",SKIP 5,COUNT Catalog_lines
```

Note that if SKIP is included, the count remains the same (as long as at least one file is cataloged). If the number of files to be skipped equals the number of files selected, COUNT returns a value of zero.

```
   CAT;SELECT "BCD",SKIP 6,COUNT Files_and_headr
```

The following program shows an example of looking at the files in a catalog by viewing a small "window" of files at one time. The technique is useful for decreasing the amount of memory required to hold a catalog listing in a string array.

```
100   ! Declare a small string array (7 elements).
110   DIM Array$(1:7)[80]
120   !
130   ! Send header to the array.
140   CAT TO Array$(*)
150   ! Print header.
160   FOR Element=1 TO 7
170     PRINT Array$(Element)[1,45]
180   NEXT Element
190   !
200   ! Now get 7-line "windows" and print files therein.
210   First_file=1  ! Begin with first file in directory.
220   REPEAT  ! Send file entries to Array$ until last file sent.
230     !
240     ! Send files to Array$; SKIP files already printed;
250     ! return index (with COUNT) of last file sent to Array$.
260     CAT TO Array$(*);SKIP First_file-1,COUNT Last_file,NO HEADER
270     DISP "First file=";First_file;"; Last file=";Last_file
280     !
290     ! Print file entries (no entry printed when Last_file=0).
300     FOR Element=1 TO (Last_file-First_file)+1  ! (6 or less)+1.
310       PRINT Array$(Element)[1,45]
320     NEXT Element
330     !
340     First_file=Last_file+1  ! Point to next "window."
350     !
360   UNTIL Last_file=0  ! Until SKIP >= number of files.
370   !
380   END
```

Using a Printer 8

Using a Printer 8

Sooner or later a program needs to print something. A wide range of printers, supported by BASIC, can be connected to the Series 200/300 computers. This chapter covers the statements commonly used to communicate with external printers, including printers controlled by an SRM spooler.

Printers Supported

The following list shows some of the printers that work with Series 200/300 BASIC:

- HP 2225 ThinkJet(TM) Printer
- HP 2686 LaserJet(TM) Printer
- HP 2601, 2602 Daisy-Wheel Impact Printers
- HP 2671, 2673, 9876 Thermal Printers
- HP 82906, 2932, 2934, 2563, 2565, 2566 Dot-Matrix Impact Printers

Check the *Configuration Reference* for your series of computer for a complete and up-to-date list of all printers supported by Series 200/300 BASIC.

Installing, Configuring, and Verifying Your Printer

Instructions for installing and configuring printers for use with this BASIC system are provided in the *Peripheral Installation Guide.* The documentation that is shipped with your printer may also contain some information about how to install, configure, and use it.

Once your printer is installed, you should verify its operation by using the "Peripheral Verification Utility" (VERIFY) described in the "Verifying and Labeling Peripherals" chapter of *Installing and Maintaining the BASIC System.*

The System Printer

The PRINT statement normally directs text to the screen of the CRT.

```
PRINT "This message goes to the printer."
PRINT 3.14159,2.71828,"String",55
```

Text may be re-directed to an external printer by using the PRINTER IS statement. The PRINTER IS statement is used to change the system printer.

```
PRINTER IS CRT
PRINTER IS 701
```

The default **system printer** is the screen of the CRT.

After the printer is switched on and the computer and printer have been connected via an interface cable, there is only one piece of information needed before printing can begin. The computer needs to know the correct **device selector** for the printer. This is analogous to knowing the correct telephone number before making a call.

Device Selectors

A device selector is a number that uniquely identifies a particular device connected to the computer. When only one device is allowed on a given interface, it is uniquely identified by the **interface select code**. In this case, the device selector is the same as the interface select code.

For example, the internal CRT is the only device at the interface whose select code is 1. To direct the output of PRINT statements to the CRT, use the following statements:

```
PRINTER IS 1
 or
PRINTER IS CRT
```

This statement defines the screen of the CRT to be the system printer. Until changed, the output of PRINT statements will appear on the screen of the CRT.

When more than one device can be connected to an interface, such as the internal HP-IB interface (interface select code 7), the interface select code no longer uniquely identifies the printer. Extra information is required. This extra information is the **primary address**.

Primary Addresses

Each printer has a set of switches, usually located on the back panel, which set the primary address of the printer.

The following photographs show the switch locations on various printers. In addition to the primary address switch segments, there are usually segments that control the printers response to other signals on the HP-IB bus.

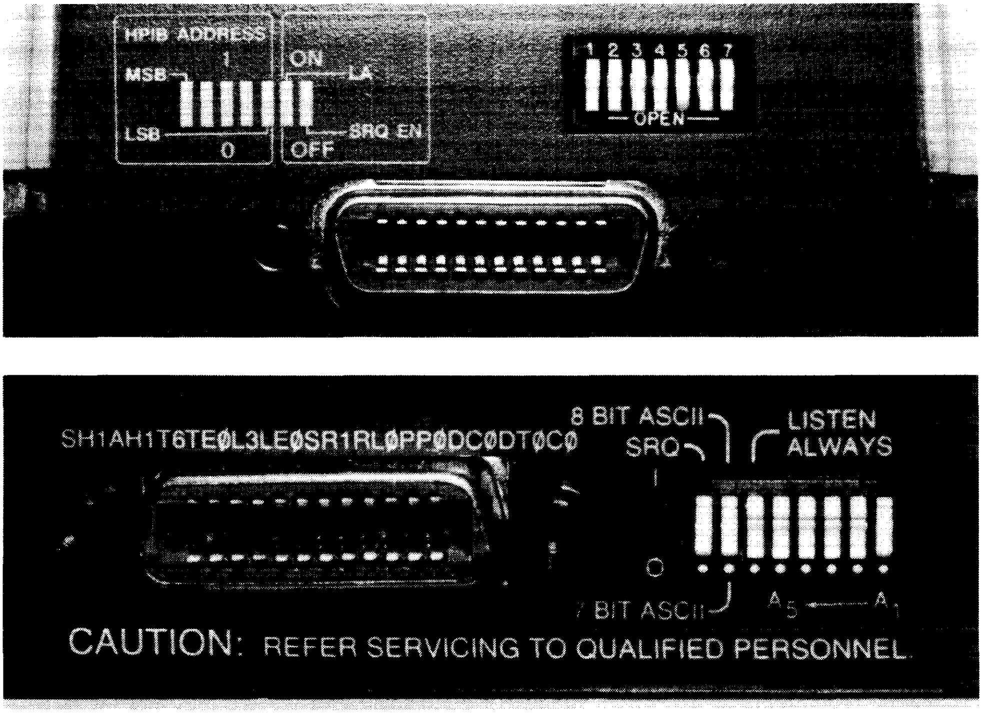


Figure 8-1. Printer Address Switches

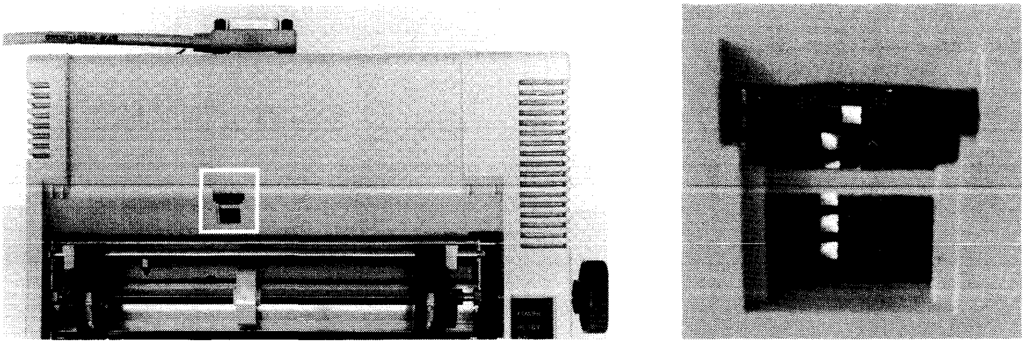


Figure 8-1. Printer Address Switches (Continued)

The primary address, determined by the switch settings, is combined with the interface select code to make up the device selector. In the following example the primary address **01** is appended to the interface select code **7** to produce the device selector **701**.

```
PRINTER IS 701
 or
PRINTER IS PRT
```

This statement tells the computer to use a the internal HP-IB interface (select code 7) to communicate with a printer whose switches are set to the primary address 01. If the printer's primary address is set to 11, the device selector would be 711.

A device selector can be created mathematically by multiplying the interface select code by 100 and adding the primary address. For example, a printer on the internal HP-IB bus whose primary address is set to 9 would have the device selector 709 (7 × 100 + 9 = 709).

To determine your printer's primary address, see the *Peripheral Installation Guide* or your printer's installation manual.

Using Device Selectors

A device selector is used by several different statements. In each of the following, the numeric constant is a device selector.

`PRINTER IS 1`	Specifies the internal CRT (default).
`PRINTER IS 701`	Specifies a printer with interface select code 7 and primary address 01.
`PRINTER IS 22`	Specifies a printer connected to interface select code 22.
`CAT TO #701`	Prints a disc directory at 701.
`PRINTALL IS 707`	Logs information on a printer whose select code is 7 and whose primary address is 07 (binary 00111).
`LIST #701`	Lists the program in memory to a printer connected to the internal HP-IB interface at primary address 01.

Most statements allow a device selector to be assigned to a variable. Either INTEGER or REAL variables may be used.

```
PRINTER IS Hal

CAT TO #Dog
```

The following three-letter mnemonic functions have been assigned useful values.

Table 8-1. Mnemonic Function Values

Mnemonic	Value
PRT	701
KBD	2
CRT	1

For example, the following statements perform the same action:

```
PRINTER IS PRT
PRINTER IS 701
```

The mnemonic may be used anywhere the numeric device selector can be used.

Another method may be used to identify the printer within a program. An I/O path name may be assigned to the printer; the printer is subsequently referenced by the I/O path name. This technique is fully explained in the *BASIC Interfacing Techniques* manual.

Using the External Printer

Most ASCII characters are printed on an external printer just as they appear on the screen of the CRT. Depending on your printer, there will be exceptions. Several printers will also support an alternate character set: either a foreign character set, a graphics character set, or an enhanced character set. If your printer supports an alternate character set, it usually is accessed by sending a special command to the printer.

Control Characters

In addition to a "printable" character set, printers usually respond to control characters. These non-printing characters produce a response from the printer. The following table shows some of the control characters and their effect.

Table 8-2. Printer Control Characters

Printer's Response	Control Character	ASCII Value
ring printer's bell	CTRL-G	7
backspace one character	CTRL-H	8
horizontal tab	CTRL-I	9
line-feed	CTRL-J	10
form-feed	CTRL-L	12
carriage-return	CTRL-M	13

One way to send control characters to the printer is the CHR$ function. Execute the following:

```
PRINTER IS 701
```

```
PRINT CHR$(12)
```

The printer responds with a formfeed. To resume printing on the internal CRT, execute the following:

```
PRINTER IS 1
PRINT "Back to the CRT."
```

Other control characters may be valid for your printer. For example, sending a control-N to the 82905A printer changes the character size (font) of subsequent text.

```
10  Crt=1
20  PRINTER IS 701
30  Big$=CHR$(14)
40  PRINT Big$;"Double-Width Text"
50  PRINTER IS Crt
60  END
```

Refer to the appropriate printer manual for a complete listing of control characters and their effect on your printer. Some control characters will only affect the current line of text.

Escape-Code Sequences

Similar in use to control characters, escape-code sequences allow additional control over most printers. These sequences consist of the escape character, CHR$(27), followed by one or more characters.

For example, the HP 2631 printer is capable of printing characters in several different fonts. To print extended characters on the HP 2631, an escape code sequence is sent to the printer before the actual text to be printed is sent.

```
10  PRINTER IS 701
20  Esc$=CHR$(27)
30  Big$="&k1S"
40  Regular$="&k0S"
50  PRINT Esc$;Big$;"Extended-Font Text"
60  PRINT Esc$;Regular$;"Back to normal."
70  PRINTER IS 1
80  END
```

Many escape code sequences can be used by more than one printer. For instance, the HP 2671 and the HP 2631 share the same escape code sequence for underlining text.

```
10  PRINTER IS PRT
20  Under$=CHR$(27)&"&dD"
30  Normal$=CHR$(27)&"&d@"
40  PRINT "This is not underlined"
50  PRINT Under$&"This is underlined"&Normal$
60  PRINT "Done."
70  PRINTER IS CRT
80  END
```

Since each printer may respond differently to control characters and escape code sequences, check the manual that came with your printer.

Formatted Printing

For many applications the PRINT statement provides adequate formatting. The simplest method of print formatting is by specifying a comma or semicolon between printed items.

When the comma is used to separate items, the printer will print the items on field boundaries. Fields start in column one and occur every ten columns (columns 1,11,21,31,...). Using the following values in a PRINT statement: A=1.1, B=-22.2, C=3E+5, D=5.1E+8.

```
PRINT A,B,C,D
```

Produces:

```
123456789012345678901234567890123456789
 1.1      -22.2      300000    5.1E+8
```

Note the form of numbers in a normal PRINT statement. A positive number has a leading and a trailing space printed with the number. A negative number uses the leading space position for the "–" sign. This is why the positive numbers in the previous example appear to print one column to the right of the field boundaries. The next example shows how this form prevents numeric values from running together.

```
PRINT A;B;C;D
```

```
123456789012345678901234567890123
 1.1 -22.2  300000  5.1E+8
```

Using the semicolon as the separator caused the numbers to be printed as closely together as the "compact" form allows. The compact form always uses one leading space (except when the number is negative) and one trailing space.

The comma and semicolon are often all that is needed to print a simple table. By using the ability of the PRINT statement to print the entire contents of of a array, the comma or semicolon can be used to format the output.

If each array element contained the value of its subscript, the statement:

```
PRINT Array(*);
```

Produces:

```
0  1  2  3  4  5  6  7  8  9  10  11  12  13  14 ...
```

Another method of aligning items is to use the tabbing ability of the PRINT statement.

```
PRINT TAB(25);-1.414
```

```
12345678901234567890123456789012 3
                        -1.414
```

While PRINT TAB works with an external printer, PRINT TABXY may not. PRINT TABXY may be used to specify both the horizontal and vertical position when printing to the internal CRT.

A more powerful formatting technique employs the ability of the PRINT statement to allow an image to specify the format.

Using Images

Just as a mold is used for a casting, an image can be used to format printing. An image specifies how the printed item should appear. The computer then attempts to print to item according to the image.

One way to specify an image is to include it in the PRINT statement. The **image specifier** is enclosed within quotes and consists of one or more **field specifiers**. A semicolon then separates the image from the items to be printed.

```
PRINT USING "D.DDD";PI
```

This statement prints the value of pi (3.141592659...) rounded to three digits to the right of the decimal point.

```
3.142
```

For each character "D" within the image, one digit is to be printed. Whenever the number contains more non-zero digits to the right of the decimal than provided by the field specifier, the last digit is rounded. If more precision is desired, more characters can be used within the image.

```
PRINT USING "D.10D";PI
```

```
3.1415926536
```

Instead of typing ten "D" specifiers, one for each digit, a shorter notation is to specify a repeat factor before each field specifier character. The image "DDDDDD" is the same as the image "6D".

The image specifier can be included in the PRINT statement or on it's own line. When the specifier is on a different line, the PRINT statement accesses the image by either the line number or the line label.

```
100   Format: IMAGE 6Z.DD
110   PRINT USING Format;A,B,C
120   PRINT USING 100;D,E,F
```

Both PRINT statements use the image in line 100.

Numeric Image Specifiers

Several characters may be used within an image to specify the appearance of the printed value.

Table 8-3. Numeric Image Specifiers

Image Specifier	Purpose
D	Replace this specifier with one digit of the number to be printed. If the digit is a leading zero, print a space. if the value is negative, the position may be used by the negative sign.
Z	Same as "D" except that leading zeros are printed.
E	Prints two digits of the exponent after printing the sequence "E+". This specifier is equal to "ESZZ". See the *BASIC Language Reference* for more details.
K	Print the entire number without leading or trailing spaces.
S	Print the sign of the number: either a "+" or "-".
M	Print the sign if the number is negative; if positive, print a space.
.	Print the decimal point.
H	Similar to K, except the number is printed using the European number format (comma radix). (Requires IO)
R	Print the comma (European radix). (Requires IO)
*	Like Z, except that asterisks are printed instead of leading zeros. (Requires IO)

To better understand the operation of the image specifiers examine the following examples and results.

Table 8-4. Examples of Numeric Image Specifiers

Statement	Output
PRINT USING "K";33.666	33.666
PRINT USING "DD.DDD";33.666	33.666
PRINT USING "DDD.DD";33.666	33.67
PRINT USING "ZZZ.DD";33.666	033.67
PRINT USING "ZZZ";.444	000
PRINT USING "ZZZ";.555	001
PRINT USING "SD.3DE";6.023E+23	+6.023E+23
PRINT USING "S3D.3DE";6.023E+23	+602.300E+21
PRINT USING "S5D.3de";6.023E+23	+60230.000E+19
PRINT USING "H";3121.55	3121,55
PRINT USING "DDRDD";19.95	19,95
PRINT USING "***";.555	**1

To specify multiple fields within the image, the field specifiers are separated by commas.

Table 8-5. Multiple-Field Numeric Image Specifiers

Statement	Output
PRINT USING "K,5D,5D";100,200,300	100 200 300
PRINT USING "DD,ZZ,DD";1,2,3	102 3

If the items to be printed can use the same image, the image need be listed only once. The image will then be re-used for the subsequent items.

```
PRINT USING "5D.DD";3.98,5.95,27.50,139.95

123456789012345678901234567890123
    3.98    5.95   27.50  139.95
```

The image is re-used for each value. An error will result if the number cannot be accurately printed by the field specifier.

String Image Specifiers

Similar to the numeric field image characters, several characters are provided for the formatting of strings.

Table 8-6. String Image Specifiers

Image Specifier	Purpose
A	Print one character of the string. If all characters of the string have been printed, print a trailing blank.
K	Print the entire string without leading or trailing blanks.
X	Print a space.
”literal”	Print the characters between the quotes.

The following examples show various ways to use string specifiers.

```
PRINT USING "5X,10A,2X,10A";"Tom","Smith"

12345678901234567890123456789
     Tom         Smith

PRINT USING "5X,""John"",2X,10A";"Smith"

12345678901234567890123456789
     John  Smith

PRINT USING """PART NUMBER"",2X,10D";90001234

12345678901234567890123456789
PART NUMBER    90001234
```

Additional Image Specifiers

The following image specifiers serve a special purpose.

Table 8-7. Additional Image Specifiers

Image Specifier	Purpose
B	Print the corresponding ASCII character. This is similar to the CHR$ function.
#	Suppress automatic end-of-line (EOL) sequence.
L	Send the current end-of-line (EOL) sequence; with IO, see the PRINTER IS statement in the *BASIC Language Reference* for details on re-defining the EOL sequence.
/	Send a carriage-return and a linefeed.
@	Send a formfeed.
+	Send a carriage-return as the EOL sequence. (Requires IO)
−	Send a linefeed as the EOL sequence. (Requires IO)

For example:

`PRINT USING "@,#"` outputs a formfeed.

```
PRINT USING "D,X,3A,""OR NOT"",X,B,X,B,B";2,"BE",50,66,69
```

Special Considerations

If nothing prints, check if the printer is ON LINE. When the printer if OFF LINE the computer and printer can communicate but no printing will occur.

Sending text to a non-existent printer will cause the computer to wait indefinitely for the printer to respond. ON TIMEOUT may be used within a program to test for the printer. To clear the error press CLR I/O or Break, check the interface cable and switch settings, then try again.

Since the printer's device selector may change, keep track of the locations in the program where a device selector is used. If most of the program's output is sent to a printer, you may wish to use the PRINTER IS statement at the beginning of the program and then send messages to the CRT screen by using the OUTPUT statement.

```
PRINTER IS 701
PRINT "Text to the printer."
OUTPUT 1;"Screen Message"
PRINT "Back to the printer."
```

If the program must use the PRINTER IS statement frequently, assign the device selector to a variable; then if the device selector changes, only one program line will need to be changed.

Using SRM Printers through the Spooler

The SRM system not only provides shared access to printers and plotters, but also manages their use so that workstations never need to wait for output to be generated.

To use shared printers, you place files to be printed into a special directory where they are held until the printer is free. The system keeps track of the order in which files arrive from the workstations, and outputs them in the same order. This method is called "spooling," and the directory where the files are kept is called the "spooler directory." Spooler directories are created for the SRM controller's use when the shared peripherals are installed on the SRM system.

After a file is placed in a spooler directory, the workstation is free to do other processing.

Using a Spooler

Use of special SRM directories called "spooler directories" allows you to access a shared printer or plotter. Setting up a spooler directory is explained in the "Interfaces and Peripherals" chapter of the *SRM System Manager's Guide*. The examples in this section assume that the spooler directories `LP` (for "Line Printer") and `PL` (for "PLotter") have been created at the root of the SRM directory structure.

Spooling Using PRINTER IS

You can use the PRINTER IS statement to direct data to your shared printer. The following command sequence illustrates this spooling method:

```
CREATE BDAT "/LP/Print_file",1
PRINTER IS "/LP/Print_file"
LIST
XREF
PRINTER IS CRT
```

PRINTER IS works only with BDAT files. Because the SRM 1.0 operating system's spooling works only with ASCII files, you cannot use PRINTER IS for spooling with that version of SRM.

Note

The DUMP DEVICE IS and PRINTALL IS statements do **not** support directing data to files as shown above, so cannot be used for printer spooling.

Writing Files to the Spooler Directories

You may also access the printer associated with `LP` by placing the data to be printed in an ASCII or BDAT file in that spooler directory. For example, to list a program currently in memory, you could SAVE the program in `LP` as the file `P1_LISTING` by typing either:

```
SAVE "LP/P1_LISTING:REMOTE"   [Return] or [ENTER]
```

or

```
SAVE "/LP/P1_LISTING"   [Return] or [ENTER]
```

The SAVE statement creates an ASCII file. Although this is the same syntax used to save programs on a shared disc, the SRM system knows that `LP` is a spooler directory and prints the file as soon as possible.

Note

When used for spooling, SAVE places a file in the spooler directory. The file is printed, then purged. You may wish to save or create the file first, then use the COPY statement to place the file into the spooler directory.

Sending Program Output to a Shared Printer

To spool program output to a shared printer, create an ASCII or BDAT file, assign an I/O path name to the file (which opens the file), and OUTPUT the data to that file. With BDAT files, you should ASSIGN with FORMAT ON. When the file's contents are to be printed, close the file. The following example program segment outputs the data stored in the string array called `Data$` to an ASCII file named `PERFORMANCE`.

```
760   CREATE ASCII "/LP/PERFORMANCE",100
770   ASSIGN @Spool TO "/LP/PERFORMANCE"
780   OUTPUT @Spool;"Performance Summary"
790   OUTPUT @Spool;Data$(*)
800   ASSIGN @Spool TO *   ! Initiate printing.
```

The system waits until the file is non-empty and closed before sending its contents to the output device. If your file is not printed or plotted within a reasonable amount of time, you may not have closed it. You can verify that your file is ready to be printed or plotted by cataloging the spooler directory:

```
CAT "/LP" [Return] or [ENTER]
```

The open status (`OPEN STAT`) of the file currently being printed or plotted is listed as locked (`LOCK`). Files currently being written to the spooler directory (either printer or plotter) are listed as `OPEN`. Files that do not have a status word in the catalog are ready for printing or plotting.

Version 2.0 of the SRM operating system (and later versions) allow BDAT files to be sent to the printing device as a byte stream. (With SRM version 1.0, only ASCII files can be used.)

Note

With the SRM 2.0 operating system, a BDAT file sent to the spooler is printed exactly as the byte stream sent. Unless you set up the BDAT file correctly, improper printer output or operation could result. Therefore, you should ASSIGN BDAT files with FORMAT ON before outputting data.

The spooler prints each string and numeric item on a separate line by inserting a carriage return and line feed after each item. To put several strings on one line, concatenate them into one string before using OUTPUT to send them to the spooler file. You may insert ASCII control characters in the data by using the CHR$ string function.

Appearance of Output

Printed output for each file includes a one-page header, which identifies the directory path to the file, the file's name, and the date and time of the printing.

To cause the printer to skip the paper perforation after printing a page (60 lines), prefix your file name with "FF". For example:

```
SAVE "/LP/FF_MYTEXT" [Return] or [ENTER]
```

The Real-Time Clock 9

The Real-Time Clock 9

All Series 200 and 300 computers have a real-time clock that you can set and read to monitor the time of day and date. In addition, all Series 300 computers (and some Series 200 computers) have a battery-backed, non-volatile clock that keeps time even when the power is removed from the computer. This chapter describes using the clock and related functions and statements.

Note

Many of the statements described in this chapter require the CLOCK binary. Check the *BASIC Language Reference* for specific requirements of each statement.

Initial Clock Value

When you initially boot the BASIC system, the real-time clock is set to one of these values:

- With Series 300 computers, the clock value is read from the non-volatile clock and placed into the real-time (volatile) clock.
- With Series 200 computers which have the 98270 Powerfail Option installed, the real-time (volatile) clock time is set to the value of the non-volatile clock.
- With computers on the Shared Resource Management (SRM) system that *don't* have a non-volatile clock, the clock value is taken from the SRM system. (This occurs when the SRM and DCOMM binaries are loaded.)
- If the computer does not have a non-volatile clock (and is not connected to an SRM system), the time is set to 12:00:00 a.m. (midnight), March 1, 1900.

Do You Have a Non-Volatile Clock?

There is a status register that you can interrogate to determine whether or not you have a non-volatile clock in your computer. The following program shows how to use this register.

```
100  STATUS 32,4;Clock_type
110  SELECT Clock_type
120  CASE 0
130    DISP "No battery-backed clock."
140    !
150  CASE 1
160    DISP "Series 200 (98270) battery-backed clock."
170    !
180  CASE 2
190    DISP "Series 300 (HP-HIL) battery-backed clock."
200    !
210  END SELECT
220  END
```

If you don't have a non-volatile clock, you will need to determine whether or not the real-time clock is set to the proper time. Subsequent sections describe how to do this.

Clock Range and Accuracy

The range of Series 200 volatile and non-volatile clocks is March 1, 1900 through August 4, 2079. The Series 300 volatile and non-volatile clocks both have a lower limit of March 1, 1900. However, the upper limit of the volatile clock is August 4, 2079, while the upper limit of the non-volatile clock is February 29, 2000.

The volatile real-time clocks provide an accuracy of ±5seconds per day. The Series 200 battery-backed "powerfail" (98270) clock maintains time to within ±2.5 seconds per day. The Series 300 battery-backed clock maintains time to within ±5 seconds per day.

Reading the Clock

Internally, the clock maintains the year, month, day, hour, minute, and second as a single real number. This number is scaled to an arbitrary "dawn of time," thus allowing it to also represent the Julian date. The current value of the clock is returned by the TIMEDATE function.

```
PRINT TIMEDATE
```

While the value returned contains all the information necessary to uniquely specify the date and time to the nearest one-hundredth of a second, it needs to be "unpacked" to provide understandable information.

Determining the Date and Time of Day

The following functions are available to extract the date and time of day from TIMEDATE.

The DATE$ function extracts the date from the value of TIMEDATE.

```
PRINT DATE$(TIMEDATE)
```

Prints: `1 Mar 1900`

This is the default power-up date for machines without the battery-backed real-time clock.

The TIME$ function returns the time of day.

```
PRINT TIME$(TIMEDATE)
```

Prints: `00:05:27`

Setting the Clock

TIMEZONE and HP-UX Clock Compatibility

If you are sharing an HFS disc with an HP-UX system, you will need to use the TIMEZONE IS statement before setting the clock. This statement specifies the offset from Greenwich Mean Time, providing compatibility with HP-UX time stamps on files when switching back and forth between the BASIC and HP-UX operating systems. See the *BASIC Language Reference* entry TIMEZONE IS for details.

The SET TIMEDATE statement is used to set the clock.

```
SET TIMEDATE DATE("1 OCT 1987") + TIME("8:37:30")
```

The time of day can be changed without affecting the date by the SET TIME statement.

```
SET TIME TIME("9:55")
```

Note that an error is reported if you try to set the clock to a value outside the range stated on the preceding page.

Clock Time Format

To minimize the space required to store the date and time, and yet insure a unique value for each point in time, both time and date are combined as a single real number. This value is the Julian date multiplied by the number of seconds in a day. By recalling that there are 86400 seconds in a day, the date and time of day can be extracted from TIMEDATE by the following simple algorithms.

`TIMEDATE MOD 86400` returns the time of day, and

`TIMEDATE DIV 86400` returns the Julian date.

The time of day is expressed in seconds past midnight and is easily divided into hours, minutes, and seconds. The Julian date requires a bit more processing to extract the month, day, and year but this method insures a unique value for each day over the entire range of the clock.

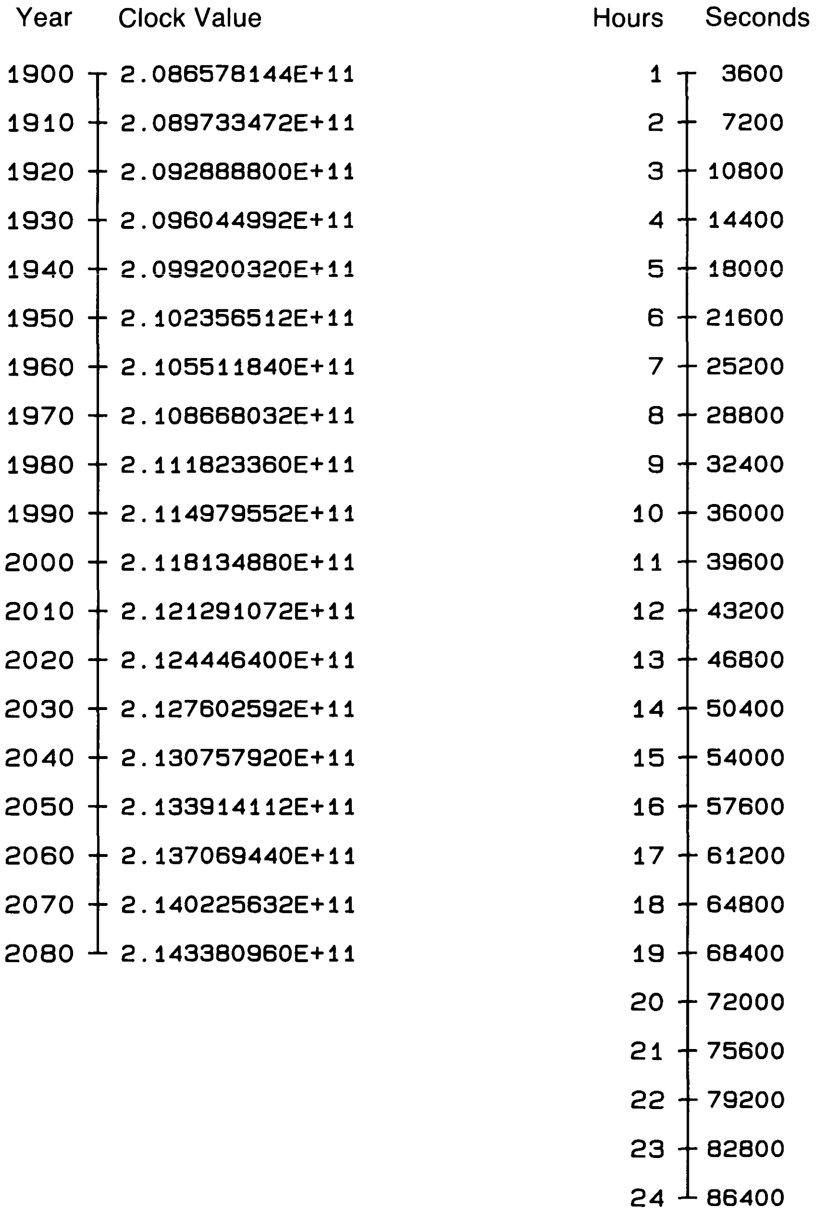


Figure 9-1. Clock Time

Setting Only the Time

The time of day is changed by SET TIME X, where X is the number of seconds past midnight. The value of X must be in the range: 0 through 86399.99 seconds. The TIME function will convert twenty-four hour formatted time (HH:MM:SS) into the value needed to set the clock.

The TIME function converts an ASCII string representing a time of day, in twenty-four hour format, into the number of seconds past midnight. For example:

```
SET TIME TIME("15:30:10")
```

Is equivalent to:

```
SET TIME 55810
```

Either of these statements will set the time of day without changing the date. Use the SET TIMEDATE statement to change the date.

To display the new time, the TIME$ function formats the clock's value (TIMEDATE) into hours, minutes, and seconds.

```
PRINT TIME$(TIMEDATE)
```

Prints: `15:30:16`

Even though TIMEDATE returns a value containing both time of day and the Julian date, TIME$ performs an internal modulo 86400 on the value passed to the function and will always return a string in the range: `00:00:00` thru `23:59:59`.

The following program contains the routines to set and display the time of day. The routines are written as user-defined functions that may be appended to your program. Once appended to a program, the routines duplicate the TIME and TIME$ functions available with CLOCK. The formatted time can then be displayed by the following statement.

```
PRINT FNTime$(TIMEDATE)
```

Prints: `15:31:05`

Given the clock's value, the **FNTime$** function returns the time of day in 24 hour format (HH:MM:SS). The **FNTime** function converts the time of day to seconds and is used to set the clock.

```
10 Show_time:DISP FNTime$(TIMEDATE)
20    GOTO Show_time
30    END
40    !
50    ! While the program is running, type:
60    ! SET TIME FNTIME("11:5:30")
70    ! then press <EXECUTE> to show the new time.
80    !
90    !*********************************************************
100   !
110   DEF FNTime$(Now) ! Given 'SECONDS' Return 'hh:mm:ss'
120     !
130     Now=INT(Now) MOD 86400
140     H=Now DIV 3600
150     M=Now MOD 3600 DIV 60
160     S=Now MOD 60
170     OUTPUT T$ USING "#,ZZ,K";H,":",M,":",S
180     RETURN T$
190   FNEND
200   !
210   DEF FNTime(T$) ! Given 'hh:mm:ss' Return 'SECONDS'
220     !
230     ON ERROR GOTO Err
240     ENTER T$;H,M,S
250     RETURN (3600*H+60*M+S) MOD 86400
260 Err:OFF ERROR
270     RETURN TIMEDATE MOD 86400
280   FNEND
```

After entering the program, follow the instructions at the beginning of the program to verify correct operation. Store this program in a file named "FUNTIME". The functions can be extracted from this program and appended to other programs by the LOADSUB statement.

Note that the **FNTime** function requires hours, minutes, and seconds, while the TIME function only requires hours and minutes.

Setting Only the Date

The date is changed by SET TIMEDATE X, where X is the Julian date multiplied by the number of seconds in a day (86400). The DATE function converts a formatted date (DD MMM YYYY) into the value needed to set the clock. Due to the wide range of values allowed by the DATE function, negative years can be specified, but not when using the function to set the clock.

The following statement will set the clock to the proper date.

```
SET TIMEDATE DATE("1 Jun 1984")
```

When programming without CLOCK, the user-defined function `FNDate` can be used.

```
SET TIMEDATE FNDate("1 Jun 1984")
```

Both of these statements are equivalent to the following statement.

```
SET TIMEDATE 2.113216992E+11
```

The DATE and `FNDate` functions convert the accompanying string (or string expression) into the numeric value needed to set the clock. To read the clock, the DATE$ and `FNDate$` functions format the clock's value as the day, month, and year. For example, the following line will print the date.

```
PRINT DATE$(TIMEDATE)
```

Prints: `1 Jun 1984`

Programs that need to run without CLOCK can use the following user-defined functions appended to the end of the program. These functions simulate the DATE and DATE$ keywords available in CLOCK. The algorithm is valid over the entire range of the clock.

Note the following functions are restricted to values the clock will accept, the DATE and DATE$ functions available with CLOCK allow a much wider range of values (including negative years).

```
10 Show_date:   DISP FNDate$(TIMEDATE)
20              GOTO Show_date
30              END
40    !
50    ! While the program is running, type:
60    ! SET TIMEDATE FNDATE("1 JAN 82")   <EXECUTE>
70    !
80    !**********************************************************
90    !
100   DEF FNDate$(Seconds) ! Given 'SECONDS' Return 'dd mmm yyyy'
110     !
120     DATA JAN,FEB,MAR,APR,MAY,JUN,JUL,AUG,SEP,OCT,NOV,DEC
130     DIM Month$(1:12)[3]
140     READ Month$(*)
150     !
160     Julian=Seconds DIV 86400-1721119
170     Year=(4*Julian-1) DIV 146097
180     Julian=(4*Julian-1) MOD 146097
190     Day=Julian DIV 4
200     Julian=(4*Day+3) DIV 1461
210     Day=(4*Day+3) MOD 1461
220     Day=(Day+4) DIV 4
230     Month=(5*Day-3) DIV 153     ! Month
240     Day=(5*Day-3) MOD 153
250     Day=(Day+5) DIV 5           ! Day
260     Year=100*Year+Julian        ! Year
270     IF Month<10 THEN
280       Month=Month+3
290     ELSE
300       Month=Month-9
310       Year=Year+1
320     END IF
330     OUTPUT D$ USING "#,ZZ,X,3A,X,4Z";Day,Month$(Month),Year
340     RETURN D$
350   FNEND
360   !
370   DEF FNDate(Dmy$) ! Given 'dd mmm yyyy' Return 'SECONDS'
380     !
390     DATA JAN,FEB,MAR,APR,MAY,JUN,JUL,AUG,SEP,OCT,NOV,DEC
400     DIM Month$(1:12)[3]
410     READ Month$(*)
420     !
430     ON ERROR GOTO Err
440     I$=Dmy$&"    "
450     ENTER I$ USING "DD,4A,5D";Day,M$,Year
460     IF Year<100 THEN Year=Year+1900
470     FOR I=1 TO 12
480     IF POS(M$,Month$(I)) THEN Month=I
490     NEXT I
```

```
500     IF Month=0 THEN Err
510     IF Month>2 THEN
520       Month=Month-3
530     ELSE
540       Month=Month+9
550       Year=Year-1
560     END IF
570     Century=Year DIV 100
580     Remainder=Year MOD 100
590     Julian=146097*Century DIV 4+1461*Remainder DIV 4+(153*Month+2) DIV
5+Day+1721119
600     Julian=Julian*86400
610     IF Julian<2.08662912E+11 OR Julian>=2.143252224E+11 THEN Err
620     RETURN Julian ! Return Julian date in SECONDS
630 Err:OFF ERROR      ! ERROR in input.
640     RETURN TIMEDATE ! Return current date.
650   FNEND
```

Store the program in a file named "FUNDATE". Later the functions can be appended to other programs by the LOADSUB statement.

The functions `FNDate$` and `FNDate` format the date as "DD MMM YYYY", where DD is the day of the month, MMM is the first three letters of the month, and YYYY is the year. The function `FNDate` will accept the last two digits of the year. See line 460. Note that the `FNDate` function requires two digits for the day, while the DATE function does not.

Different formats require only slight modification. By changing the following lines, the date is formatted as "MM/DD/YYYY".

```
330 OUTPUT D$ USING "#,2D,A,2D,A,2D";Month;"/";Day;"/";Year
```

```
450 ENTER I$ USING "#,ZZ,K";Month;Day;Year
```

European date format is obtained by swapping the month and day in the above statements. When changing the format, be sure to switch both functions.

If the all numeric format is chosen, delete the three lines in each function that load the array with the month mnemonics.

Using Clock Functions and Example Programs

The following statements summarize setting and displaying the clock.

```
SET TIMEDATE FNDate("12 DEC 1981") + FNTime("13:44:15")

SET TIME FNTime("8:30:00")

PRINT FNTime$(TIMEDATE)

DISP FNDate$(TIMEDATE)
```

It is important to note that `SET TIMEDATE` expects a date and time while the DATE function and the user-defined function `FNDate` return only a date. This effectively sets the clock to midnight of the date specified.

To keep the functions short, minimal parameter checking is performed. Additional checking may be incorporated within the functions or within the calling context. If `FNDate` or `FNTime` cannot correctly decode the input, the current value of the clock is returned.

The date and time functions can be used with the following program shell to provide a "friendly" interface to the clock.

```
10    ! PROGRAM SHELL FOR SETTING TIME AND DATE.
20    !
30    ! REQUIRES THE TIME AND DATE FUNCTIONS.
40    !
50    DIM Day$(0:6)[9]
60    DATA Monday,Tuesday,Wednesday,Thursday,Friday,Saturday,Sunday
70    READ Day$(*)
80    !
90    ON ERROR GOTO Nofun      ! Test if functions
100   Dmy$=FNDate$(TIMEDATE)   ! have been loaded
110   Hms$=FNTime$(TIMEDATE)
120   OFF ERROR
130 Main:                      ! Get NEW date
140   CLEAR SCREEN
150   F$=CHR$(255)&CHR$(72)
160   !
170   PRINT TABXY(1,14);"Enter the date, and press CONTINUE."
180   OUTPUT 2 USING "#,11A,2A";Dmy$,F$
190   !
200   INPUT Dmy$               ! WAIT for INPUT
210   !
220   ENTER Dmy$ USING "2D,4A,5D";D,M$,Y
230   CLEAR SCREEN
```

```
240    !
250    PRINT TABXY(1,14);"Enter the time of day and press CONTINUE"
260    OUTPUT 2 USING "#,11A,2A";Hms$,F$
270    INPUT Hms$
280    ENTER Dmy$ USING "2D,4A,5D";D,M$,Y
290    !
300    SET TIMEDATE FNDate(Dmy$)+FNTime(Hms$)
310    !
320    CLEAR SCREEN
330    W=(TIMEDATE DIV 86400) MOD 7 ! Day of week
340    PRINT TABXY(1,1);"The clock has been set to:"
350    PRINT TABXY(1,3);Day$(W);" ";Dmy$;"  ";FNTime$(TIMEDATE)
360    GOTO Quit
370    !
380    ! ************ SUBROUTINES ************
390    !
400 Nofun:PRINT "The TIME & DATE FUNCTIONS must be appended,"
410    PRINT "(via LOADSUB) before program will work."
420 Quit:END
430    !
440    ! ************* FUNCTIONS **************
450    !
460    ! append time and date functions here
```

The program tests to see if the functions have been loaded by trying to use them. If they are not loaded the program ends with an error message. With the CLOCK binary, this program can still be used. Replace the calls to the user-defined functions with the appropriate keywords. The error trapping can then be deleted.

To append the user-defined functions, execute the following statements while the demonstration program is in memory.

```
LOADSUB ALL FROM "FUNDATE"
LOADSUB ALL FROM "FUNTIME"
```

Examine the program to be sure the functions have been loaded.

The program will prompt for the date and time, then set the clock accordingly. A program such as this may be used as the system start up program for applications requiring the date or time.

Day of the Week

An advantage of Julian dates is the simplicity of finding the day of the week. `TIMEDATE DIV 86400 MOD 7` returns a number which represents the day of the week. Monday is represented by zero (0), and the numbering continues through the week to Sunday which is represented by six (6). See the previous program for an example of using this routine.

Days Between Two Dates

The number of days between two dates is easily calculated as the following program demonstrates.

```
10     ! Days between two dates
20     INPUT "ENTER THE FIRST DATE (DD MMM YYYY)",D1$
30     INPUT "ENTER THE SECOND DATE (DD MMM YYYY)",D2$
40     Days=(DATE(D2$)-DATE(D1$)) DIV 86400
50     DISP Days;"days between '";D1$;"' and '";D2$;"'"
60     END
```

Interval Timing

Timing a single event of short duration is quite simple.

```
10     T0=TIMEDATE          ! Start
20     FOR J=1 TO 5555
30     !
40     NEXT J
50     T1=TIMEDATE          ! Finish
60     !
70     PRINT "It took";DROUND(T1-T0,3);"seconds"
80     END
```

Programs can and should be written so that they do not change the setting of the clock. A short program, which simulates a stopwatch, allows interval timing without changing the clock.

```
10     ! Program: STOPWATCH
20     ! Interval timing without changing the clock
30     ON KEY 5 LABEL " START " GOTO Start
40     ON KEY 6 LABEL " STOP  " GOTO Hold
50     ON KEY 7 LABEL " RESET " GOTO Reset
60     ON KEY 8 LABEL "  LAP  " GOSUB Lap
70     !
80 Reset:PRINT CHR$(12)             ! form-feed
90     H=0                          ! Set all
100    M=0                          !   to
110    S=0                          !  zero.
120    !
130 Hold:DISP TAB(9);H;":";M;":";S  ! Wait til
```

```
140   GOTO Hold                         !  keypress
150   !
160 Lap:PRINT H;":";M;":";S             ! Print lap
170   RETURN
180   !
190 Start:Z=3600*H+60*M+S-TIMEDATE      ! Elapsed-
200 Loop:T=(TIMEDATE+Z) MOD 86400       !     time
210   T=INT(T*100)/100                  ! .01 sec.
220   H=T DIV 3600                      ! Hours
230   M=T MOD 3600 DIV 60               ! Minutes
240   S=T MOD 60                        ! Seconds
250   DISP TAB(9);H;":";M;":";S         ! Show time
260   GOTO Loop                         ! Do again
270   END
```

Branching on Clock Events

Several additional branching statements, available with CLOCK, allow end-of-statement branches to be triggered according to the real-time clock's value.

- ON TIME enables a branch to be taken when the clock reaches a specified time of day.
- ON DELAY enables a branch to be taken after a specified number of seconds has elapsed.
- ON CYCLE enables a recurring branch to be taken with each passage of a specified number of seconds.

The specified time can range from 0.01 thru 167772.15 seconds for the ON CYCLE and ON DELAY statements and 0 thru 86399.99 seconds for ON TIME. The value specified with ON TIME indicates the time of day (in seconds past midnight) for the branch to occur.

Each of these statements has a corresponding statement to cancel the branch (OFF TIME, OFF DELAY, and OFF CYCLE). A statement is also canceled by executing another ON TIME, ON DELAY, or ON CYCLE statement.

All of the statements use the internal real-time clock. Care should be taken to avoid writing programs that could change the clock's setting during execution. Since only one resource is dedicated to each statement, certain restrictions apply to the use of these statements.

Cycles and Delays

Both the ON CYCLE and ON DELAY statements enable a branch to be taken as soon as the specified number of seconds has elapsed. ON CYCLE remains in effect, re-enabling a branch with each passage of time. For example:

```
10     ON CYCLE 1 GOSUB Five   ! Print 5 random numbers every second.
20     ON DELAY 6 GOTO Quit    ! After 6 seconds quit.
30     !
40 T:  DISP TIME$(TIMEDATE)    ! Show the time.
50     GOTO T
60     !
70 Five:FOR I=1 TO 5
80       PRINT RND;
90     NEXT I
100    PRINT
110    RETURN
120    !
130 Quit:END
```

The program will print five random numbers every second for six seconds and then stop.

Only one ON CYCLE and one ON DELAY statement can be active in a program context. Executing a second ON CYCLE or ON DELAY statement in the same program context deactivates the first ON CYCLE or ON DELAY statement. If a branch is missed, due to priority restrictions or execution of a subprogram, the event is logged and the branch will be taken when the restriction is removed or the original context is restored. If an active ON CYCLE or ON DELAY statement gets canceled in an alternate context (subprogram) the branch is restored when execution returns to the defining context. (See Branching Restrictions for more information about this).

Time of Day

The ON TIME statement allows you to define and enable a branch to be taken when the clock reaches a specified time of day, where time of day is expressed as seconds past midnight. Using the TIME function simplifies setting an ON TIME statement by allowing a formatted time of day to be used. For example:

```
ON TIME TIME("11:30") GOTO Lunch
```

Typically, the ON TIME statement is used to cause a branch at a specified time of day. By adding an offset to the current clock value, the ON TIME statement can be used as an interval timer. In the following example, both ON DELAY and ON TIME are used as interval timers.

```
10    ON DELAY 5 GOSUB Takeoff                        ! delay 5 seconds
20    ON TIME (TIMEDATE+10) MOD 86400 GOSUB Touchdown ! delay 10 seconds
30    PRINT "STARTING... ",TIME$(TIMEDATE)
40 Clock:DISP TIME$(TIMEDATE)
50    GOTO Clock
60    !
70 Takeoff:PRINT "TAKEOFF at ",TIME$(TIMEDATE)
80    RETURN
90 Touchdown:PRINT "TOUCHDOWN at ",TIME$(TIMEDATE)
100   RETURN
110   END
```

The starting time is printed when the program is executed. Five seconds later the first subroutine is executed. Ten seconds after the program starts, the second subroutine is executed.

Only one ON TIME statement can be active in a program context. If a branch is missed, due to priority restrictions or execution of a subprogram, the event is logged and the branch will be taken when the restriction is removed or the original context is restored. If an active ON TIME statement gets canceled in an alternate context (subprogram) the branch is restored when execution returns to the defining context. (See Branching Restrictions for more information about this).

Due to the "match an exact time" nature of the ON TIME statement, if the specified time occurs when the statement is temporarily canceled (by an OFF TIME in an alternate context), no branch will be taken when the defining context is restored.

Priority Restrictions

A priority can be assigned to the branch defined by ON CYCLE, ON DELAY, and ON TIME. For example:

```
ON CYCLE Seconds,Priority GOTO Label
```

If the system priority is higher than the branch priority at the time specified for the branch, the event will be logged but the branch will not be taken until the system priority falls below the branch priority. An example program follows.

```
10    COM Start
20    P=0
30 Up:P=P+1
40    IF P>15 THEN Quit          ! Priority from 1 thru 15
50    PRINT
60    PRINT "Priority:";P;
70    Start=TIMEDATE             ! Save the start-time for subprogram.
80    ON CYCLE 1,P RECOVER Up    ! New priority every second if not Busy.
90    ON DELAY .5,6 CALL Busy    ! DELAY overrides CYCLE until priority
100                              !                  (P) is greater than 6.
110 W:GOTO W
120 Quit:END
130   !----------------- SUB has priority of 6 ---------------------
140   SUB Busy
150     COM Start
160     PRINT "SUB";
170     WHILE I<10
180       IF TIMEDATE>Start+1 THEN  ! Has ON CYCLE time been exceeded?
190         PRINT "*";              ! YES (only prints if Priority<7)
200       ELSE
210         PRINT ".";              ! NO
220       END IF
230       I=I+1                     ! Loop ten times
240       WAIT .1
250     END WHILE
260     PRINT "DONE";
270   SUBEND
```

Once the priority assigned to the ON CYCLE statement is greater than the priority assigned to the ON DELAY statement (6), the subprogram will be interrupted. After running the program, change line 80 in the above program to the following:

```
80    ON CYCLE 1,P GOTO Up
```

Running the new version of the program will show that GOTO (or GOSUB) will not interrupt a subprogram regardless of the assigned priority. The branch will be logged but not taken until execution returns to the main program. If you write a program that makes extensive use of subprograms and branching statements, use CALL and RECOVER to insure proper operation.

Branching Restrictions

Certain restrictions apply to the use of ON TIME, ON CYCLE, and ON DELAY because only one resource is dedicated to each statement. Assuming an active branch has been defined in the main program, execution of a subprogram which sets up a new branch will cause the loss of the original time. When the main program context is restored, the original branch will be restored, but at the time defined in the subprogram. The following program will illustrate this effect.

```
10    COM Counter
20    Counter=0
30    GINIT
40    GRID 1,1              ! Fill graphics raster with grid.
50    DISP Counter
60    ON CYCLE 2 CALL Flash  ! Flash graphics every 2 seconds.
70 W: GOTO W
80    END
90    !----------------- SUB to flash graphics raster --------------
100   SUB Flash
110     COM Counter
120     GRAPHICS ON
130     Counter=Counter+1
140     DISP Counter
150     IF Counter=5 THEN          ! Change CYCLE value during fifth CALL.
160       ON CYCLE .1,2 CALL Quit ! New value (.1) will replace old (2).
170                               ! Flash will end before Quit gets called.
180     END IF
190     GRAPHICS OFF
200   SUBEND
210   !----------------- SUB that won't get called -----------------
220   SUB Quit
230     PRINT "PROGRAM HAS STOPPED"
240     STOP
250   SUBEND
```

The program starts out by flashing the graphics raster on and off every two seconds. When the subprogram's ON CYCLE statement is activated during the fifth call to the subprogram, the new value (0.1 second) replaces the old value (2.0 seconds) and the program begins flashing the graphics raster at the new rate. Note that the branch to the second subprogram (Quit) is not executed because the first subprogram is finished before the specified time. To see the second subprogram execute, insert the following line.

```
191    WAIT 1
```

The delay caused by the WAIT statement allows the subprogram's ON CYCLE statement to branch to the second subprogram and stop execution.

If an active branch defined in the main program is canceled in a subprogram (by OFF TIME, OFF DELAY, or OFF CYCLE) any branch missed during the execution of the subprogram will be lost. When the context containing the original statement is restored, the branch will be reactivated and processing will continue as if no branch was missed.

```
10    ON DELAY 1 GOTO Done        !  GOTO "Done" in one second.
20    CALL Busy                   ! Call to "Busy" takes two seconds.
30    !
40    PRINT "THIS WON'T BE PRINTED UNLESS BRANCH IS CANCELED BY THE SUB"
50    !
60 Done:PRINT "THIS LINE WILL BE PRINTED EVERY TIME"
70    END
80    ! ----------------------------------------------------------
90    SUB Busy
100     WAIT 2
110   ! OFF DELAY  !  RUN then remove the "!" on this line and RUN again.
120   SUBEND
```

By removing the comment symbol (!) from the beginning of line 110, the OFF DELAY statement will be executed causing any branch that has already been logged to be canceled and allow line 40 to be printed.

Since branches only occur at the end of a line, no branch can be taken during an INPUT or LINPUT statement. The following program shows a method of monitoring the keyboard without preventing branches to be taken.

```
10     ON KBD GOTO Yes         ! If key is pressed go get new value.
20     ON DELAY 3 GOTO Gone    ! If no keypress in 3 seconds use defaults
30     DISP "PRESS A KEY"
40 W:  GOTO W                  ! Wait here until keypress or end of delay.
50     !
60 Yes:OFF DELAY               ! Someone is there.
70     OFF KBD
80     LINPUT "NEW VALUE?",A$
90     DISP "USING",A$
100    GOTO More
110    !
120 Gone:DISP                  ! Nobody there.
130    DISP "USING DEFAULTS"
140    !
150 More:WAIT 2
160    DISP "program continues...."
170    END
```

The program waits a few seconds for a response. Processing continues with default values if no key is pressed. Pressing a key will cause the program tg accept the new information.

Communicating with the Operator 10

Communicating with the Operator 10

It is very unlikely that a computer could perform useful work without receiving input. Much of that input is from electronic devices: instruments, mass storage devices, other computers, and so on. Because a computer is an electronic device, it is very good at these tasks. There are also times when the computer's input must come from the human sitting in front of the computer.

Good human interfaces do not happen without some effort from the programmer. In many programs, at least one fourth of the code is dedicated to human interface. It is not unusual to use one half of a good program for operator interaction, error trapping, explanatory messages, etc. Obviously, these estimates depend upon many factors, like the task being performed and the intended operators. If you are the only person who uses a program, that program may not need a friendly human interface. However, the demands for a good human interface rise greatly if a program is used by many people with different backgrounds. When the intended users do not understand computers, your program must be very skillfully written so that it does not intimidate the operator or make great demands on their ability to guess what your program wants from them.

Overview of Human I/O Mechanisms

Here are the elements of a human interface that this chapter discusses:

- Sending messages to the operator:
    - Displaying text for the operator to read (DISP and PRINT).
    - Changing display fonts (CHRX, CHRY, and SET CHR).
    - Generating sound (BEEP and SOUND).
- Handling messages from the operator:
    - Softkeys (ON KEY, KEY LABELS, LOAD KEY, SET KEY)
    - Rotary pulse generators (ON KNOB and ON CDIAL)
    - High-level alphanumeric input (INPUT, LINPUT, ENTER)
    - Low-level keyboard input (trapping key codes with ON KBD)

Other Factors

These are certainly **not** the only elements in a human interface. A good human interface can involve the placement of hardware, use of graphic and voice communication, data base management, artificial intelligence theories, and much more. However, you must begin somewhere. Hopefully, the hints in this chapter will help your present programs and whet your appetite for more elaborate human interfaces in future programs.

Displaying and Prompting

One of the simpler things to do for the operator is to display an explanation of what is happening or what is expected. In the early days of computers, memory was a scarce and expensive resource. Programmers were encouraged to use as little memory as possible. It seemed as though there was a contest to see who could put the most information into a 32-character message.

Please realize that those days are over. For example, there is no significant restriction on program size: the standard machine is shipped with over a half-million characters of memory, and there are usually at least 18 lines of 80 characters visible at all times on the display. If you are sending your operator tiny, cryptic messages, you are making an unnecessary mistake.

Displaying Messages: A Two-Step Process

Giving instructions to the operator can be viewed as two basic steps:

1. Prepare to use the display by putting it into a known, usable state (disable any unwanted modes, clear the screen, etc.).
2. Use as much of the display as necessary to give readable instructions.

Turning Off Unwanted Modes

There are several modes that affect the appearance of the display. Each is very useful for certain purposes; however, some are undesirable for the display of simple text. It is embarrassing to the programmer and confusing to the operator when two or more displays combine in an unplanned manner. The culprits are "left-over" alpha and "left-over" graphics. Left-over alpha can occur for a number of reasons:

- The operator may have used the knob or cursor-control keys to scroll text from the off-screen buffer.
- With TABXY, the PRINT statement overwrites any old characters on a line with new characters. However, if the old text is longer than the new text, the end of the old line remains visible. Therefore, the following statements do **not** print three blank lines. They just move the print position. Any old lines will still be on the screen.

```
100 PRINT
110 PRINT
120 PRINT
```

- With color displays, each of the different display regions may have a different color alpha pen in effect.
- If the PRINTALL mode is on, all interactions on the display line and keyboard input line are sent to the PRINT/OUTPUT area; this may interact in an undesired manner with your program's display.

Most, but not all, of these modes are discussed in this section. For a complete listing of all display modes, see the "Display Interfaces" chapter of *BASIC Interfacing Techniques.*

Disabling and Enabling Alpha Scrolling

If you want to disable the cursor-control keys (such as ↑ and ↓) from scrolling the alpha display, execute this statement:

```
CONTROL KBD,16;1
```

This will prevent the user from interfering with which part of the alpha buffer is currently shown. This techniques is useful, for instance, to prevent scrolling graphics when using a bit-mapped alpha display (on which graphics and alpha are on the same raster).

If scrolling is currently disabled and you want to re-enable it, execute this statement:

```
CONTROL KBD,16;0
```

Disabling Printall Mode

The PRINTALL mode is canceled by writing a zero in the PRINTALL control register (1). The following statement turns off the PRINTALL mode.

```
CONTROL KBD,1;0
```

Disabling Display Functions Mode

The DISPLAY FUNCTIONS mode can make a display look sloppy. The following equivalent statements turn off the DISPLAY FUNCTIONS mode.

```
CONTROL CRT,4;0
DISPLAY FUNCTIONS OFF
```

Softkey Labels

The following statement displays the softkey labels on the bottom of the screen:

```
KEY LABELS ON
```

For most programs that use softkeys (discussed later in this chapter), this is a desirable mode to be in.

If you have an ITF keyboard, you can select which softkey menu to display:

```
SYSTEM KEYS
USER 1 KEYS
USER 3 KEYS
```

These menus are also discussed later in this chapter (as well as in the "Introduction to the System" chapter of the *Using the BASIC System* manual).

Clearing the Screen

You can use either of the following (equivalent) statements to clear the display screen:

```
CLEAR SCREEN
```

or

```
CLS
```

If you do not want to load the CRTX binary, you can use an OUTPUT to the keyboard to accomplish the same purpose:

```
OUTPUT KBD;CHR$(255)&"K";
```

(The "Keyboard Interfaces" chapter of *BASIC Interfacing Techniques* fully describes the details of keyboard outputs.)

Printing Blank Lines

To print a line that is blank is a different operation from sending only an end-of-line sequence. A PRINT statement with no parameters simply sends an end-of-line sequence. If the print position is at the start of a blank line when PRINT is executed, that line remains blank. However, if there is text on that line, the text remains. This is not to say that it is "wrong" to use PRINT with no parameters. It just means that you cannot guarantee the output of a blank line by using PRINT with no parameters.

To print a blank line, blanks must be printed. One of the most convenient ways to send a line full of blanks is the TAB function. Here is a sequence that prints three blank lines:

```
100  STATUS CRT,9;Screen_width
110  PRINT TAB(Screen_width)
120  PRINT TAB(Screen_width)
130  PRINT TAB(Screen_width)
```

Disabling and Clearing Graphics Rasters

Left-over graphics can be turned off by the following statement (on displays which have separate alpha and graphics rasters):

```
GRAPHICS OFF
```

To clear the graphics raster, use this statement:

```
GCLEAR
```

Series 300 color (multi-plane) displays may be configured to use different planes for alpha and graphics so as to simulate the separate alpha and graphics rasters of some Series 200 displays.

```
SEPARATE ALPHA FROM GRAPHICS
```

Or you can also re-configure these displays so that alpha and graphics are not separate:

```
MERGE ALPHA WITH GRAPHICS
```

For further information concerning this topic, see the section called "Multi-Plane Bit-Mapped Displays" in the chapter called "Using Graphics Effectively" of *BASIC Graphics Techniques*.

Determining Screen Width and Height

The first step in displaying information on the screen is to determine its size. Programs written in this BASIC language can be used on either 50, 80 or 128-column displays. The height of displays may also vary. There are CRT status registers that contain the width and height of the screen.

If you are developing programs that will be transported between computers, status register 9 will be very helpful to you. The screen width is useful in centering displays, labeling softkeys, formatting tabular data, and other display tasks. The following statement places the screen width in a variable called `Crt_width`.

```
100  STATUS CRT,9;Crt_width
110  DISP "Screen width =";Crt_width
120  END
```

There is also a SYSTEM$ function that returns useful information about the CRT. The specifier "CRT ID" returns a string containing (among other things) the screen width and availability of highlights and graphics. The following example shows one method of determining the screen width with SYSTEM$.

```
120  Test$=SYSTEM$("CRT ID")
130  Crt_width=VAL(Test$[3,6])
140  DISP "Screen width =";Crt_width
150  END
```

You can also determine the screen's "current height," which is the number of lines currently enabled to display alpha information:

```
150  STATUS CRT,13;Height
160  DISP "Screen width =";Crt_width
170  END
```

The number of lines returned includes key labels, system message line, keyboard input line, DISP line, and OUTPUT area. See the subsequent discussions of display regions for locations of these lines.

Changing Alpha Height

You can also change the alpha height by writing to CRT control register 13; the range is 9 lines through the maximum for your particular display (25, 26 or 48).

These (equivalent) statements set the alpha height to 10 lines (which yields a 3-line output area, since the 10 lines begin at the KEY LABELS area at the bottom of the screen):

```
CONTROL CRT,13; 10
```

o

```
ALPHA HEIGHT 10
```

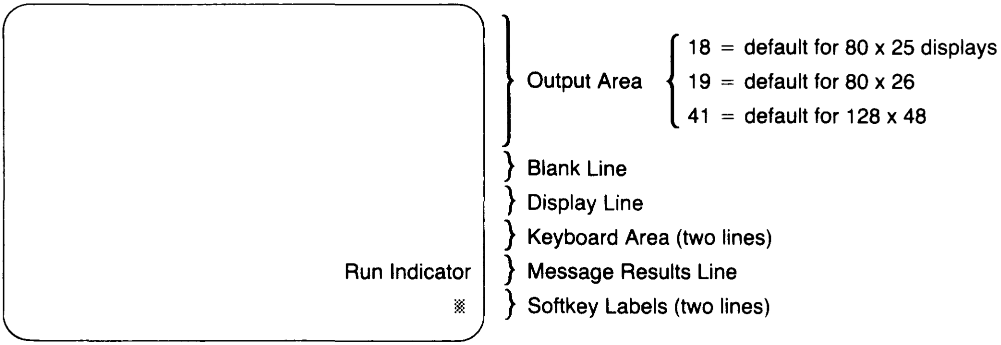
This is a handy way of specifying which part of the display is to be used for alpha and scrolling (particularly useful when using a bit-mapped alpha display and you want to use the top of the screen for graphics and the bottom part for text).

In order to return to the default alpha height, execute this statement:

```
ALPHA HEIGHT
```

Displaying Characters on the Screen

There are five regions of the display available for displaying messages for the operator.


Output Area
18 = default for 80 x 25 displays
19 = default for 80 x 26
41 = default for 128 x 48
Blank Line
Display Line
Keyboard Area (two lines)
Run Indicator
Message Results Line
Softkey Labels (two lines)

Each requires a slightly different method of displaying characters on the CRT screen:

- PRINT and OUTPUT CRT—place characters in the "Output Area" of the screen.
- DISP—places characters on the "Display Line".
- OUTPUT KBD—places characters in the "Keyboard Input Line", just as if you had typed them in at the keyboard.

- There is no *direct* method of displaying characters in the "System Message/Results" line (but you can do it indirectly, such as with `OUTPUT KBD;"Message"&CHR$(255)&"E";`).
- ON KEY—allows you to put characters in the "Softkey Labels".

Since these topics are fully covered in the "Display Interfaces" chapter of *BASIC Interfacing Techniques*, they will not be discussed here. See that chapter if you are not already familiar with these keywords.

Custom Character Fonts

With displays on which the alpha and graphics share the same raster—usually called "bit-mapped alpha displays"—you can re-define the bit patterns for characters. Here are the displays that have this capability:

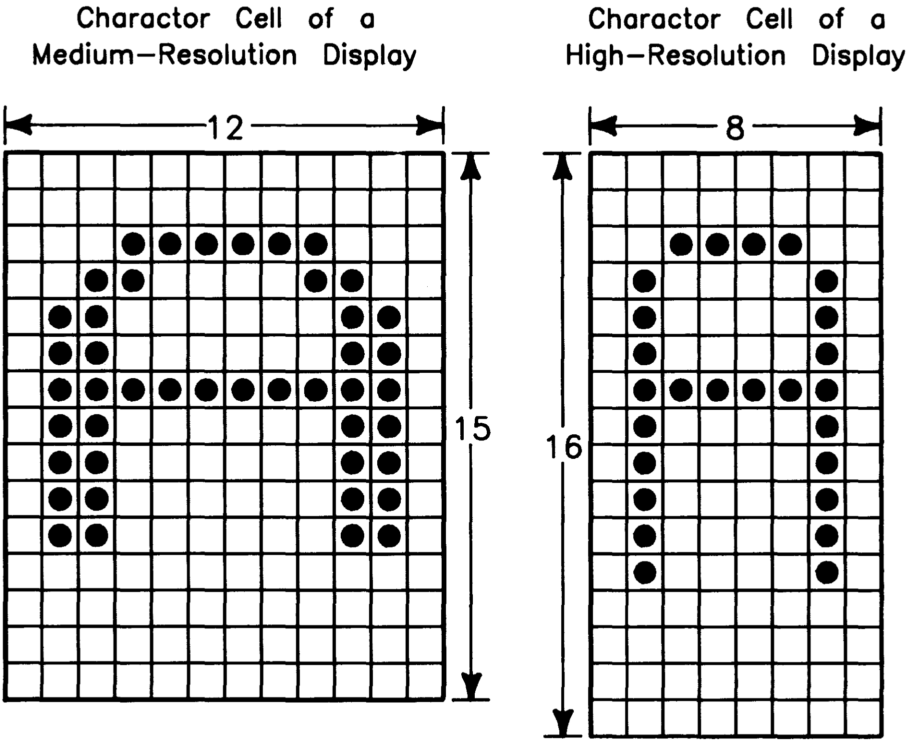
- Series 200 displays—*only* the Model 237 display.
- Series 300 displays—*all* displays *except* the HP 98546 Display Compatibility Interface.

This display architecture makes possible the definition of custom characters (and entire fonts), with the ability to change them under program control.

Character Cells

Before getting into how to change the pixel (dot) patterns in individual characters, let's see how characters are displayed on bit-mapped alpha screens.

Display characters are produced by turning on patterns of pixels—picture elements, or dots—in the shape of the intended characters. The following diagram shows the patterns for the letter "A" for each of the two sizes of bit-mapped alpha displays:



Charactor Cell of a Medium–Resolution Display

Charactor Cell of a High–Resolution Display

Determining Character Cell Size

As shown in the above diagrams, the character cell sizes are:

- 12 × 15 for medium-resolution bit-mapped displays
- 8 × 16 for high-resolution bit-mapped displays

You can determine the size of the character cell on the display currently in use with these BASIC functions:

- CHRY returns the height of the character cell (i.e., number of rows).
- CHRX returns the width of the character cell (i.e., number of columns).

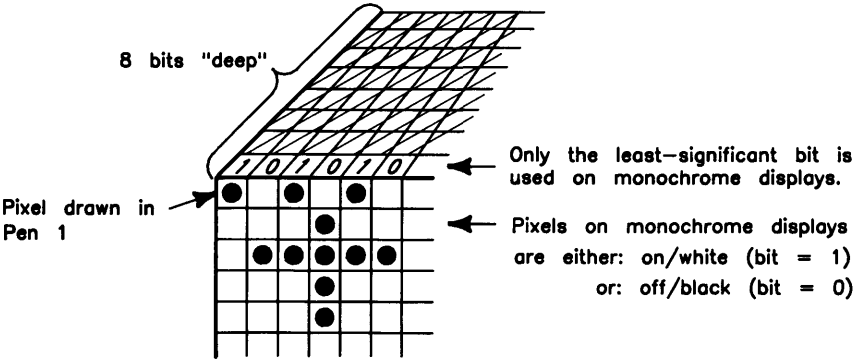
For instance, here are the results of executing these functions on a high-resolution display:

```
CHRX [Return]
 8
CHRY [Return]
 16
```

Character Font Storage in Memory

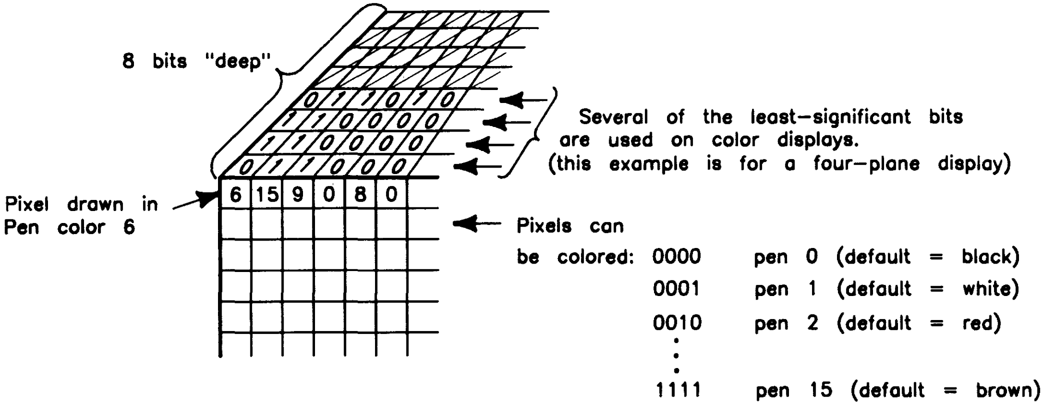
For each pixel of a character, there is one location in frame-buffer memory used to store the pixel.

- On monochrome displays, only the least-significant bit of this frame-buffer memory[1] location is used (one bit "deep", "single-plane" buffer).


8 bits "deep"
1 0 1 0 1 0
Pixel drawn in Pen 1
Only the least-significant bit is used on monochrome displays.
Pixels on monochrome displays are either: on/white (bit = 1)
or: off/black (bit = 0)

[1] The frame buffer is an area of memory on the display card used to hold the pixel patterns shown on the screen (frame). It also has some memory which is not displayed.

- On color displays, several of the least-significant bits are used— one for each plane; for instance, if it is a four-plane display (on which 16 colors can be displayed simultaneously), the low-order four bits of this INTEGER are used.


8 bits "deep"
0 1 1 0 1 0
1 1 0 0 0 0
1 1 0 0 0 0
0 1 1 0 0 0
Several of the least-significant bits are used on color displays. (this example is for a four-plane display)
Pixel drawn in Pen color 6
6 15 9 0 8 0
Pixels can be colored: 0000 pen 0 (default = black)
0001 pen 1 (default = white)
0010 pen 2 (default = red)
1111 pen 15 (default = brown)

Soft Font Usage

The font used by the BASIC system is stored in a fixed location of the undisplayed portion of frame-buffer memory[1]. Whenever a character is to be displayed, the display driver (CRTB) reads the bit pattern for the character, and then writes that pattern into the display buffer (the read/write memory that is displayed on the screen).

This is the character font used by the system at the following times:

- Whenever you type characters at the keyboard.
- Whenever PRINT is executed (when PRINTER IS CRT is in effect).
- Whenever DISP is executed.
- Any time the system writes characters on the display (such as when using CAT, when the system reports an error, when softkey labels are displayed, etc.).

If you change one or more characters of this font, these characters will be used by the system in *all* of these operations which the system subsequently performs.

[1] These locations are read by the "Read_chrs" subprogram of the FONT_ED utility, which is explained in the "BASIC Utilities Library" chapter of the *Installing and Maintaining the BASIC System* manual.

Restoring the Default Soft Font

The next few sections show you how to modify the font currently in memory. If you should for any reason want to restore the default font (the one in place when BASIC is booted), execute the following statement:

```
CONTROL CRT,21;0
```

This statement re-initializes the font to its boot-time default character set.

Example of Changing One Character

The following program shows an example of setting a new bit pattern for the character A (the example is for a high-resolution monochrome[1] display, which has a 16 row by 8 column character cell). Note that there is an *easier* way to do this, as shown in subsequent sections; however, this example is useful to show how the soft-font re-definition mechanism (SET CHR) works.

```
100  DATA 0,0,0,0,0,0,0,0
110  DATA 0,0,0,0,0,0,0,0
120  DATA 0,0,0,0,0,0,0,0
130  DATA 0,0,0,0,0,0,0,0
140  DATA 0,0,0,0,0,0,0,0
150  DATA 0,0,0,0,0,0,0,0
160  DATA 0,0,0,1,1,0,0,0
170  DATA 0,0,1,0,0,1,0,0
180  DATA 0,1,0,0,0,0,1,0
190  DATA 0,1,1,1,1,1,1,0
200  DATA 0,1,0,0,0,0,1,0
210  DATA 0,1,0,0,0,0,1,0
220  DATA 0,1,0,0,0,0,1,0
230  DATA 0,0,0,0,0,0,0,0
240  DATA 0,0,0,0,0,0,0,0
250  DATA 0,0,0,0,0,0,0,0
260  !
270  INTEGER Char_cell(1:16,1:8)
280  READ Char_cell(*)  !  Read data above into array.
290  !
300  PRINT "Before character re-definition: ";"A A A A"
310  PRINT
320  SET CHR NUM("A"),Char_cell(*)
330  PRINT "After  character re-definition: ";"A A A A"
340  PRINT
350  END
```

[1] For color displays, the example should use −1 instead of 1.

Lines 100 through 250 specify the bit patterns of the new character.

Line 270 dimensions an array used to store these bit patterns (one INTEGER element per pixel.

Line 280 reads these bit patterns specified in the DATA statements.

Line 300 shows what the character looks like before it is re-defined.

Line 320 changes the pattern currently used for the character "A" to the pattern read from the DATA statements.

Line 330 prints four A's to show what the character looks like after it has been re-defined.

Editing Supplied Fonts

The FONT_EDitor utility on the *BASIC Manual Examples Disc* provides a method of reading, decoding, and editing bit patterns. This section briefly describes the capabilities of this utility. For information on how to use the FONT_EDitor, see the "BASiC Utilities Library" chapter of the *Installing and Maintaining the BASIC System* manual.

Font Editor Utility Capabilities

Here are the tasks we need to describe how to perform using the supplied Font Editor Utility:

- Editing bit patterns of characters in the font
- Storing the edited font (in a file)
- Loading the font into memory
- Restoring the default font (the one that was in memory at the time that the BASIC system was booted)

Re-Defining an Entire Font

The SET CHR statement was used in a preceding example to re-define one character. It can also be used to re-define an entire font. The difference is that the array that stores the bit patterns will have another dimension, which is used to index the characters in the font.

```
       .
       .
870  ALLOCATE INTEGER Entire_font(0:256,1:CHRY,1:CHRX)
       .
       .
940  SET CHR 0,Entire_font(*)
       .
       .
```

For a closer look at this type of technique, see the description of the Font Editor Utility in the *BASIC Utilities Library* manual.

Generating Sound

Most Series 200/300 computers have the ability to generate single tones[1].

- On computers that are *not* equipped with an HP-HIL interface, the sound capability is limited to the BEEP statement. BEEP provides you with the ability to generate tones of software-selectable *frequency* and *duration*. For instance, the following statement generates a tone of approximately[2] 1220 Hz for 0.25 seconds:

  ```
  BEEP 1220 , 0.25
  ```

- On computers equipped with an HP-HIL interface, there is a sound-generator chip[3] which you can access from BASIC with the SOUND statement. You can use it in one of two modes:

  - When using simple numeric parameters (not a numeric array), SOUND allows you to generate a single tone; you may software-select *which tone generator* to use, as well as its *frequency*, *volume*, and *duration*. For instance, the following statement uses voice 1 to generate a tone at frequency 1220 Hz, of maximum volume, and with duration of 0.25 seconds:

    ```
    SOUND 1,1220,15,0.25
    ```

  - When using an INTEGER array, SOUND takes values from the array and interprets them in a special way to produce a series of tones on one or more of the available voices. Examples of this use are given subsequently in this section.

    ```
    SOUND Array_of_instrs(*)
    ```

The remainder of this section describes how to use the SOUND statement.

[1] These tones are actually square waves that are sent through a simple power-amplifier circuit to a low-power speaker, if present.

[2] See the *BASIC Language Reference* for the range of frequencies and durations available with this statement.

[3] The chip is a TI SN76494 Four-Voice Sound Generator.

Example of Single Tones

Load and run the "CScale" program from the *Manual Examples* disc. Here is a listing of the program. It plays all eight major notes in the key of the C.

```
120   DATA C,C#,D,D#,E,F,F#,G,G#,A,A#,B,C
130   !
140   Base_freq=523.25  !  Base_freq = C
150   FOR Note=0 TO 12
160     Freq=Base_freq*2^((Note)/12)
170     READ Note$
180     IF NOT POS(Note$,"#") THEN ! "Natural" note.
190       PRINT USING 200;Note$,Freq
200       IMAGE "Note:",X,2A,3X,"Frequency:",2X,4D.DD
210       SOUND 1,Freq,8,.5
220       WAIT .5
230     END IF ! Natural note.
240   NEXT Note
250   !
260   !
270   END
```

Here are the printed results of running the program.

```
Note:  C   Frequency:  523.25
Note:  D   Frequency:  587.33
Note:  E   Frequency:  659.26
Note:  F   Frequency:  698.46
Note:  G   Frequency:  783.99
Note:  A   Frequency:  880.00
Note:  B   Frequency:  987.77
Note:  C   Frequency: 1046.50
```

Here is a line-by-line description of the program:

Line 120 lists the notes of the scale.

Line 140 specifies the frequency of middle C. (This will be used in calculating subsequent frequencies.)

Lines 150 through 240 define a loop which reads the notes in the DATA statement and calculate the corresponding frequency (if the note is a "natural" note—not a sharp or a flat).

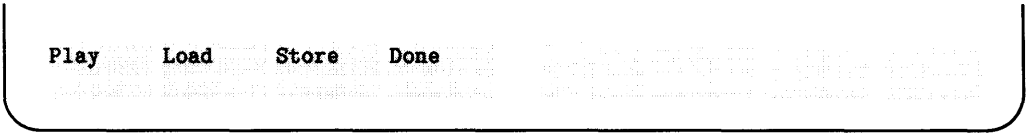
Lines 190 and 200 print the note and its calculated frequency.

Line 210 generates the note. (The actual frequency often does not *exactly* match the specified frequency. See SOUND in the *BASIC Language Reference* for a table of target frequencies and errors.)

Line 220 executes a WAIT statement, which allows the note to finish playing before the next note is sent to the sound chip.

A Simple Music Editor

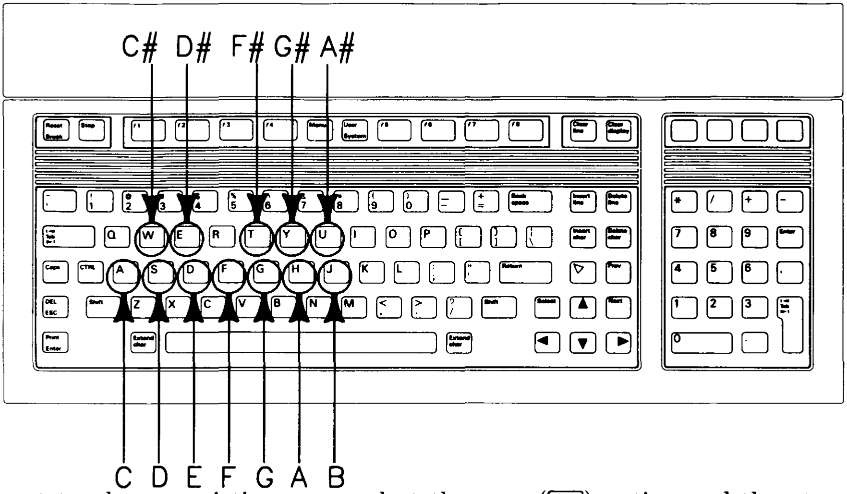
The "InputSong" program on the *Manual Examples* disc re-defines the keyboard to produce notes in the equal-tempered scale. Here is the softkey menu shown by the program (when using an ITF keyboard), along with the things you can do with the program:



```
Play    Load    Store   Done
```

- Input a sequence of tones (see key definitions below)
- Play the notes back (press f1 or k1)
- Load notes from a file (f2 or k2)
- Store them in a file (f3 or k3)
- Quit the program (f4 or k4)

Here are the definitions of the keys provided by the program:



If you want to play an existing song, select the **Read** ([f2]) option and then type in the file name `OdeToJoy`. It plays a short song.

The program has been kept simple for the sake of brevity—*very* simple. It would be fairly easy to enhance the program's capabilities: allow longer songs, add some elementary editing capabilities, as well as some graphic output to the program, etc. Such modifications, to use a familiar phrase, "are left as an exercise for the reader."

Arrays of Sound Instructions

As mentioned earlier, the SOUND statement also has the ability to play several tones, based on *instructions* given in an array specified in the statement.

```
SOUND Array_of_instrs(*)
```

The values in the array are interpreted by the SOUND statement as follows:

Instruction	Sound Chip Effect Produced
0	Exit the SOUND statement (and stop reading array elements)
1 to 3	The specified voice is to be used; also says to read the *next three* array elements, and interpret them as follows, respectively: • tone number—used to set the frequency (frequency = 83 333 / tone number). • volume—0 = off; 1 thru 15 are lowest to highest volume. • duration—values 0 thru 255 are interpreted as follows: 0 is interpreted as "sound indefinitely". 1 thru 255 are interpreted as 10's of milliseconds (i.e., 1/100 second);

4	Specifies that the **noise voice** is to be used; also says to read the next *three* array elements and interpret them as above (the same as with voice numbers 1 to 3), *except* that the *tone number* parameter is interpreted as follows:
	0 => *periodic* noise; *fast* shift register clock;
	1 => *periodic* noise; *medium* shift register clock;
	2 => *periodic* noise; *slow* shift register clock;
	3 => *periodic* noise; clock shift register *with voice 3*;
	4 => *white* noise; *fast* shift register clock;
	5 => *white* noise; *medium* shift register clock;
	6 => *white* noise; *slow* shift register clock;
	7 => *white* noise; clock shift register *with voice 3*.

5 to 8	Wait for voice 1 to 4, respectively, to finish sounding before executing the next sound instruction (if any).
9	Read the following array element, and wait the specified interval (100 microseconds × that element's value) before executing the next instruction (if any).

If the end of the array is reached on one of these boundaries, then the SOUND statement terminates normally; however, if the last element of the array has been reached and the BASIC system expects to read more values, then error 17 will be reported (subscript out of range).

Executing Example SOUND Instructions

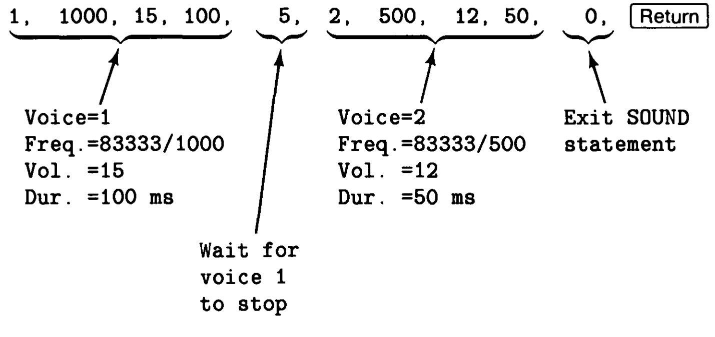
Here is a simple program that will allow you to experiment with some of these instructions. It is called "SoundInstr", and it is also on the *Manual Examples* disc.

```
120   OPTION BASE 1
130   ALLOCATE INTEGER Sound_array(10)
140   !
150   DATA  1,1000,15,100,5,2,500,12,50,0
160   READ Sound_array(*)
170   LOOP
180     OUTPUT KBD;Sound_array(*), ! Put "template" on input line.
190     INPUT "Edit SOUND array parameters.",Sound_array(*)
200     !
210     SOUND Sound_array(*) ! Now execute instructions.
220     !
230   END LOOP
240   !
250   END
```

After loading the program (or typing it in), run it and begin to experiment by varying the instructions. (Use the Stop key to terminate the program.)

The **first time through the loop**, merely press Return to execute the instructions shown on the keyboard input line. Here are the default instructions, with an explanation of their effects.

```
Edit SOUND parameters:
```



The **second time through the loop**, move the cursor to the left and modify one or two of the parameters, and then press Return to execute the instructions. For instance, this is a legal set of instructions:

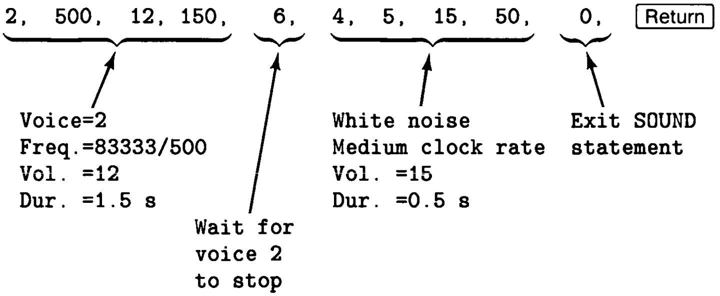
```
1, 1000,  15, 100,   6,   2, 500,  12,  50,   0, Return
                     ↑
                     │
                Changed only
               this parameter.
```

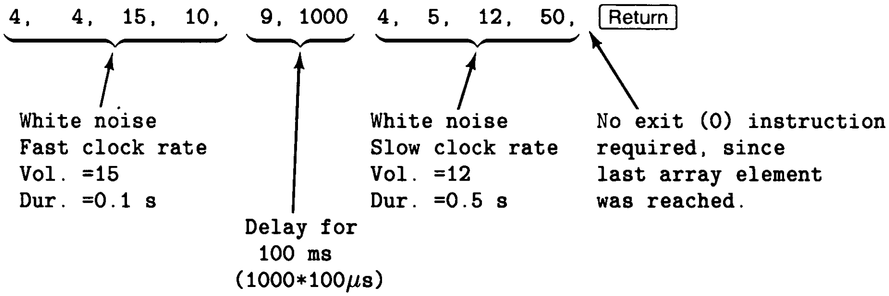
Since the only parameter that was changed was the 6, the sounds on voices 1 and 2 are now played simultaneously (instead of waiting for voice 1 to stop before starting voice 2). Note that voice 2 (the higher pitch) stops first, since it had the smaller duration parameter.

The **third and subsequent times through the loop**, you can do either of the following things:

- Press Return to re-play the preceding set of instructions.
- Move the cursor (◄ or ►), modify one or more of the instructions (type over existing characters, or use Insert char/Delete char), and then press Return to hear the instructions' effects.

Here are several additional examples of instruction sequences and their effects.





Example Song (Using SOUND Array Parameters)

The program in the file named "SoundArray" on the *Manual Examples* disc produces a song using the SOUND statement and an array of instructions. Load and run the program.

Operator Input

After sending messages to the operator about what you want, you can expect that you will get some sort of meaningful feedback. This section summarizes the different methods of handling operator inputs.

- Softkeys (ON KEY, KEY LABELS, LOAD KEY, SET KEY)
- Rotary pulse generators (ON KNOB and ON CDIAL)
- High-level alphanumeric input (INPUT, LINPUT, ENTER)
- Low-level keyboard input (trapping key codes with ON KBD)

Softkey Inputs

Softkeys are the keys at the top of your keyboard labeled f1 through f8 (on ITF keyboards) or k0 through k9 (on 98203 keyboards)

There are two types of uses of softkeys:

- **Typing-aids keys:** these keys generate sequences of alphanumeric and system keystrokes, which will save you time when repeatedly typing in information or commands from the keyboard
- **Program-interrupt keys:** while a program is running, the softkeys can generate interrupts (when ON KEY defines service routines for the keys)

Note that if a softkey does not have a current ON KEY definition, it can still be used as a typing-aid.

Since both of these topics are already discussed in other places of the BASIC manual set, they will not be discussed here. For further information about:

- **Typing-aid keys:** see "Introduction to the System" in the *Using the BASIC System* manual.
- **Program-interrupt keys:** see the "Program Structure and Flow" chapter of the *BASIC Programming Techniques* manual, and the "Interrupts and Timeouts" chapter of *BASIC Interfacing Techniques*.

However, since programmatically re-defining the softkeys is not an appropriate topic for *Using the BASIC System* manual, it will be discussed here.

Defining Typing-Aid Softkeys Programmatically

There are two ways to programmatically re-define the typing-aid definitions of softkeys:

- Use LOAD KEY to load definitions from a file (for *all* keys). Note that LOAD KEY with no arguments restores the default definitions of the softkeys.
- Use SET KEY to load definitions from a simple string (one key) or from a string array (ranges of keys, or all keys).

The **main differences** between these statements are shown in the following table:

Method	Source of Definitions	Number of Keys Defined
LOAD KEY	BDAT file	*All* existing definitions are first cleared; then *only* keys with definitions in file are re-defined.
SET KEY	Simple string, or string array	*Single* keys or *ranges* of keys may be re-defined.

Listing Current Typing-Aid Softkey Definitions

Before getting started into how to *change* typing-aid definitions, it is handy to have a tool for checking the *current* definitions. The LIST KEY statement allows you to show these current definitions. The destination is either the current PRINTER IS device:

```
LIST KEY
```

or the specified device:

```
LIST KEY #Dev_selector
```

Storing and Loading Typing-Aids from Files

To store the current typing-aid definitions, use the STORE KEY statement. STORE KEY first creates a BDAT file of the specified name, and then stores two types of information in this file *for each key* (written in FORMAT OFF[1] representation):

- A key number (2-byte INTEGER)
- The corresponding typing-aid softkey's definition (a 4-byte string-length header, followed by a string of ASCII characters that comprise the key's definition)

[1] For details of FORMAT OFF attribute, see the *BASIC Language Reference* description of ASSIGN; or see the *BASIC Interfacing Techniques* description in the "I/O Path Attributes" chapter.

In order to load these keys back into the computer, use the LOAD KEY statement. LOAD KEY first clears *all* current typing-aid definitions, and then loads new typing-aid softkey definitions from a BDAT file.

This file was created in one of two ways:

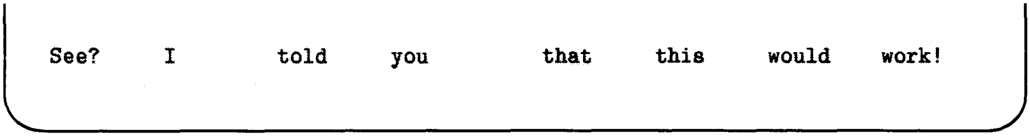
- By using STORE KEY (after making sure that all typing-aid softkey definitions were as desired); for example:

  ```
  STORE KEY "SOFTKEYS"
  ```

- By using OUTPUT to send the same information to a BDAT file. Here is an example program that does essentially what the preceding STORE KEY statement does (assuming the same typing-aid definitions, of course):

  ```
  100   ! File "LOAD_KEY"                 ! File name of this program.
  110   DIM Key_def$[160]                 ! In case of LONG definitions.
  120   INTEGER Key_number                ! 16-bit integer.
  130   CREATE BDAT "SOFTKEYS",3          ! Create a 3-record file.
  140   ASSIGN @Keys TO "SOFTKEYS"        ! Open file (default=FORMAT OFF).
  150   FOR I=1 TO 8                      ! For all softkeys (ITF keyboard).
  160     READ Key_number,Key_def$        ! Read key# and definition.
  170     OUTPUT @Keys;Key_number,Key_def$ ! Write them in file.
  180   NEXT I
  190   ASSIGN @Keys TO *
  200   LOAD KEY "SOFTKEYS"                ! Now install the definitions.
  210   !
  220   STOP
  230   DATA 5,"that",8,"work!",4,"you",7,"would"
  240   DATA 2,"I",1,"See?",3,"told",6,"this"
  250   END
  ```

  Here are the resultant softkey labels on an ITF keyboard. (Details about the number of characters available for softkey labels are shown in a subsequent section.)



```
See?   I       told    you        that    this    would   work!
```

The proper way for a program to handle typing-aid definitions when it does not want to make permanent modifications is to store the existing definitions in a file and reload them at exit time. Here is an example of how this can be done:

```
10 INITIALIZE ":,0",9      ! create a memory volume to hold the file
20 STORE KEY "Key_defs:,0" ! store the key definitions in the file "Key_defs"
30 DIM A$(23)[1]
40 SET KEY 0,A$(*)         ! redefine all the keys to undefined
50 PAUSE
60 LOAD KEY "Key_defs:,0"  ! reload the old definitions of the keys
70 INITIALIZE ":,0",0      ! reclaim the memory volume storage
80 END
```

Note that LOAD BIN and memory volumes use a mark/release stack, so that the memory volume storage can only be reclaimed if:

- no LOAD BIN was done after the INITIALIZE in line **10** above
- other memory volumes INITIALIZEd after it have been reclaimed

It should be released even if the second case mentioned above is not satisfied, since a subsequent release of the later volumes will reclaim as many released memory volumes as it can.

Using SET KEY

SET KEY allows you to either define a single typing-aid or to define multiple typing-aids, depending on the string parameter you specify in the statement.

The following example program lines define typing-aid softkey f2 (k2) to clear the current line and produce some characters:

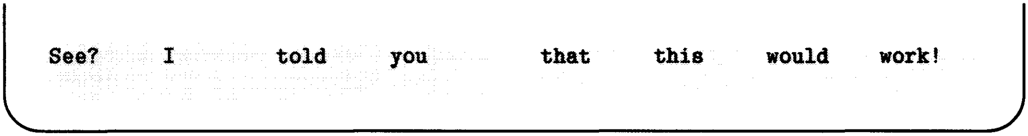
```
100  String$=CHR$(255)&"#"&"Some characters"
110  SET KEY 2,String$
```

The softkey label depends on the first few letters of the string `Some characters`. (The characters used for the label vary for different keyboards, as well as other factors. See the subsequent table for details.)

The following example program defines typing-aid softkeys f1 through f8 (k1 through k8) exactly as in the preceding example.

```
100   ! File "SET_KEY"                   ! File name of this program.
110   DIM Key_def$(1:8)[160]             ! In case of LONG definitions.
120   INTEGER Key_number                 ! 16-bit integer.
150   FOR I=1 TO 8                       ! For all softkeys (ITF keyboard).
160     READ Key_number                  ! Read key number (from DATA statements).
170     READ Key_def$(Key_number)        ! Read corresponding key definition.
180   NEXT I
200   SET KEY 1,Key_def$(*)              ! Now install the definitions.
210   !
220   STOP
230   DATA 5,"that",8,"work!",4,"you",7,"would"
240   DATA 2,"I",1,"See?",3,"told",6,"this"
250   END
```

Here, again, are the resultant softkey labels:



```
See?    I       told    you       that    this    would   work!
```

Softkey Labels

The following table shows the number of characters available for softkey labels for each type of keyboard and display used with Series 200/300 computers.

Display Type	ITF Keyboard	ITF Keyboard in KBD CMODE	98203 Keyboard
High-resolution display (128-columns)	16 (2×8)	14 (2×7)	16
Medium-resolution display (80-columns)	16 (2×8)	14 (2×7)	14
Model 226 display (50-columns)	n/a	n/a	8

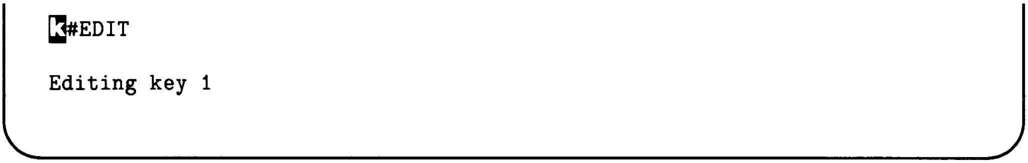
Note that the figures of “2×8” and “2×7” show that there are 2 lines of 8 or 7 characters each.

Some strings produce special effects if present at the beginning of the key label text. Most of these character sequences represent "System key" presses, such as CLR LN, (Clear line on an ITF keyboard), and STEP. You can type these characters into a typing-aid key by holding down the CTRL key while pressing the desired system key. The two-character key code produced contains a leading CHR$(255), which shows up as an inverse video K, followed by the character shown in the following table.

Characters	System Key Represented	Effect on Key Label
KS	STEP (f1)	`Step` displayed in key label (if these are the only 2 characters in the label).
KC	CONTINUE (f2)	`Continue` displayed in key label (if these are the only 2 characters in the label).
KR	RUN (f3)	`RUN` displayed in key label (if these are the only 2 characters in the label).
KA	PRT ALL (f4)	`Print All` displayed in key label (if these are the only 2 characters in the label).
KF	DISPLAY FCTNS (f6)	`Display Fctns` displayed in key label (if these are the only 2 characters in the label).
tenttk$	ANY CHAR (f7)	`Any Char` displayed in key label (if these are the only 2 characters in the label).
K#	CLR LN (Clear line)	Not displayed in key label (if they are the 1st two characters in the label).
$^{\mathrm{U}}\underline{\mathrm{L}}$ CHR$(132)	"Underline" character (not a system keycode)	If the key label has two rows, and if this character is either the 1st character in the key label or immediately follows K#, the system draws a line between the top row and the bottom row of key label characters. (Otherwise, no effect on key labels.)

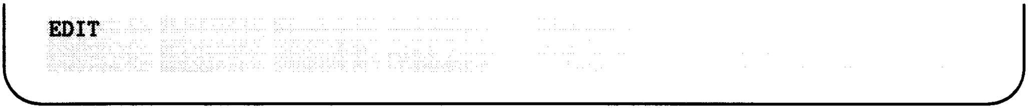
Examples

On anD ITF keyboard, [f1] clears the current line and produces the characters `EDIT`. The contents of this typing-aid key are:



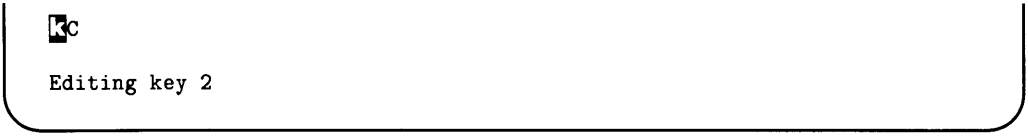
```
k#EDIT

Editing key 1
```

The label looks like this:



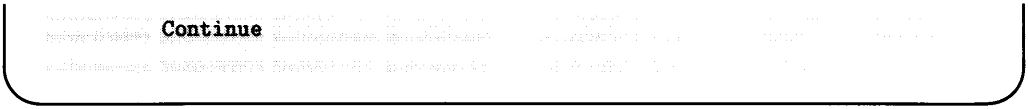
```
EDIT
```

As another example of an ITF keyboard key, [f2] produces a [CONTINUE] System key press. Its contents are *only* the following two characters:



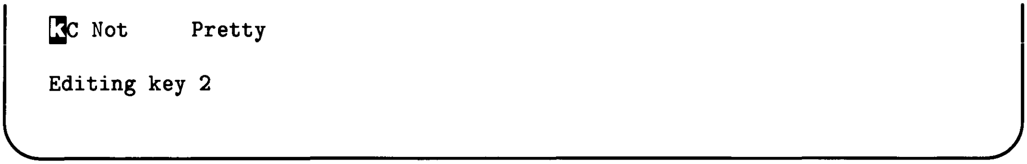
```
kC

Editing key 2
```

Here is the corresponding label:

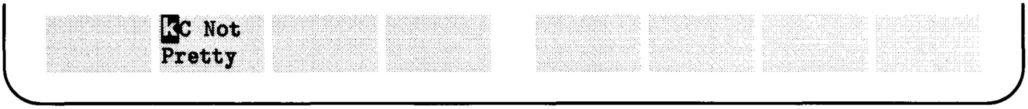


```
            Continue
```

Note that in the preceding "System key sequences" table, the first five of the two-character sequences must be the *only* two characters in the softkey's definition. If there are other characters in the definition, then the label shown in the table will not be displayed; the subsequent characters are instead displayed. For instance, the following typing-aid key contents would produce the following label:

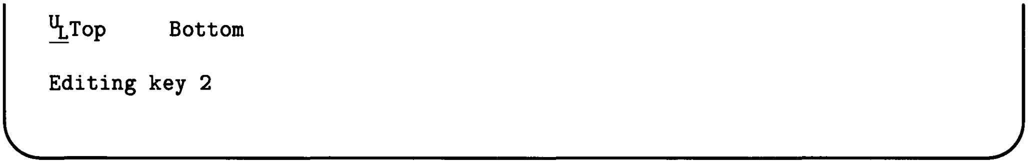


```
KC Not     Pretty

Editing key 2
```

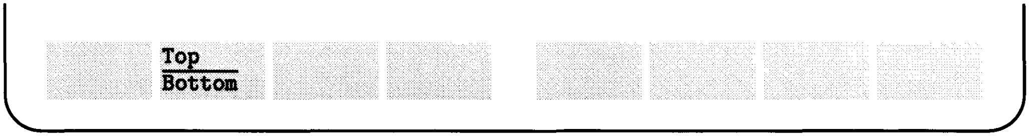


```
KC Not
Pretty
```

As another example, suppose that you have an ITF keyboard and an 80-column display. Since the key labels can have 16 characters, in 2-row-by-8-column format, it might be desirable to use the underline character to split the rows. This typing-aid key contents would produce a two-row key label:



```
ULTop     Bottom

Editing key 2
```



```
Top
Bottom
```

Using Knobs

The ON KNOB and GRAPHICS INPUT IS statements allow you to programmatically sense knob rotation. Knob inputs can be received from built-in knobs on 98203-type keyboards, or from HP-HIL knobs, and they can also be received from a mouse. This section only discusses how to trap knob rotation by using ON KNOB. See the "Interactive Graphics and Graphics Input" chapter of *BASIC Graphics Techniques* for examples of using GRAPHICS INPUT IS.

The following program is a very short example that demonstrates a real-time interaction between the knob and the CRT. If you run this example program and turn the knob, you will see the kind of interaction that might be used for cursor control in a text editor, for instance. Obviously, a real cursor-control routine would be much more sophisticated, but this demonstrates the basic idea.

```
100  ON KNOB .1 GOSUB Move_blip
105  STATUS CRT,13;Alpha_height
110 Spin:  GOTO Spin
120  !
130 Move_blip:  !
140    PRINT TABXY(Spotx,Spoty);" ";  ! Erase old 'blip'.
150    Spotx=Spotx+KNOBX/5            ! Scale knob inputs.
160    Spoty=Spoty+KNOBY/5
170    IF Spoty<1 THEN Spoty=1        ! Check range.
180    IF Spoty>Alpha_height THEN Spoty=Alpha_height
190    IF Spotx<1 THEN Spotx=1
200    IF Spotx>50 THEN Spotx=50
210    PRINT TABXY(Spotx,Spoty);CHR$(127); ! Display new 'blip'.
220    RETURN
230  END
```

This example uses a short infinite loop to wait for pulses from the knob (line 110). Interrupts from the knob are enabled by the ON KNOB statement in line 100. The service routine erases the old "blip", performs some scaling and range checking on the knob input, and prints the new "blip".

The scaling and range checking are very important in this kind of interactive routine. Humans expect their interface to have a certain "feel." Displays should not change too quickly or too slowly. Certain kinds of displays are expected to change logarithmically, others are expected to change linearly. The boundary values of variables are expected to conform to the boundaries of the display. To initiate yourself to some of these concepts, try modifying this simple example. Remove one or more of the range checking lines. (An easy way to do this kind of editing is to place an exclamation point in front of the statement. This turns it into a comment, removing it from the flow of execution. But it can be easily returned to the program by deleting the exclamation point.) Also try changing the scaling factor in lines 150 and 160. Notice the "feel" that results from larger and smaller increments, or even logarithmic scaling.

Using Control Dials

BASIC provides the ability to set up event-initiated branches upon detecting the rotation of knobs on "Control Dial" devices (such as the HP 46085A Control Dial Box).

Keywords and Capabilities

There are three BASIC keywords for accessing multiple-knob devices:

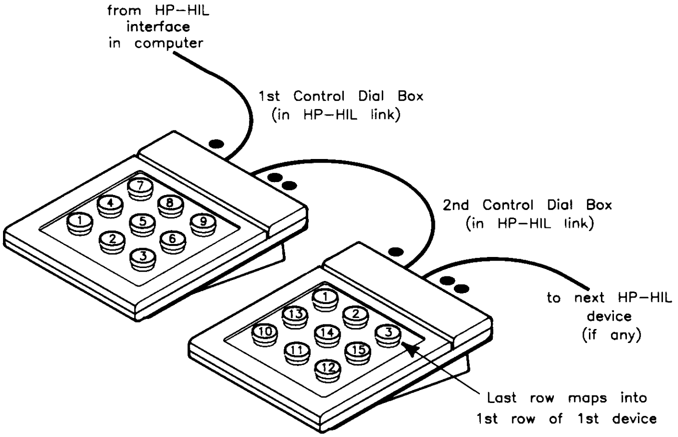
- ON CDIAL—sets up and enables interrupt branch upon detecting rotation of one of the control dials.
- CDIAL—interrogates the BASIC system to determine:
  - Which knob(s) have been rotated?

    `CDIAL(0)` returns a 16-bit status word, with each bit corresponding to a dial number (for example, bit 15 set indicates that dial number 15 has been rotated, while bit 1 indicates the same for dial 1.

- How much a particular knob has been rotated?

  `CDIAL(1)` returns the number of pulses accumulated for dial 1;
  `CDIAL(2)` returns the number of pulses accumulated for dial 2;
  .
  .
  `CDIAL(15)` returns the number of pulses accumulated for dial 15.

  Here is the numbering of dials used by CDIAL:


from HP-HIL interface in computer
1st Control Dial Box (in HP-HIL link)
2nd Control Dial Box (in HP-HIL link)
to next HP-HIL device (if any)
Last row maps into 1st row of 1st device
1
2
3
4
5
6
7
8
9
1
2
3
10
11
12
13
14
15

- OFF CDIAL—disables interrupt branching for control dials.

Does BASIC See the Control Dial Box?

The "Verifying and Labeling Devices" chapter of the *Installing and Maintaining the BASIC System* manual describes how to check to see that HP Human Interface Link (HP-HIL) devices have been properly connected to the computer, are functioning correctly, and have been logged in by the BASIC system. If you have not performed that verification yet, you should do so now.

An Example Control Dial Handler

The following example program sets up an interrupt service routine for one Control Dial box. The program is named "CDials", and it is on the *Manual Examples* disc. The program draws a box in a 3-dimensional coordinate system. The dials are defined to perform the following actions:

Dial 1 Changes the "X" location of the box.

Dial 2 Changes the "Y" location of the box.

Dial 3 Changes the "Z" location of the box.

Dial 4 Changes the "X" size of the box.

Dial 5 Changes the "Y" size of the box.

Dial 6 thru 9 No action has been implemented.

Here is the pertinent part of the interrupt-service routine for the Control Dial box:

```
350   Bits=CDIAL(0)           ! Read 16-bit status word (which knobs?)
360   !
370   IF BIT(Bits,1) THEN X=X+.1*CDIAL(1) ! Dial 1 turned; change X pos.
380   IF BIT(Bits,2) THEN Y=Y+.1*CDIAL(2) !  "   2 turned;   "    Y pos.
390   IF BIT(Bits,3) THEN Z=Z+.1*CDIAL(3) !  "   3 turned;   "    Z pos.
400   !
410   IF BIT(Bits,4) THEN X_size=X_size+.2*CDIAL(4)   ! Change "X_size".
420   IF BIT(Bits,5) THEN Y_size=Y_size+.3*CDIAL(5)   ! Change "Y_size".
430   !
```

Line 350 interrogates the "status word" to determine which dial(s) have been rotated. For each one that has been rotated, the corresponding action is taken (lines 370 through 420). For instance, rotating dial 1 moves the box along the X axis (while the Y and Z coordinates remain fixed). Rotating dial 5 changes the "Y size" of the box (while the "X size" remains constant).

In order to implement additional functions, all you need to do is to add similar IF...THEN statements (or segments) that execute the appropriate action.

Accepting Alphanumeric Input

When possible, it is a very good choice to used only softkeys and knobs to get input from the operator. It eliminates the need for translating an endless variety of typing mistakes that might be supplied as input to program variables. Softkey input is very tightly controlled by the programmer. Unfortunately, it is often necessary to leave that comfortable, controlled world. Suppose you need to get a device selector from the operator. You can't very well define a softkey that increments a variable and expect the operator to press it 701 times!

The proper handling of keyboard input may be one of the most neglected areas of applications programs. Programmers often fail to see the program as users see it, underestimate the potential for operator error, and balk at the amount of code needed to skillfully handle incoming text. However, you need not write input routines that can parse broken English with misspelled words. The objective is simply to keep the program from terminating and to take some unnecessary pressure off the operator. Obviously, a program can't tell if the operator misspelled a file name until it accesses the disc. Therefore, error trapping is an important part of handling operator input.

One task that can be performed by the input routine is **anticipating common problems**. Many of these are covered in this section's examples, but here is a preview. You know that exceeding the dimensioned length of a string gives error 18. So don't use short strings in an INPUT statement. You know that CAPS LOCK might be on or off when the operator starts typing. So use an uppercase function to compare input with constants. You know that an operator is likely to just press CONTINUE if he isn't sure how to respond. So use reasonable defaults and don't try to send a null string to a NUM function.

Get Past the First Trap

Before you can do anything with a keyboard input, the computer must satisfy the items in the input list and complete the input statement. There are two keywords available for accepting input from the keyboard line: INPUT and LINPUT. Let's start by looking at the features of these two statements.

The main advantages of INPUT are:

- Either numeric or string values can be input.
- If a variable does not receive a value from the keyboard, the value of that variable is left unchanged.
- A single INPUT statement can process multiple fields, and those fields can be a mix of string and numeric data.

The INPUT statement can be powerful and flexible. When you know the skill level of the person running the program, INPUT can save some programming effort. However, this statement does demand that the operator enter the requested fields properly. To find out the details of INPUT, see the *BASIC Language Reference.* This section discusses an alternative to INPUT that can make fewer demands on the operator. Some of the **disadvantages** of INPUT are:

- Improper entries to numeric variables can cause errors such as "string is not a valid number" and overflows.
- Certain characters can cause problems. Commas and quote marks have special meanings and are the primary offenders.
- If DISP is used to supply a prompt, and multiple values are entered separately, the prompt is lost.

The problem with INPUT is that the program is powerless to overcome the disadvantages. If you are asking for a numeric quantity, and the operator keeps trying to enter a name, the program will never leave the INPUT statement. The operating system will beep and display error 32 until the operator gets tired or gets smart. In the event of an error, the computer automatically re-executes the INPUT statement until the operator satisfies all the requirements. Your program never gets a look at his input and you can't trap the errors.

The LINPUT statement can help with these potential problems. LINPUT stands for "Literal INPUT." The result of any LINPUT statement is a single string that contains an exact image of what the operator typed. If CONTINUE (f2 on the ITF keyboard) is pressed with no entry, the result is the null string. (Nothing typed, nothing returned.) If you need things like default values, numeric quantities, and multiple values, you will need to process the string after you get it.

Since LINPUT accepts any characters without any special considerations, the only normal error would be string overflow. If the string used to hold the LINPUT characters is dimensioned to 256 characters or more, it becomes impossible to overflow the string from the keyboard line. Therefore, LINPUT is a very "safe" way to get data from the keyboard line. The following example shows some common techniques for accepting operator input.

Entering a Single Item

This program segment requests the current month for use later in the program. A detailed discussion follows the listing. Note that the general techniques presented can be used to process many kinds of input. Entering a month is merely a convenient example.

```
100   OPTION BASE 1
110   DIM In$[160],Months$(12)[3]
120   INTEGER Temp,Current_month
130   OUTPUT KBD;"SCRATCH KEYkX";      ! Typing aids distracting if not needed
140   FOR Temp=1 TO 12
150     READ Months$(Temp)             ! String data for month names
160   NEXT Temp
170   DATA JAN,FEB,MAR,APR,MAY,JUN,JUL,AUG,SEP,OCT,NOV,DEC
180   Current_month=3                  ! Default value
190   !
200 Try_numeric:   !
210   DISP "Enter the month.  Default = ";Months$(Current_month);
220   LINPUT "",In$                    ! Ask for operator input
230   IF NOT LEN(In$) THEN             ! Check for no input
240     Temp=Current_month             ! Use default value
250     GOTO Found
260   END IF
270   ON ERROR GOTO String             ! If no numerals, may be a string name
280   ENTER In$;Temp                   ! Try to extract a number
290   OFF ERROR                        ! ENTER worked; change error trap
300   IF Temp<1 OR Temp>12 THEN Not_valid   ! Check for impossible month value
310   GOTO Found                       ! Value is OK; use it
320   !
330 String:  !
340   OFF ERROR                        ! ENTER error trap no longer needed
350   In$=UPC$(In$)
360   FOR Temp=1 TO 12                 ! Search for 1st three letters of month
370     IF POS(In$,Months$(Temp)) THEN Found  ! Match found; use that value
380   NEXT Temp                        ! If loop finishes, no match was found
390   !
400 Not_valid:   !
410   BEEP
420   DISP "Not a valid month. Please try again."
430   WAIT 2
440   GOTO Try_numeric
450   !
460 Found:   !
470   Current_month=Temp
480   !
490   ! Program execution continues here
```

The first statement after the variable declarations removes the typing-aid key definitions. This is done with an OUTPUT to the keyboard because SCRATCH commands cannot be stored as a program line. You may or may not want to include this in your programs. If you are not using softkeys, the presence of softkey labels may be distracting to the operator. They may indicate that many response choices are available when the keys are actually unrelated to the current question. On the other hand, your program may have loaded the typing aids with responses intended to help the operator. This is possible, but was not done in the example. Obviously, if KBD is not present, the SCRATCH KEY command will generate an error and shouldn't be included. For another method of removing the typing-aid key definitions, read the section in this chapter entitled "Storing and Loading Typing-Aids from Files."

An interesting feature of this example is that the operator may respond with the number of the month, the name of the month, or an abbreviation of the name of the month. The array `Months$` is loaded with the first three letters of each month name so that name responses can be identified.

The final initialization step is to provide a default for the current month. When possible, requests for input should be accompanied by a default. If the default is well chosen, this increases the chances that the operator will not have to do any typing. Even if the default will usually be changed, it can help show the operator an acceptable format for the response.

The prompts available with INPUT and LINPUT statement must be literals and therefore cannot show any program variables. This restriction is easily overcome. Prompts appear in the same line as DISP items. The DISP statement can contain variables. To use DISP items as a prompt, a trailing semicolon is used in the DISP statements, and a null prompt is used in the LINPUT statement. This is a very useful technique that is applicable to both LINPUT and single-prompt INPUT statements.

After the keyboard input is received, the first check determines if any data was entered. It is reasonable to assume that the space bar might have been bumped accidentally during any keyboard input. The TRIM$ function corrects this "problem." A null input indicates that the operator wanted the default value, so no further processing is done.

The next check is to see if the number of the month was entered. Numerals can be converted to numeric data with the VAL function, but this demands the same strict format as INPUT. A much more powerful and flexible way to extract numeric data from a string is by using the ENTER statement. Admittedly, it is not likely that an operator would enter extra text with the number—but why generate an error if he does? The LINPUT/ENTER combination can extract the month from responses like these:

```
4
"4"
MONTH=4
4th month
```

If a number is found, the error trap is disabled. In actual applications, the OFF ERROR statement would be replaced by an ON ERROR statement that re-establishes the normal error trapping used in the program. The final check ensures that the month is within a meaningful range. You want to give the operator maximum flexibility, but accepting the 54th month is **too** flexible. Range checking is a technique that should be used in all good operator interfaces.

Although ENTER can do a lot, it cannot extract a number from a string that has no numerals. Since the operator is permitted (and encouraged) to use the name of the month, the program must handle this case. That is the purpose of the ON ERROR statement before the ENTER. If the ENTER cannot find any numeric value, the error trap directs program execution to the segment labeled `String`. This segment changes the error trap, since it has served its purpose. Then the input data is searched for the presence of a month name. A string comparison could be used, but that requires that the month name be in a fixed location within the response. Again, there is no reason for such a restriction. The POS function will find the desired letters anywhere in the line. The UPC$ function eliminates any requirements about letter case. Thus, responses like the following would all be valid:

```
JAN
January
MONTH=JAN
"january"
```

In any keyboard-input situation, there is always some possibility that the operator entered pure garbage. If all the attempts to find a meaningful number or name fail, an error message is displayed, and the entire process is repeated. Another programming choice is to assume the default if no meaningful input is found. You must judge for yourself which choice is best. If accurate operator input is very important to the program, then the program should keep asking until the operator gets smart. If the value in question is not important, it might be best the assume a default and move on to the next stage of the program.

Note that the desired variable, Current_month, is not updated unless a valid input was received. All the testing and searching is done using a temporary variable. This is done so that the default value is not destroyed by an invalid input.

LINPUT with Multiple Fields

This example requests the entire date: day, month, and year. As in the previous example, there is nothing special about dates. The techniques shown have general applications. A detailed discussion follows the listing.

```
100   OPTION BASE 1
110   DIM In$[160],Months$(12)[3],Left$[2]
120   INTEGER Temp,Current_day,Current_month,Current_year
130 Fmt:  IMAGE #,2D,",",3A,",",K,K      ! Format of date input
140   FOR Temp=1 TO 12
150     READ Months$(Temp)               ! String data for month names
160   NEXT Temp
170   DATA JAN,FEB,MAR,APR,MAY,JUN,JUL,AUG,SEP,OCT,NOV,DEC
180   Left$=CHR$(255)&CHR$(72)           ! Moves cursor to beginning of line
190   Current_day=1                      ! Set up default values...
200   Current_month=11                   ! In real applications, these might
210   Current_year=1982                  ! come from the clock or a file.
220   !
230 Get_date:   !
240   OUTPUT KBD USING Fmt;Current_day,Months$(Current_month),
Current_year,Left$
250   LINPUT "Enter the date, using this format.",In$
260   ON ERROR GOTO Not_valid            ! No numerals = error for ENTER
270   ENTER In$;Temp                     ! Extract the day
280   OFF ERROR                          ! ENTER worked; change error trap
290   IF Temp<1 OR Temp>31 THEN Not_valid  ! Check for impossible day-of-month
300   Current_day=Temp                   ! Value OK; use it
310   !
320   Temp=POS(In$,",")                  ! Look for first delimiter
330   IF NOT Temp THEN Not_valid         ! No delimiter = bad format
340   In$=UPC$(In$[Temp+1])              ! Remove date field; make uppercase
350   FOR Temp=1 TO 12                   ! Try to find 1st three letters
360     IF POS(In$,Months$(Temp)) THEN Found_month
370   NEXT Temp
380   !
390 Not_valid:   !
400   OFF ERROR                          ! Change ENTER error trapping
410   BEEP
420   DISP "Improper entry. Please try again."
430   WAIT 2
440   GOTO Get_date                      ! Start over with this routine
450   !
460 Found_month:   !
470   Current_month=Temp                 ! Value OK; use it
```

```
480   ON ERROR GOTO Not_valid              ! No numerals = error for ENTER
490   ENTER In$;Temp                       ! Extract the year
500   OFF ERROR                            ! ENTER worked; change error trap
510   IF Temp<100 THEN Temp=Temp+1900      ! Maybe there is no century?
520   Current_year=Temp                    ! Value OK; use it
530   !
540   ! Program execution continues here
```

The first segment declares the variables, stores the month abbreviations, establishes some defaults, and contains an IMAGE statement that specifies the desired date format. Although defaults are important, program constants are not always the best way to supply defaults. Using the constant "12" as a default for a GPIO interface select code makes sense. But the date will almost always be different from a constant stored in the program. A real program should adopt some other method of assuming the date. If your computer has a battery-backed real-time clock, the date might be extracted from the clock value. If the program uses a file with the date stored in it, the last access date might be close to the current date.

A significant feature of this example is the handling of multiple fields. Multiple fields bring with them two special considerations. First, there is the need to show the operator the proper format for the fields. Second, there is the need to extract those fields from a single string, assuming that LINPUT is used.

The proper format for the fields is shown to the operator by using an OUTPUT to the keyboard. The default values are sent to the keyboard line, formatted by an IMAGE statement. This not only gives the operator the choice of simply pressing CONTINUE (f2 on an ITF keyboard), but it also shows the appearance of a correct response. If the default date is generated by a good source, it is reasonable to expect that the "day" field will be changed more often than the month or year. Therefore, the OUTPUT to the keyboard finishes by placing the cursor at the beginning of the line, in the day field.

The ON ERROR/ENTER technique is similar to the previous example. The ENTER statement extracts only the day because the comma terminates that field. The day is checked against resonable limits and assigned to the actual variable if it is acceptable. This range checking could be expanded to check for the maximum day allowed in a specific month.

After the day is extracted, the string is searched for the comma delimiter, and the day field is removed. This is done to prevent the day number from interfering with the extraction of the year number. The resulting string is searched for the month name using the same technique as the previous example.

The year is extracted using the ENTER technique. If a valid number is found, one last test is performed. The response might have contained only the last two digits of the year. This is not likely, since the recommended format showed all four digits; but why complain if it happens? If only two digits are found, the program supplies the 1900 automatically. By the way, this technique is not too effective if the dates being entered might cross century boundaries.

Yes and No Questions

Frequently, all the computer needs from the operator is a simple "yes" or "no." The "Expanded Softkey Menu" example showed one way to handle yes/no states. However, that much processing is not always desired. If you only need to ask a single question, why program 10 softkeys and 18 CRT lines? The following user-defined function shows some simple, but friendly, processing for yes/no answers.

The objective of this routine is to provide as much flexibility as possible. This means that we don't bother the operator about such things as bumping the space bar, pressing CAPS LOCK (Caps on the ITF keyboard), or responding with a simple CONTINUE (f2 on the ITF keyboard). The main program provides a prompt or explanation and performs a LINPUT with a 256-character string. It then passes that string to this function and tests the results.

The function uses a local copy of the string just in case you need the actual input for some other purpose in the main program. The response is trimmed and placed in uppercase. Then the first letter is tested. Four cases are identified: the answer was "Y" (for yes), the answer was "N" (for no), no answer was given, or the answer was not recognized.

```
2000  DEF FNYes(X$)
2010    DIM Temp$[1]
2020    Temp$[1,1]=TRIM$(X$)
2030    SELECT Temp$
2040    CASE "Y","y"
2050      RETURN 1
2060    CASE "N","n"
2070      RETURN 0
2080    CASE " "
2090      RETURN -1
2100    CASE ELSE
2110      RETURN -2
2120    END SELECT
2130  FNEND
```

As mentioned previously, every question should have a default answer. The default answer for a yes/no question depends greatly upon the nature of the question. If you are asking the operator for permission to use standard, reasonable parameters for an operation, then "yes" is a helpful default. If you are asking for permission to initialize a disc and destroy all files, then the default answer had better be "NO"! When a question or choice occurs more than once in a program, it is usually a good technique to use the operator's previous response as the default. Put yourself in the user's place and think about how the program should run.

To use this function to best advantage, the result must be tested thoughtfully. If the operator simply presses CONTINUE (f2 on the ITF keyboard), the result will be −1. Therefore, the default should be assumed if `FNYes=-1`. A "yes" answer is indicated by `FNYes=1`; whereas a **non-negative** answer can be tested simply as `IF FNYes`. A **non-affirmative** answer is `FNYes<1`. Any result less than zero is a noncommittal reply. Perhaps the default could be assumed for any negative result, or perhaps a negative result should cause the question to be repeated. The test `IF NOT FNYes` reveals a negative reply. As you can see, many shades of interpretation are possible.

Example Human Interfaces

This section puts together some of the techniques discussed in preceding sections.

An Expanded Softkey Menu

A good human interface often involves the coordination of multiple resources. The softkeys are a very good tool for accepting operator input. The biggest problem with using softkeys is the severe limitation on the number of prompt characters associated with each key. Therefore, a softkey interface is an appropriate task to demonstrate the increased use of CRT space.

The goal of this technique is to display a readable and informative menu that monitors the operator's input. The following program segment displays a summary of the parameters that are controlled by softkeys. This summary is updated every time a softkey is pressed, providing immediate feedback to the operator. This example uses many of the CRT-control techniques already presented. It also helps to show why the human interface of a program can require so much code. This segment simply logs the operator's choice of four items, and it is over 100 lines long. The purpose of each section of code is explained after the listing.

```
1000  DIM Disc$[5],Clear$[2],Home$[2],Cmd$[1]
1010  INTEGER Std_fmt,Roman,Screen,Center
1020  !
1030  Clear$=CHR$(255)&CHR$(75)       ! CLEAR SCR key
1040  Home$=CHR$(255)&CHR$(84)        ! HOME key
1050  Disc$="RIGHT"                   ! Default parameters
1060  Cmd$="\"
1070  Std_fmt=1
1080  Roman=0
1090  STATUS 1,9;Screen               ! Get screen width
1100  Center=(Screen-36)/2            ! Leading spaces for centering
1110  MASS STORAGE IS ":INTERNAL"
1120  PRINTER IS 1                    ! Use CRT for displaying menu
1130  GRAPHICS OFF
1140  CONTROL 2,1;0                   ! PRT ALL off
1150  CONTROL 1,4;0                   ! DISPLAY FCTNS off
1160  OUTPUT KBD;Clear$;                ! Clear CRT
1170  !
1180 Menu:  !
1190  OUTPUT KBD;Home$;                 ! Home display
1200  PRINT TABXY(1,1)                ! Start at top with blank line
1210  PRINT TAB(Center);"KEY   PURPOSE";TAB(Center+30);"VALUE"
1220  PRINT TAB(Center);"------------------------------------"
1230  PRINT
1240  PRINT TAB(Center);" 5    Command Delimiter";TAB(Center+31);Cmd$
1250  PRINT
```

```
1260  PRINT TAB(Center);" 6    Source Disc Drive";TAB(Center+30);Disc$
1270  PRINT
1280  PRINT TAB(Center);" 7    Standard Format OK?";TAB(Center+30);
1290  IF Std_fmt THEN
1300    PRINT "YES"
1310  ELSE
1320    PRINT "NO "
1330  END IF
1340  PRINT
1350  PRINT TAB(Center);" 8    Use Roman Numerals?";TAB(Center+30);
1360  IF Roman THEN
1370    PRINT "YES"
1380  ELSE
1390    PRINT "NO "
1400  END IF
1410  PRINT
1420  PRINT TAB(Center);" 9    START PRINTOUT"
1430  !
1440  IF Screen=50 THEN               ! Use short labels
1450    ON KEY 5 LABEL " Delim  " GOTO Command
1460    ON KEY 6 LABEL "  Disc  " GOTO Drive
1470    ON KEY 7 LABEL " Format " GOTO Standard
1480    ON KEY 8 LABEL " Roman  " GOTO Numbers
1490    ON KEY 9 LABEL " START  " GOTO Begin
1500  ELSE                            ! Use long labels
1510    ON KEY 5 LABEL "Command Delim " GOTO Command
1520    ON KEY 6 LABEL " Select Drive " GOTO Drive
1530    ON KEY 7 LABEL " Stand. Fmt.? " GOTO Standard
1540    ON KEY 8 LABEL "Roman Numeral?" GOTO Numbers
1550    ON KEY 9 LABEL " START PRINT  " GOTO Begin
1560  END IF
1570  ON KEY 0 GOTO Not_used          ! Turn off unused keys
1580  ON KEY 1 GOTO Not_used
1590  ON KEY 2 GOTO Not_used
1600  ON KEY 3 GOTO Not_used
1610  ON KEY 4 GOTO Not_used
1620  !
1630 Spin:  GOTO Spin                 ! Wait for softkey interrupt
1640  !
1650 Not_used:  !
1660  BEEP 300,.1                     ! Feedback for unused keys
1670  GOTO Spin
1680  !
1690 Command:  !
1700  IF Cmd$="\" THEN                ! Choose command delimiter
1710    Cmd$="^"
1720  ELSE
1730    Cmd$="\"
1740  END IF
1750  GOTO Menu
```

```
1760  !
1770 Drive:  !
1780  IF Disc$="RIGHT" THEN           ! Choose text source
1790    MASS STORAGE IS ":INTERNAL,4,1"
1800    Disc$="LEFT "
1810  ELSE
1820    MASS STORAGE IS ":INTERNAL,4,0"
1830    Disc$="RIGHT"
1840  END IF
1850  GOTO Menu
1860  !
1870 Standard:  !
1880  IF Std_fmt THEN                 ! Choose text format
1890    Std_fmt=0
1900  ELSE
1910    Std_fmt=1
1920  END IF
1930  GOTO Menu
1940  !
1950 Numbers:  !
1960  IF Roman THEN                   ! Choose numeral type
1970    Roman=0
1980  ELSE
1990    Roman=1
2000  END IF
2010  GOTO Menu
2020  !
2030 Begin:  !
2040  OUTPUT KBD;Clear$;               ! Clear CRT
2050  OFF KEY                         ! Remove selection menu
2060  !
2070  ! Program continues here when user presses "START"
2080  !
```

The program uses softkeys 5 through 9. If you have an ITF keyboard, your softkeys are labeled 1 through 8. You can modify the program to use the softkeys most useful for your applications.

It is always good programming practice to declare all variables. The first two lines do this. Next, the variables are given their starting values. Initialization is completed by turning off unwanted modes and clearing the CRT.

The section at "Menu" displays a description and current status for each menu item. This example shows some of the parameters that might be used by a simple text-printing program. The items used are representative only. A real text formatter would have many more parameters (all the more reason to present them clearly). The operator can choose the following:

- Back-slash or up-caret as a command delimiter
- Right or left disc drive for the source of the text
- Standard or alternate format for the text
- Page numbering with Arabic or Roman numerals

Notice some important aspects of this menu. All items have default values and all defaults are visible simultaneously. This is very important. It is irritating and confusing when an operator must answer question after question to get a program to begin. It is far better to show the default environment and allow a single keypress to start the program if the defaults are acceptable. If any defaults need to be changed, the operator changes only those items he wants to change. He can press "START" at any time, and in this simple case, never answers any questions. The operator wants a printout, not a game of "20 questions."

The current state of all items is displayed in a form that is meaningful to the operator. It is reasonably safe to assume that all operators know what "RIGHT" and "LEFT" mean. Very few would have any idea what ":INTERNAL,4,1" means. Programmers need to learn about concepts like "mass storage unit specifier." Operators shouldn't be bothered by such things. Likewise, don't expect anyone to answer "1" or "0" to a question that should be answered "YES" or "NO."

A more technical aspect of this menu is the method used to update the display. Since the scrolling keys are on one side of the softkeys and the knob is on the other side (of 98203 keyboards), it is reasonable to assume that the operator might accidentally move the display out of place. One way to correct this would be to start each display update with a CLEAR SCREEN statement. This guarantees the state of the CRT and the print position. Unfortunately, it also causes a very undesirable "blinking off" of the display each time a key is pressed. A constantly disappearing menu is very distracting. The BASIC system now has the capability of disabling scrolling. See the discussion near the beginning of this chapter for details.

The objective is to give the impression that nothing changed except the selected item. Therefore, the "clear" sequence is sent before the first display only. Subsequent updates use a "home" sequence to ensure the position of the text, and a TABXY to set the print position. As a result, the new menu is written on top of the old menu. (The same visual effect could be achieved by using individual TABXY functions to access each item display, but that is a more difficult program to write.)

Since the old display is overwritten each time, it is important to erase all unneeded characters. Notice that the "NO" displays are padded with a trailing blank to erase the "S" left over from "YES." This technique can be extended to clear old displays of unknown length. The following example displays a number and erases any remaining digits from the old number. The variable `Screen` contains the screen width.

```
1300  PRINT Value;TAB(Screen)
```

The example also uses screen width for centering. Centering is not as important as keeping the display properly updated, and centering slows down the update process slightly. However, the technique is shown here in case you want to use it. During the initialization of variables, the current screen width is determined. This might be 50 80 or 128 characters if the program is used on different models of computers. The width of the menu display is subtracted from the screen width to determine the amount of left-over space. If half of this space is sent at the beginning of the line, the remaining half will be at the end of the line. This produces a centered display. The amount to be sent at the beginning of the line is placed in the variable `Center`. This value is used to position the start of each line and is also used as a reference point to position the second column.

Models with ITF keyboards allow 16 characters (2 rows of 8) in a softkey label. Models with 80-column CRTs allow 14 characters in a softkey label. Models with 50-column CRTs allow only 8 characters for these labels. Therefore, the variable `Screen` is also used to control the display of softkey labels. This is the purpose of the segment at line 1440. The alternative is to restrict all softkey labels to 8 characters. This is possible, but undesirable. It is difficult to say anything meaningful in 8 characters. Users with 80-column CRTs will appreciate the extra meaning that is available with longer labels. The 128-column CRT can use longer labels, but this program uses the 14-character labels.

The ON KEY statements for keys 0 through 4 are used to turn off any typing-aid definitions that might exist for those keys. An ON KEY definition overrides a typing-aid definition when the program is running. However, if no ON KEY definition is supplied, the typing-aid definition remains active. This is not desirable when you are trying to achieve a program-controlled softkey menu. Therefore, the unused keys are given a "dummy" ON KEY definition to keep the menu clean. For ITF keyboards, you should "turn off" all 24 softkeys.

Notice also that when five or less softkeys are used, keys 5 through 9 are defined. This is to accommodate the Model 216 small keyboard. On the small keyboard, those are the unshifted keys. Why make the operator press the shift key? If you have an ITF keyboard, use keys 1 through 5.

The softkeys are defined to send program execution to a parameter-changing routine. Each such routine ends by sending program execution to the display-update routine. In this example, there is no demonstrated reason for repeating the ON KEY definitions for every keypress. Those definitions could have been placed above the "Menu" line and executed only once. However, some applications might need to change the key definitions in response to changes in program variables. For example, a key that produces an "insert" operation would be disabled when enough inserts had been performed to fill an array. Also, it is possible to include the value of a string variable in a key label. Therefore, the key labels may need to be rewritten as new selections are made. In cases like these, the ON KEY statements need to be in the update path.

The final "cleanup" action takes place when the operator presses "START." This is the signal that the selection menu is no longer needed. The menu display is cleared to reflect the fact that it is no longer in use. The OFF KEY statement performs two functions[1]. It turns off the softkey label area, which helps keep the CRT neat. More importantly, it cancels all the ON KEY branches. If this were not done, the operator could cause the program to jump back to the selection menu at any time. This is probably not desirable. You may want to define some sort of "Abort" key that lets the operator stop a lengthy operation. But it is not likely that the selection menu would be the destination of an abort operation. Remember, ON KEY definitions stay around forever unless you turn them off or the program stops.

[1] This example is intended for use on an HP 98203 keyboard. For an ITF keyboard, the default conditions for key labels are slightly different. In this case, you can use KEY LABELS OFF to turn the softkey labels off, and KEY LABELS ON to re-enable displaying them. You can alternatively use CRT control register 12 to set the key-labels display mode to match the behavior of the 98203 keyboard. See the *BASIC Language Reference* description of the KEY LABELS statement for details.

Not much has been said about the parameter-changing routines. The examples shown use a simple IF...THEN...ELSE structure to select between to alternatives. This concept can be expanded to allow selection of more than two choices. The MOD function is handy when you want to cycle through several choices. The following example shows a routine that rotates through four choices. This routine is intended to fit into our menu selection process. Accent protocols for different languages are shown here, but the technique is applicable to any selection item.

```
1910 Accents:  !
1920  Lang=(Lang+1) MOD 4                 ! Choose accent protocol
1930  SELECT Lang
1940  CASE 0
1950    Language$="ENGLISH"
1960  CASE 1
1970    Language$="FRENCH "
1980  CASE 2
1990    Language$="SPANISH"
2000  CASE 3
2010    Language$="GERMAN "
2020  END SELECT
2030  GOTO Menu
```

Moving a Pointer

Many programs have a main menu from which the operator chooses a subtask. An example might be an editing program that gives the choice of getting a file, storing a file, editing a file, merging files, listing a file, protecting a file, deleting a file, etc. As with all other tasks, there are many ways to present this choice to the operator. Each task might be assigned to a softkey. The ON KBD statement might be used to equate individual keys to each task. For example, E for edit, M for merge, G for get, and so on. Depending on the application, one of these methods may be good. However, there are some considerations. There might be more choices than softkeys, or the arrangement of the softkeys might be awkward. The single-letter method is always just a little "dangerous". What if the operator tries to type a word? Did "P" stand for "protect" or "purge"?

One alternative is to display all the choices, with a pointer to the current selection. When the operator is sure that the selection is proper, a single press of a softkey tells the computer "Do it." The menu choices can be full phrases with no abbreviations, since the whole CRT is available for the display. The pointer can be moved by softkeys or by the knob. Since we just discussed the softkeys, let's use the knob for this example.

The following example clears the CRT, displays seven selections, and allows the knob to cycle a pointer through the selections in either direction. In a real application, meaningful phrases would be used to identify the selections, and a softkey would be defined to start the selected process. Softkeys could also be used to move the pointer up and down. This could be in addition to the knob or in place of it. A detailed discussion follows the listing.

```
100   DIM Marker$[4],Home$[2],Clear$[2]
110   INTEGER Point
120   !
130   Clear$=CHR$(255)&CHR$(75)        ! CLEAR SCR key
140   Home$=CHR$(255)&CHR$(84)         ! HOME key
150   Marker$="=>"&CHR$(8)&CHR$(8)     ! Pointer arrow
160   Point=1                          ! Default selection
170   PRINTER IS 1                     ! Use CRT for menu display
180   GRAPHICS OFF
190   CONTROL 2,1;0                    ! PRT ALL off
200   CONTROL 1,4;0                    ! DISPLAY FCTNS off
210   OUTPUT KBD;Clear$;                 ! Clear CRT
220   !
230   PRINT "Use shift and knob to move marker"
240   PRINT "   Selection 1"             ! Display menu
250   PRINT "   Selection 2"
260   PRINT "   Selection 3"
270   PRINT "   Selection 4"
280   PRINT "   Selection 5"
290   PRINT "   Selection 6"
300   PRINT "   Selection 7"
310   PRINT TABXY(1,Point);Marker$;    ! Display starting marker
320   !
330   ON KNOB .2 GOTO Move_pointer     ! Enable knob
340 Spin:   GOTO Spin                  ! Wait for knob interrupt
350   !
360 Move_pointer:   !
370   IF KNOBY>0 THEN                  ! Check knob direction
380     Point=Point+1
390   ELSE
400     Point=Point-1
410   END IF
420   IF Point<1 THEN Point=7          ! Keep pointer within limits
430   IF Point>7 THEN Point=1
440   OUTPUT KBD;Home$;                  ! Home the display
450   PRINT "  ";                      ! Erase old marker
460   PRINT TABXY(1,Point);Marker$;    ! Display new marker
470   GOTO Spin
480   !
490   END
```

The program starts by declaring and initializing the variables. The "clear" and "home" sequences should look familiar to you by now. The **Marker$** string is a contrived arrow followed by two backspace characters. The backspace characters return the print position to the beginning of the arrow each time it is displayed. This facilitates the erase operation that is part of moving the arrow.

After the display is cleared, the menu selections are printed. This is done only once, since the choices do not include any changing parameters. The TABXY function is used to position a marker to the left of the default selection. Then the knob is enabled, and the program sits in an idle loop waiting for an interrupt from the knob.

When the knob is turned, program execution branches to the pointer-moving routine. In this example, the amount of knob movement is not used, only its direction is extracted from the KNOBY function. It is possible to add an algorithm that accumulates the counts from the knob so that a fixed amount of rotation is needed to move the pointer. Such an improvement would give a more positive "linkage" between the knob and the display, but is not necessary to this demonstration.

The pointer value is stored in the variable **Point**. This variable is increased or decreased depending upon the direction of knob rotation. After the variable is updated, it is necessary to keep it within the limits of the available selections. The option used here was to "wrap around" when the pointer reached either end of the list. Another option is to "freeze" the pointer when it reaches an end position. To do this, lines 420 and 430 would be modified as follows:

```
420   IF Point<1 THEN Point=1
430   IF Point>7 THEN Point=7
```

After the pointer value is updated, the display must be changed to reflect the new value. First, the display is returned to home position. Although the knob no longer scrolls the display, the scrolling keys are still active. They may have been pressed (perhaps accidentally) and moved the display out of position[1]. Since the print position is always at the beginning of the old pointer, that pointer can be erased by printing two blanks. The new pointer is then printed using a TABXY function. Notice that end-of-line sequences are not needed or desired. All the PRINT statements used in this updating process use a trailing semicolon to suppress the EOL sequence.

[1] See the discussion of "Disabling and Enabling Scrolling" near the beginning of this chapter for a method of preventing this problem.

In this example, the x-coordinate was always 1. If needed, the x-coordinate is available in the TABXY function to work with multi-column displays.

Assumed, but not shown, is an ON KEY statement that would start the selected process. This key would branch to a routine that cleared the display, turned off the knob, and used the variable `Point` in a SELECT or ON statement to access the chosen routine.

An Example Custom Keyboard Interface

An example subprogram called `Kled` that implements a custom keyboard interface is provided on the *BASIC Utilities* disc in the "MEM_UTILS" file. It enables a branch to an interrupt service routine for any keystroke using the ON KBD mechanism[1]. When a branch is initiated, it traps the key codes (including "system key" codes) with the KBD$ function, and then initiates corresponding actions. Note that the SYSTEM$("KBD LINE") function allows you to use the BASIC system's keyboard-input editing features with OUTPUT to the keyboard (select code 2).

The MEM_UTILS program also shows how to combine typing-aids and memory volumes to create memory resident utility programs.

[1] Since the ON KBD statement is described fully in the "Keyboard Interfaces" chapter of *BASIC Interfacing Techniques*, it will not be described here.

Handling Errors

11

Handling Errors 11

Most programs are subject to errors at run time, even if all the typographical/syntactical errors have been removed in the process of entering the program into the computer in the first place. This chapter describes how BASIC programs can respond to these errors, and shows how to write programs that attempt to either correct the problem or direct the program user to take some sort of corrective action.

Overview of Error Responses

There are three courses of action that you may choose to take with respect to errors:

1. Try to prevent the error from happening in the first place (by communicating clearly with the program user, by using range-checking routines, and so forth).
2. Once an error occurs, try to recover from it and continue execution (this involves the BASIC program trapping and correcting errors).
3. Do nothing—let the system stop the program when an error happens.

The remainder of this chapter describes how to implement the first two alternatives.

The last alternative, which may seem frivolous at first glance, is certainly the easiest to implement, and the nature of HP Series 200/300 BASIC is such that this is often a feasible choice. Upon encountering a run-time error, the computer will pause program execution and display a message giving the error number and the line in which the error happened, and the programmer can then examine the program in light of this information and fix things up. The key word here is "programmer." If the person running the program is also the person who wrote the program, this approach works fine. If the person running the program did not write it, or worse yet, does not know how to program, some attempt should be made to prevent errors from happening in the first place, or to recover from errors and continue running.

Anticipating Operator Errors

When a programmer writes a program, he or she knows exactly what the program is expected to do, and what kinds of inputs make sense for the problem. Given this viewpoint, there is a strong tendency for the programmer not to take into account the possibility that other people using the program might **not** understand the boundary conditions. A programmer has no choice but to assume that every time a user has the opportunity to feed an input to a program, a mistake can be made and an error can be caused. If the programmer's outlook is noble, he or she will try to save the user from needless anguish and frustration. Even if the programmer's outlook is less altrusitic, he or she will try to keep from getting involved in future support problems. In either case, an effort must be made to make the program more resistant to errors.

Boundary Conditions

A classic example of anticipating an operator error is the "division by zero" situation. An INPUT statement is used to get the value for a variable, and the variable is used as a divisor later in the program. If the operator should happen to enter a zero, accidentally or intentionally, the program pauses with an error 31. It is far better to be watching for an out-of-range input and respond gracefully. One method is shown in the following example.

```
100  INPUT "Miles traveled and total hours",Miles,Hours
110  IF Hours=0 THEN
120    BEEP
130    PRINT "Improper value entered for hours."
140    PRINT "Try again!"
150    GOTO 100
160  END IF
170  Mph=Miles/Hours
```

Consider another simple example of giving a user the choice of six colors for a certain bar graph. It might be preferable to have the user pick a number corresponding to the color he wished to choose instead of having to type in up to six characters. In this case, the program wouldn't have to check for each number, but rather it could use the logical comparators to check for an entire range:

```
4030  CLEAR SCREEN
4040  DATA GREEN,BLUE,RED,YELLOW,PURPLE,PINK
4050  ALLOCATE Colors$(1:6)[6]
4060  READ Colors$(*)
4070  FOR I=1 TO 6
4080     PRINT USING "DD,X,K";I,Colors$(I)
4090  NEXT I
4100 Ask: INPUT "Pick the number of a color",I
```

```
4110  IF I>=1 AND I<=6 THEN Valid_Color
4140  BEEP
4150  DISP "Invalid answer -- ";
4160  WAIT 1
4170  GOTO Ask
```

The above example needs a little extra safeguarding. I, the variable being input, should be declared to be an integer, since the only valid inputs are 1, 2, 3, 4, 5, and 6. An answer like "pick the 3.14th color listed" does not make sense.

Real number boundaries are tested for in a manner similar to that of integers:

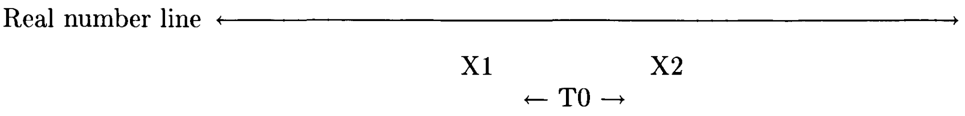
```
7010  INPUT "Enter the waveform's frequency (in KHz)",Frequency
7020  IF Frequency<=0 THEN 7010
7030  INPUT "Enter the amplitude (0-10 volts)",Amplitude
7040  IF Amplitude<0 OR Amplitude>10 THEN 7030
7050  INPUT "Enter the phase angle (in degrees)",Angle
7060  IF Angle<0 OR Angle>180 THEN 7050
7070  Angle=Angle*PI/180
```

REAL and COMPLEX Numbers and Comparisons

A word of caution is in order about the use of the = comparator in conjunction with REAL and COMPLEX (full-precision) numbers. Numbers on this computer are stored in a binary form, which means that the information stored is not guaranteed to be an exact representation of a decimal number—but it will be very close! What this means is that a program should not use the = comparator in an IF statement where the comparison is being performed on REAL or COMPLEX numbers. The comparison will yield a 'false' or '0' value if the two are different by even one bit, even though the two numbers might really be equal for all practical purposes.

There are two ways around this problem. The first is to try to state the comparison in terms of the <= or >= comparators.

If it is necessary to do an equality comparison with a pair of REAL numbers, then the second method must be used. This involves picking an error tolerance for how close to being equal the two numbers can be to satisfy the test.



So if the difference between two REAL numbers X1 and X2 is less than or equal to a tolerance T0, we'll say that X1 and X2 are "equal" to each other for all practical purposes. The value of T0 will depend upon the application, and must be chosen with care.

For an example, assume that we've picked a tolerance of 10^{-12} for comparing two real numbers for equality. The proper way to compare the two numbers would be:

```
950 IF ABS(X1-X2)<=1E-12 THEN Numbers_equal
960 ! Otherwise they're not equal
```

Another technique for comparing two REAL or COMPLEX values is to use the DROUND function. This is especially suited to applications where the data is known to have a certain number of significant digits. For more details on binary representations of decimal numbers, refer to the "Numeric Computation" chapter.

Note that >=, <=, and DROUND **do not** work with COMPLEX numbers, but you can compare real parts and imaginary parts. For example, comparing two COMPLEX values for equality would require something like this:

```
IF (ABS(REAL(C1)-REAL(C2)) <= 1E-12) AND
   (ABS(IMAG(C1)-IMAG(C2)) <= 1E-12) THEN ...
```

Trapping Errors with BASIC Programs

Despite the programmer's best efforts at screening the user's inputs in order to avoid errors, errors will still happen occasionally. It is possible to recover from run-time errors, provided the programmer predicts the places where errors are most likely to happen and adequately handles the error cause(s).

Setting Up Error Service Routines (ON/OFF ERROR)

The ON ERROR statement sets up a branching condition which will be taken any time a recoverable error is encountered at run time. Here are some example statements (further examples of each type of branch—GOSUB, GOTO, etc.—are given in subsequent sections).

```
100  ON ERROR GOSUB Fix_it         400 Fix_it: ! Solve problem.
                                             .
                                             .
                                   530  RETURN ! If GOSUB used.



100  ON ERROR GOTO Fix_it
100  ON ERROR RECOVER Fix_it       400 Fix_it: ! Solve problem.



200  ON ERROR CALL Fix_it_sub      800 SUB Fix_it_sub
                                             .
                                             .
                                   950 SUBEND
```

Choosing a Branch Type

The type of branch that you choose (GOTO vs. GOSUB, etc.) depends on how you want to handle the error.

- Using GOSUB indicates that you want to return to the statement that caused the error (RETURN) or to the one following the statement that caused the error (ERROR RETURN) when finished with your attempt to correct the error's cause.
- GOTO, on the other hand, may indicate that you do not want to re-attempt the operation after attempting to correct the source of the error.

Scope of Error Trapping and Recovery

GOTO and GOSUB are purely local in scope—that is, they are active only within the context in which the ON ERROR is declared. CALL and RECOVER are global in scope—after the ON ERROR is set up, the CALL or RECOVER will be executed any time an error occurs, regardless of subprogram environment.

ON ERROR Execution at Run-Time

When an ON ERROR statement is executed, the BASIC system will make sure that the specified line or subprogram exists in memory before the program will proceed.

- If GOTO, GOSUB, or RECOVER is specified, then the *line identifier* must exist in the current context (at pre-run).
- If CALL is used, then the specified *subprogram* must currently be in memory (at run-time).

In either case, if the system can't find the given line, error 49 is reported.

ON ERROR Priority

ON ERROR has a priority of 17, which means that it will *always* take priority over any other ON *event* branch, since the highest user-specifiable priority is 15.

Disabling Error Trapping (OFF ERROR)

The OFF ERROR statement will cancel the effects of the ON ERROR statement, and no branching will take place if an error is encountered.

The DISABLE statement has no effect on ON ERROR branching.

Determining Error Number and Location (ERRN, ERRLN, ERRL, ERRDS, ERRM$)

ERRN is a function which returns the error number which caused the branch to be taken. ERRN is a global function, meaning it can be used from the main program or from any subprogram, and it will always return the number of the most recent error.

```
100  IF ERRN=80 THEN ! Media not present in drive.
110      PRINT "Please insert the 'Examples' disc,"
120      PRINT "and press the 'Continue' key (f2)."
130      PAUSE
140  END IF
```

ERRLN is a function which returns the *line number* of the program line in which the most recent error has occurred.

```
100  IF ERRLN<1280 THEN GOSUB During_init
110  IF (ERRLN>=1280 AND ERRLN<=2440) THEN GOSUB During_main
120  IF ERRLN>2440 THEN GOSUB During_Last
```

You can use this function, for instance, in determining what sort of situation-dependent action to take. As in the above example, you may want to take a certain action if the error occurred while "initializing" your program, another if during the "main" segment of your program, and yet another if during the "last" part of the program.

ERRL is another function which is used to find the line in which the error was encountered; however, the difference between this and the ERRLN function is that ERRL is a *boolean* function. The program gives it a line identifier, and either a 1 or a 0 is returned, depending upon whether or not the specified identifier indicates the line which caused the error.

```
100  IF ERRL(1250) OR ERRL(1270) THEN GOSUB Fix_12xx
110  IF ERRL(1470) THEN GOSUB Fix_1470
120  IF ERRL(2450) OR ERRL(2530) THEN GOSUB Fix_24xx
```

ERRL is a *local* function, which means it can only be used in the same environment as the line which caused the error. This implies that ERRL *cannot* be used in conjunction with ON ERROR CALL, but it *can* be used with ON ERROR GOTO and ON ERROR GOSUB. ERRL can be used with ON ERROR RECOVER only if the error did not occur in a subprogram which was called by the environment which set up the ON ERROR RECOVER.

The ERRL function will accept either a *line number* or a *line label*:

```
1140  DISP ERRL(710)
```

```
910   IF ERRL(Compute) THEN Fix_compute
```

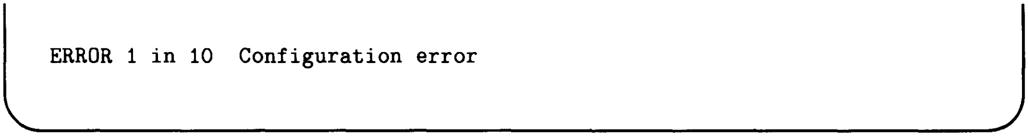
ERRDS returns the *device selector* of the device which was involved in the last error. For instance:

```
IF ERRDS=12 THEN GOSUB Fix_gpio
```

Note that this function is *only* updated when an error that involves an interface or device occurs; otherwise, it remains unchanged until another error involving a device selector occurs. Therefore, if the last error did not involve a device, then the value returned by ERRDS may be irrelevant to the current situation.

ERRM$ is a string function which returns the text of the error which caused the branch to be taken.

```
100  DISP ERRM$ ! Display default message.
```



```
ERROR 1 in 10  Configuration error
```

A Closer Look at ON ERROR GOSUB

The ON ERROR GOSUB statement is used when you want to return to the program line where the error occurred. You have two choices of returning:

- RETURN returns program control back to *the line that caused the error*, thus indicating that you have corrected/resolved the error condition and want to *re-execute* this line.
- ERROR RETURN returns program control to the line *following* the line that caused the error, thus indicating that you have taken alternative action in the subroutine and *do not want to re-execute* the line that initially caused the error.

Note that if you do not correct the problem and subsequently use RETURN, the BASIC system will repeatedly re-execute the problem-causing line (which will result in an infinite loop between the ON ERROR GOSUB branch and the RETURN).

When an error triggers a branch as a result of an ON ERROR GOSUB statement being active, system priority is set at the highest possible level (17) until the RETURN statement is executed, at which point the system priority is restored to the value it was when the error happened.

```
100   Radical=B*B-4*A*C
110   Imaginary=0
120   ON ERROR GOSUB Esr
130   Partial=SQRT(Radical)
140   OFF ERROR
       .
       .
       .
350 Esr: IF ERRN=30 THEN
360         Imaginary=1
370         Radical=ABS(Radical)
380      ELSE
390         BEEP
400         DISP "Unexpected Error (";ERRN;")"
410         PAUSE
420      END IF
430      RETURN
```

Note

You cannot trap errors with ON ERROR while in an ON ERROR GOSUB service routine.

A Closer Look At ON ERROR GOTO

The ON ERROR GOTO statement is often more useful than ON ERROR GOSUB, especially if you are trying to service more than one error condition. However, ON ERROR GOTO does not change system priority.

As with ON ERROR GOSUB, one error service routine can be used to service all the error conditions in a given context. By testing both the ERRN (what went wrong) and the ERRLN (where it went wrong) functions, you can take proper recovery actions.

One advantage of ON ERROR GOTO is that you can use another ON ERROR statement in the service routine (which you cannot use with ON ERROR GOSUB). This technique, however, requires that you re-establish the original error service routine after correcting any errors (by re-executing the original ON ERROR GOTO statement). The disadvantage is that more programming may be necessary in order to resume execution at the appropriate point after each error service.

```
10    RESTORE
20    PRINT
30    PRINT
40    PRINT "Coefficients of quadratic equation A"
50    DATA 0,0,0
60    READ A,B,C
70    Maxreal=1.79769313486231E+308
80    Overflow=0
90 Coefficients:    !
100   INPUT "A?",A
110   IF A=0 THEN
120      DISP "Must be quadratic"
130      WAIT .5
140      GOTO Coefficients
150   END IF
160   PRINT "A=";A
170   INPUT "B?",B
180   PRINT "B=";B
190   INPUT "C?",C
200   PRINT "C=";C
210 Compute_roots:   !
220   ON ERROR GOTO Esr
230   Imaginary=0
240   Part1=-B/2.*A
250   Part2=SQR(B*B-4*A*C)/2.*A
260   IF NOT Imaginary THEN
270      Root1=Part1+Part2
280      Root2=Part1-Part2
290   END IF
300   OFF ERROR
310 Print_roots:  !
320   IF Imaginary=0 THEN
330      PRINT "Root 1 =";Root1
340      PRINT "Root 2 =";Root2
350   ELSE
360      PRINT "Root 1 =";Part1;" +";Part2;" i"
370      PRINT "Root 2 =";Part1;" -";Part2;" i"
380   END IF
390   IF Overflow THEN PRINT "OVERFLOW"
400   STOP
410 Esr:   !
420   IF ERRN=30 THEN      ! SQRT of negative number
430      Part2=SQRT(ABS(B*B-4*A*C))/2*A
440      Imaginary=1
450      Branch=1
460      GOTO 270
470   ELSE
480      IF ERRN=22 THEN  ! REAL overflow
490         Overflow=1
```

```
500         SELECT ERRLN
510         CASE 240
520            Part1=SGN(B)*SGN(A)*Maxreal
530            Branch=2
540         CASE 250
550            Part2=Maxreal
560            Branch=3
580         CASE 270
590            Root1=Maxreal*SGN(Part1)
600            Branch=4
620         CASE 280
630            Root2=Maxreal*SGN(Part1)
640            Branch=5
660            PRINT "Unexpected overflow"
670            Branch=6
680         CASE ELSE
690            DISP "Unexpected error";ERRN
700            Branch=6
710         END SELECT
720      END IF
730   END IF
740   ON Branch GOTO 270,250,260,280,290,10
750   END
```

A Closer Look At ON ERROR CALL

ON ERROR CALL is global, meaning once it is activated, the specified subprogram will be called immediately whenever an error is encountered, *regardless of the current context.* System priority is set to level 17 inside the subprogram, and remains that way until the SUBEXIT is executed, at which time the system priority will be restored to the value it was when the error happened.

As with ON ERROR GOSUB, you will generally use the ON ERROR CALL statement when you want to return to the program where the error occurred. You have two choices of return destinations:

- SUBEXIT sends program control back to *the line that caused the error*, thus indicating that you have corrected the cause of the problem and want to *re-execute* this line.
- ERROR SUBEXIT sends program control to the line *following* the line that caused the error, thus indicating that you have taken an alternative action and do *not* want to re-execute the line that initially caused the error.

Note that if you do not correct the problem and subsequently use SUBEXIT, the BASIC system will repeatedly re-execute the problem-causing line (which will result in an infinite loop between the ON ERROR CALL branch and the SUBEXIT).

Note

You cannot trap errors with ON ERROR while in an ON ERROR CALL service routine.

Cannot Pass Parameters Using ON ERROR CALL

Bear in mind that an ON...CALL statement can not pass parameters to the specified subprogram, so the only way to communicate between the environment in which the error is declared and the error service routine is through a COM block.

Using ERRLN and ERRL in Subprograms

You can use the ERRLN function in any context, and it returns the line number of the most recent error. However, the ERRL function will not work in a different environment than the one in which the ON ERROR statement is declared. For instance, the following two statements will only work in the context in which the specified lines are defined:

```
100  IF ERRL(40) THEN GOTO Fix40
100  IF ERRL(Line_label) THEN Fix_line_label
```

The line identifier argument in ERRL will be modified properly when the program is renumbered (such as explicitly by REN or implicitly by GET); however, that is not true of expressions used in comparisons with the value returned by the ERRLN function.

So when using an ON ERROR CALL, you should set things up in such a manner that the line number either doesn't matter, or can be guaranteed to always be the same one when the error occurs. This can be accomplished by declaring the ON ERROR immediately before the line in question, and immediately using OFF ERROR after it.

```
5010  ON ERROR CALL Fix_disc
5020  ASSIGN @File TO "Data_file"
5030  OFF ERROR
       .
       .
       .
7020  SUB Fix_disc
7030  SELECT ERRN
7040  CASE 80
7050      DISP "Door open -- shut it and press CONT"
7060      PAUSE
7080  CASE 83
7090      DISP "Write protected -- fix and press CONT"
7100      PAUSE
7120  CASE 85
7130      DISP "Disc not initialized -- fix and press CONT"
```

```
7140       PAUSE
7160   CASE 56
7170       DISP "Creating Data_file"
7180       CREATE BDAT "Data_file",20
7190   CASE ELSE
7200       DISP "Unexpected error ";ERRN
7210       PAUSE
7220   SUBEND
```

A Closer Look At ON ERROR RECOVER

The ON ERROR RECOVER statement sets up an immediate branch to the specified line whenever an error occurs. The line specified must be in the context of the ON...RECOVER statement. ON ERROR RECOVER is global in scope—it is active not only in the environment in which it is defined, but also in any subprograms called by the segment in which it is defined.

If an error is encountered while an ON ERROR RECOVER statement is active, the system will restore the context of the program segment which actually set up the branch, including its system priority, and will resume execution at the given line.

```
        .
        .
        .
3250   ON ERROR RECOVER Give_up
3260   CALL Model_universe
3270   DISP "Successfully completed"
3280   STOP
3290  Give_up:  DISP "Failure ";ERRN
3300   END
        .
        .
        .
```

Simulating Errors (CAUSE ERROR)

Since it is not always convenient to set up the conditions that cause errors, this BASIC system has a simple way of programmatically simulating errors. The following statement does this:

```
CAUSE ERROR Error_number
```

The parameter Error_number is the number of the error that you want to simulate (error numbers in the range 1001 through 1080 have special significance, as described later in this section.) Thus, CAUSE ERROR is useful in testing and verifying your error trapping routines.

The effects of this statement are the same as if the error were caused by real error conditions:

- The ERRN function still returns the error number (in this case, it is the value that you specified in the CAUSE ERROR statement).
- The ERRM$ function still returns the text of the corresponding error message (if the ERR binary is present).
- The ERRLN function still returns the line number at which the error occurred (in this case, the line number of the CAUSE ERROR statement).
- The ERRL function still returns a 1 when its argument is the line at which the error occurred.

Note, however, that CAUSE ERROR does *not* change ERRDS.

If CAUSE ERROR is executed from the keyboard, the appropriate error message will be reported, but none of the *program-related* error conditions are affected. (This is also true of other keyboard-related errors.)

Example of Simulating an Error

Here is an example of modifying one of the preceding examples to simulate an error. (Note that the original statement has been "commented out" so that it will be easy to put back in after the testing is finished.)

```
100   Radical=B*B-4*A*C
110   Imaginary=0
120   ON ERROR GOSUB Esr
130   CAUSE ERROR 30  ! Partial=SQRT(Radical)  ← Line modified.
140   OFF ERROR
        .
        .
        .
350 Esr: IF ERRN=30 THEN
360         Imaginary=1
370         Radical=ABS(Radical)
380      ELSE
390         BEEP
400         DISP "Unexpected Error (";ERRN;")"
410         PAUSE
420      END IF
430      RETURN
```

The error-trapping subroutine can then be tested to verify that it properly traps error 30. After this verification, you may want to modify the CAUSE ERROR line to simulate other errors that could possibly occur at that point in the program. (In this example, it is not necessary since all other errors are handled in the same manner; see lines 390 through 410.)

CAUSE ERROR and Error Numbers 1001 thru 1080

Error numbers 1001 through 1080 have been reserved to have special meaning for BASIC programs. These errors are used to simulate errors which may occur when a binary has not been loaded. The value returned by ERRN will be 1; the ERRM$ function will return either:

```
ERROR 1 in line_number  Missing binary binary_number
  or
ERROR 1 in line_number  Missing binary binary_name
```

The second message is returned with language-extension binaries (no binary name is returned with driver binaries).

Clearing Error Conditions (CLEAR ERROR)

After you have finished handling an error in a program, it is convenient to clear the indications that an error has occurred. The following statement performs this action:

```
  100  CLEAR ERROR
```

This statement has the following effects on the error functions' values:

`ERRN` Subsequently returns 0.

`ERRLN` Subsequently returns 0.

`ERRL` Subsequently returns 0 for all arguments (line identifiers) sent to it.

`ERRM$` Subsequently returns the null string (string with length 0).

`ERRDS` Is *not* affected by CLEAR ERROR.

Note that the CLEAR ERROR statement is *not* keyboard executable; it can only be executed from a running program.

Debugging Programs 12

Debugging Programs 12

The problem of debugging a program is distinct from the issues raised in the "Handling Errors" chapter. The "Handling Errors" chapter is based on the premise that the programmer is satisfied that the program works as it should, and that it then should be made as foolproof as possible. This could be construed as putting the cart before the horse—before you can make a program foolproof, you must get it to run correctly in the first place. One of the key characteristics of a "bug" is that it doesn't necessarily have to cause an error condition to occur—it only has to cause your program to give a wrong answer. This chapter deals with the methods available to diagnose problems in logic and semantics.

Naturally, the ideal way to debug a program is to write it correctly the first time through, and all programmers should strive constantly to achieve this goal. Hopefully, the techniques that have been been discussed in this manual will help you get a little closer to this goal. The practice of writing self-documenting code and designing programs in a top-down fashion should help immensely.

Aside from recommended methods of writing software, the computer itself has several features which aid in the process of debugging.

Using Live Keyboard

One of the pleasing characteristics of your computer is that its keyboard is "live" even during program execution. That is, you can issue commands to the computer while it is running a program the same way that you issue commands to it while it is idle. For instance, you can add two numbers together, examine the catalogue of the disk currently installed in the drive, list the running program to a printer, scroll the CRT alpha buffer up and down, or output a command to a function generator over HP-IB. Practically the only thing you can't do from live keyboard while a program is running is write or modify program lines, or attempt to alter the control structures of the program. (A complete list of illegal keyboard operations is given a little later on.)

Executing Commands While a Program Is Running

By way of illustration, key in the following program, press RUN (f3 in the System, User 1, and User 2 menus of an ITF keyboard), and then execute the commands shown underneath the listing.

```
10   FOR I=1 TO 1.E+6
20   NEXT I
30   END

CAT
2+2
SQR(6^2+17.2^2)
PRINT "THE QUICK BROWN FOX"
TIMEDATE
```

Using Program Variables

Now, this program will take a fair amount of time to complete (on the order of minutes), so to find out how far the program has gone, look at the value of the variable I. Type:

I Return or ENTER.

The current value of I will be displayed at the bottom of the screen.

If you don't want to wait for the program to go through all one million iterations, you can merely change the value of I by entering:

```
I=999000
```

Thus, we have seen that live keyboard can be used to examine and/or change the contents of the program's variables.

One aspect of live keyboard to be aware of is that the computer will only recognize variables that exist in the current program environment. For instance, suppose that we change our example program to call a subprogram inside the loop.

```
10    FOR I=1 TO 1.E+5
15      CALL Dummy
20    NEXT I
30    END
40    SUB Dummy
50    FOR J=1 TO 10
60    NEXT J
70    SUBEND
```

While this program is running and you try and test the variable `I` from the keyboard, chances are that you will only get a message saying that `I` doesn't exist in the current context—most of the time will be spent in the subprogram. On the other hand, if you test the value of `J`, it is highly likely that you will get an answer.

Similarly, operations like ASSIGN and ALLOCATE, which are declarative types of statements, must use variables that are already known to the current environment when they are executed from the keyboard. For example, in the following program, it is perfectly legal to perform the operation **`ASSIGN @Dvm TO *`** from the keyboard, although it is not legal to perform **`ASSIGN @File TO "DATA"`** from the keyboard.

```
1     ASSIGN @Dvm TO 724
10    FOR I=1 TO 1.E+5
20    NEXT I
30    END
```

Live keyboard operations are allowed to use variables already known by the running program. Live keyboard operations are not allowed to create variables.

Calling Subprograms

Although the GOTO and GOSUB commands are illegal from the keyboard, it is perfectly legal to call subprograms from the keyboard. The parameters that are passed must either be constants or must be variables that exist in the current context. Also, the program in memory must be able to pass pre-run without errors.

Here is an example:

```
10    FOR I=1 TO 1E5
20    NEXT I
30    END
40    SUB Gather(INTEGER X)
50    OPTION BASE 1
60    DIM A(32)
70    CREATE BDAT "File"&VAL$(X),1
80    ASSIGN @Dvm TO 724
90    ASSIGN @File TO "File"&VAL$(X)
100   OUTPUT @Dvm;"N100S"
110   ENTER @Dvm;A(*)
120   OUTPUT @File;A(*)
130   PRINT A(*),
140   SUBEND
150   DEF FNPoly(X)
160   RETURN X^3+3*X^2+3*X+X
170   FNEND
```

By executing `CALL Gather(1)` from the keyboard, the main program will be suspended while the subprogram is called, at which time a 1 record file will be opened, 32 readings will be taken from the voltmeter and stored in the file, and the readings will be printed on the screen. Then main program execution will resume where it left off.

Similarly, by executing `FNPoly(1)`, the value of the polynomial will be computed for X=1 and the answer (8) will be displayed at the bottom of the screen.

Pausing and Continuing a Program

You can also pause a program from the keyboard using the PAUSE (Break) key.

You may subsequently continue program execution:

- Press the CONTINUE key (f2 in the System menu of an ITF keyboard)
- Execute a CONT statement:

```
CONT  Return or ENTER
  or
CONT 100  Return or ENTER
  or
CONT Line_label Return or ENTER
```

Note that a program which has been edited **cannot** be continued.

Keyboard Commands Disallowed During Program Execution

Here is a list of commands which may not be executed from the keyboard while a program is running, although they may be executed from the keyboard if the computer is idle:

CHANGE
CONT
COPYLINES
DEL
EDIT
FIND
GET
LOAD
MOVELINES
RUN
SCRATCH
SCRATCH A
SCRATCH BIN
SCRATCH C
SYSBOOT

Cross References

When debugging a program, and you think that the problem may be that you misspelled a variable name, you can use the XREF command to alphabetically list all variable names. This listing will also contain the line numbers where the variables were used, to help you locate any problems caused by misspelling or using the wrong variable.

Another way of using a cross-reference listing is when you need to find every place a particular variable name is used, but the system (and therefore the FIND command) is not available. It is often advisable to generate a cross reference at the end of a hard-copy (printer) listing of a large program. This information makes finding every occurrence of a variable much easier.

Generating a Cross-Reference Listing

The following XREF command prints a cross-reference listing on the default PRINTER IS device:

```
XREF
```

The next command sends a cross-reference to device selector 701:

```
XREF #701
```

Example Program and Cross Reference

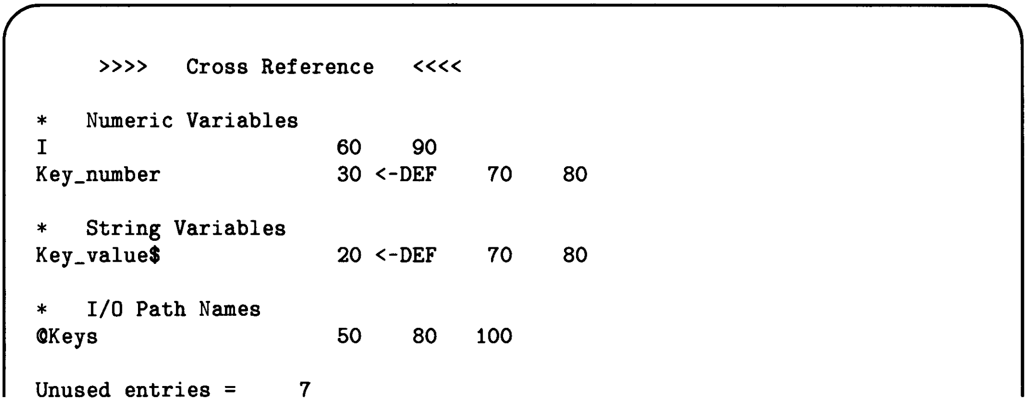
Here is an example program, with a corresponding cross reference.

```
10    ! Fil "DoKeyFile"
20    DIM Key_value$[160]
30    INTEGER Key_number
40    CREATE BDAT "SOFTKEYS",3
50    ASSIGN @Keys TO "SOFTKEYS"
60    FOR I=0 TO 9
70      READ Key_number,Key_value$
80      OUTPUT @Keys;Key_number,Key_value$
90    NEXT I
100   ASSIGN @Keys TO *
110   LOAD KEY "SOFTKEYS"
120   ! ---- Key Data ----------------------
130   DATA 8,"work!",5,"that",1,"See?",4,"you"
140   DATA 2,"I",3,"told",7,"would",6,"this"
150   END
```

Now generate a cross reference of the identifiers in the program:

XREF [Return] or [ENTER]

The following results are generated:



```
    >>>>   Cross Reference   <<<<

*   Numeric Variables
I                        60    90
Key_number               30 <-DEF    70     80

*   String Variables
Key_value$               20 <-DEF    70     80

*   I/O Path Names
@Keys                    50    80   100

Unused entries =     7
```

This is not an exhaustive list of XREF outputs, since there were no COM blocks, subprogram calls, line labels, etc. However, it does give an idea of the general format of a cross-reference listing. (For a complete description of XREF listings, see the *BASIC Language Reference.*)

Note the `<- DEF` which appears in some of the line-number lists; this symbol appears when:

- The identifier is a variable in a formal parameter list (that is, in a SUB or DEF FN statement).
- The identifier is a variable declared in a COM, DIM, REAL, INTEGER, or COMPLEX statement.
- The identifier is a line label for that line.

Unused Entries

At the end of each context, a line is printed that begins with:

```
Unused entries =
```

The number of "unused entries" deals with the internal workings of the system. It tells how many symbol table entries are available:

- for which space has already been made
- but which are not currently used by a variable

This is a count of the symbol table entries which have been marked by pre-run as "unused". Unreferenced symbol table locations which have not yet been marked "unused" by pre-run processing will show up in the lists of identifiers with empty reference lists. Note that a distinction is made here between "unused" and "unreferenced".

Pre-run will convert *unreferenced* symbol table entries (entries which are *defined* by the system but not *used* by a variable in the program) into "unused" entries. Unreferenced entries can arise because you changed your mind about a variable's name or corrected a typing error (once the system reserves space for a symbol table entry, this space is dedicated to the purpose of storing symbols until the corresponding context is destroyed, such as with SCRATCH). "Unreferenced entries" can also arise in syntaxing some statements where a numeric variable name which becomes a line label or a subprogram name is created. Also, REN (renumber) can cause line numbers to merge if you have unsatisfied line-number references. This shows up in the cross-reference as separate (but adjacent) entries for the multiple symbol table entries for the line number.

Let's go through an example to make this completely clear. At power-up the system creates an empty symbol table with space for five entries. Doing an XREF at this point will show `Unused entries = 5` and no other symbols.

Now type in the following program:

```
10 A=1
20 B=2
30 C=3
40 D=4
50 E=5
```

An XREF at this point will show five variables, each occurring in one line, and `Unused entries = 0` (all the pre-allocated spaces having been filled).

Now add one more line:

```
60 F=6
```

A subsequent XREF will show six variables and `Unused entries = 5`. This happens because when the system needs a symbol table location and none exists it always allocates six additional spaces: one for its immediate needs, and five spares for future use.

Now delete lines 10 to 60 using DEL 10,60 or the Delete line (DEL LN) key. Then perform another XREF. The listing will show six variables, each with an empty reference list, and `Unused entries = 5`.

Now store the following program line (as the only line):

```
10  END
```

and run the program. Now an XREF will show `Unused entries = 11`.

Doing a SCRATCH will restore the initial state with the symbol table reduced to five empty locations.

Now enter the following program lines:

```
10  GOTO A
20  A:END
```

Then execute XREF. This will show a numeric variable A (which is an artifact of the syntaxing process) and the line label A (referenced in two places). Running this program will cause pre-run to recognize that there is no occurrence of a numeric variable A in the program and reclaim the space for future use, converting it back into an "unused entry". Variables which are defined in the program are considered "referenced" and cannot be converted to "unused" even if no assignment or access is made to them, because they must be present in the symbol table in order for the program to list. Such variables must be found by looking at the XREF for variables with reference lists which contain only defining occurrences (`<-DEF`).

Single-Stepping a Program

One of the most powerful debugging tools available is the capability of single-stepping a program, one line at a time. This process allows the programmer to examine the values of the variables and the sequence in which the program is running at each statement. This is done with the STEP key (f1 in the System menu of an ITF keyboard).

There are three ways to use the STEP key:

1. If the program is stopped (i.e., a pre-run has to be performed), pressing the STEP key will cause the system to perform a pre-run on the program, but no program lines will actually be executed. The first line that will be executed will appear in the system message line at the bottom of the screen. Pressing the STEP key again will cause that line to be executed, and the next line after that to be executed will appear in the message line. If the STEP key is pressed causing the next line to appear in the display, and a live keyboard operation (such as examining the value of a variable) is performed, the contents of the message line will change. Pressing the STEP key again will still cause the line to be executed, even though it is no longer visible in the display line. After the statement has completed, the next line will appear.
2. If the program is in an INPUT or LINPUT statement, pressing the STEP key is sufficient to terminate the operation. Any data entered from the keyboard will be entered into the correct variables, just as though CONTINUE (f2 on the ITF keyboard) or ENTER (Return on the ITF keyboard) had been pressed, but program execution will be PAUSEd, and the statement immediately following the INPUT or LINPUT will appear in the system message line.
3. If the program is in a PAUSEd state, pressing the STEP key will cause the next line to be executed. The program counter will not be reset, nor will a pre-run be performed. Again, the next line to be executed will appear in the system message line after the last one has been completed. A paused state is indicated by a dash in the run light in the lower right-hand corner of the screen.

Type in the following example and execute it by pressing the STEP key repeatedly.

```
10  DIM A(1:5)
20  ! This is an example
30  S=0
40  FOR I=1 TO 5
50  INPUT "Enter a number",A(I)
60  S=S+A(I)
70  NEXT I
80  PRINT S
90  PRINT A(*);
100 END
```

Notice that the STEP key caused every statement to appear in the system message line, one at a time, even those statements that are not really executed, like DIM and comments.

If you are stepping a program and encounter an INPUT, LINPUT, or ENTER KBD statement, you can use Return, ENTER, or CONTINUE to enter your responses. The system will remember that you are stepping the program and remain in single-step mode after the input operation is complete (unless you press CONTINUE again *after* the input operation is complete).

If you hold down the STEP key, to continuously step through program lines, you may want to turn softkey labels off (especially when using bit-mapped alpha displays).

Tracing

The process of single-stepping, wonderful though it is, can be quite slow, especially if the programmer has little or no idea which part of his program is causing the bug. An alternative way of examining variable changes and program flow is available in the form of the TRACE ALL statement.

TRACE ALL

When the TRACE ALL command is executed, it causes the system to issue a message prior to executing every line (this shows the order in which the statements were executed), and if the statement caused any variables to change value, a message telling the variables involved and their new values is also issued. The messages are issued to the system message line, and the most useful way to use the TRACE ALL feature is to turn Print All On with the PRT ALL key (f4 in the System menu of an ITF keyboard), unless of course you're a very fast reader. (The printall mode will cause all information from the DISP line, the keyboard input line, and the system message line to be logged on the PRINTALL IS device.)

Turn Print All ON and key in the following example to see how TRACE ALL works:

```
10   TRACE ALL
20   FOR I=1 TO 10
30      PRINT I;
40      IF I MOD 2 THEN
50            PRINT " is odd."
60      ELSE
70            PRINT " is even."
80      END IF
90   NEXT I
100  END
```

There are two optional parameters that can be used with TRACE ALL. Both parameters are line identifiers (line numbers or line labels). The first parameter tells the system when to start tracing, and the second one (if it's specified) tells the system when to stop tracing. The following example illustrates the use of one optional line specifier:

```
1     TRACE ALL 40
10    DIM A(1:10)
20    FOR I=1 TO 100
30    NEXT I
40    FOR J=1 TO 10
50    A(J)=J
60    NEXT J
70    END
```

It is usually more useful to use the TRACE ALL command from the keyboard rather than from the program because a program modification is not necessary if you want to trace a different part of the program. All that's necessary is to type in a new TRACE ALL command from the keyboard to override the old one. In the above example, to trace the loop from 20 to 30 instead of the one from 40 to 60, simply delete line 1 and type in `TRACE 20,40` from the keyboard.

```
10    DIM A(1:10)
20    FOR I=1 TO 100
30    NEXT I
40    FOR J=1 TO 10
50    A(J)=J
60    NEXT J
70    END
```

The program will begin tracing at line 20, and keep on tracing until it's ready to execute line 40, at which time it will terminate the trace messages and will continue executing the program normally.

If the TRACE ALL statement uses a line label instead of a line number, be aware of what happens if you have more than one occurrence of a given line label in your program. For instance, it is perfectly legal to have the same line label in two or more different program environments—line labels are local to subprograms and branching operations addressing a given line label are treated separately in different subprograms.

However, when a TRACE ALL using a line label is executed, the first line label in memory is the one that gets used, regardless of the environment the program was in when the TRACE ALL statement was executed. Thus in the following program, even though the `TRACE ALL Printout` statement is executed inside the subprogram, tracing does not commence until the subprogram has been exited and the `Printout` statement in the main program has been executed.

```
10   DIM A(1:10)
20   FOR I=1 TO 10
30      CALL Dummy(A(*),I)
40      GOSUB Printout
50   NEXT I
60   STOP
70 Printout: !
80   FOR J=1 TO 10
90   PRINT A(J);",";
100  NEXT J
105  PRINT
110  RETURN
120  END
130  SUB Dummy(X(*),Z)
140  TRACE ALL Printout
150  FOR I=1 TO 10
160     X(I)=Z*100+I
170  NEXT I
180  GOSUB Printout
190  SUBEXIT
200 Printout: !
210  PRINT "Dummy routine executed";Z
220  RETURN
230  SUBEND
```

If two line identifiers are used, their location with respect to each other does not matter. Tracing will start when the line specified first is encountered, and it will stop when (or if) the second line is encountered.

PRINTALL IS

The PRINTALL IS command is useful for switching the tracing messages between the CRT and a hardcopy printer. For instance, turning PRINTALL ON during pre-run will allow you to see which array variable has not been dimensioned. (Again, to get any record at all of the trace messages, Print All must be On.) To cause the trace messages to be logged on the CRT, execute PRINTALL IS CRT. (The CRT is the default PRINTALL IS device that the system assumes when it wakes up.) To cause the messages to be logged on a printer, merely change the select code to the appropriate value (PRINTALL IS 701).

TRACE PAUSE

The TRACE PAUSE command can be used to set a "break point" in the program. The program will execute at a reduced speed until the specified line is reached, at which time the program will pause, and the specified line will be shown in the display line, indicating that the program will execute it when execution is resumed. Execution may be resumed with the [CONTINUE] key ([f2] in the System and User menus on an ITF keyboard), the [STEP] key ([f1] in the System menu on an ITF keyboard), or by executing CONT from the keyboard (the specified line identifier must be located in the current environment).

By executing the command `TRACE PAUSE Printout` from the keyboard, the following program will pause every time it reaches line 70.

```
10   DIM A(1:10)
20   FOR I=1 TO 10
40      GOSUB Printout
50   NEXT I
60   STOP
70 Printout: !
80   FOR J=1 TO 10
90   PRINT A(J);",";
100  NEXT J
110  PRINT
120  RETURN
130  END
```

Try the following ways of continuing execution:

- press [STEP] ([f1] on the ITF keyboard)
- press [CONTINUE] ([f2] on the ITF keyboard)
- execute `CONT 110`

As with TRACE ALL, a new TRACE PAUSE statement overrides a previous one. The same rules are applied when a line label is used in a TRACE PAUSE statement as are applied to the TRACE ALL statement—the first line in memory having that label is used.

TRACE OFF

TRACE OFF cancels the effects of any active TRACE ALL or TRACE PAUSE statements. The status of Print All and the PRINTALL IS device will be unchanged.

TRACE OFF may be executed either from the program, or from the keyboard.

The CLR I/O (Break) Key

The CLR I/O key (Break on the ITF keyboard) suspends any active I/O operation and pauses the program in such a way that the suspended statement will restart once CONTINUE (f2 on the ITF keyboard) or STEP (f1 on the ITF keyboard) is pressed. This is useful for operations which appear to "hang" the machine, such as printing to a printer which isn't turned on.

Most devices will not respond to ENTER requests unless they have first been instructed to respond. If improper values are sent to a device, it may refuse to respond. Therefore, CLR I/O can help in debugging these situations.

Here are the operations that can be suspended with CLR I/O.

PRINT
LIST
CAT
OUTPUT
DUMP GRAPHICS
DUMP ALPHA
SEND
PRINTALL outputs
ENTER
INPUT
HP-IB commands
External plotter commands
ASSIGN
PURGE
CREATE

Efficient Use of the Computer's Resources 13

Efficient Use of the Computer's Resources 13

Every model of computer has certain characteristics which can result in better performance, provided the programmer knows what those characteristics are and how he can take advantage of them. This chapter consists of a potpourri of such items.

Data Storage

It is usually desirable to minimize the usage of computer memory and mass storage. This section describes how much space is required to store various types of data, which will help you in using your storage resources for the best possible utilization.

Data Storage in Read/Write Memory

There are five data types on this computer: REAL, INTEGER, COMPLEX, strings, and I/O path names. The memory occupied by data is made up of two parts: the memory it actually takes to hold the intended information, and the memory that the system uses to keep track of the information's location and form (this is called overhead). Strings, INTEGERs, COMPLEXs and REALs can be declared either as simple variables or as arrays. Arrays take different amounts of overhead than simple variables, but each element of an array uses the same amount of memory that a corresponding simple variable uses to actually store information.

The overhead required for any given symbol is kept in three tables:

- the symbol table
- the token table
- and the dimension table

The symbol table contains pointers to the value area, where the actual information is kept, and to the other two tables. The token table contains the names of the various symbols. The dimension table contains length information for strings and arrays, and is not used for numeric scalars. The tables are not constructed in single units as symbols are added and deleted. Rather, as new space is required, the system will first look to see if there are any unused entries in the tables—if new space is allocated, usually enough for several entries is allocated. For instance, the symbol table is built in increments of five entries.

Symbol Table Overhead: 10 bytes per symbol

Token Table Overhead: Number of characters in the name + 1 (if the above number is odd, it is rounded up to an even number). Note that the name for I/O path names, strings, and functions includes the @, $, and FN, respectively.

Dimension Table Overhead: For arrays: 3 bytes (total size)
1 byte (number of dimensions)
4 bytes for each dimension (for the lower bound, and the size of each dimension)

For strings: 2 bytes (maximum length)

For string arrays: all of the normal array overhead, plus two bytes for the maximum allowed length of an element

Note that line labels, COM labels, and subprograms are considered as symbols, and occupy space in both the symbol and token tables. Line numbers used **in** statements, like GOTO 20, also occupy space in the symbol table.

Every subprogram (or context) has its own set of tables. In addition, there is a global set of COM tables, where all information concerning COM blocks is kept. Symbols that belong to a COM block will occur in both the COM tables and in any local tables in which that COM block is declared. Since each context may define the names by which it refers to COM block variables, there will be no entry in the COM token table for each variable, but an entry in the COM token table **will** occur for COM labels.

ALLOCATEd variables require four bytes of overhead in addition to the overhead already discussed for the symbol, token, and dimension tables.

The following table summarizes the storage requirements for various data types. This table does not show the extra requirements just mentioned for ALLOCATEd and COM variables.

Table 13-1. Data Type Storage Requirements

Type	Overhead	Information Storage
Simple INTEGER	10 bytes + name overhead	2 bytes
Simple REAL	10 bytes + name overhead	8 bytes
Simple COMPLEX	10 bytes + name overhead	16 bytes
Simple string	12 bytes + name overhead	1 byte per char. up to declared length (padded to even number of chars.) + 2 bytes (length information)
I/O path name	10 bytes + name overhead	100 bytes
INTEGER array	14 bytes + name overhead + 4 bytes per dimension	2 bytes per element
REAL array	14 bytes + name overhead + 4 bytes per dimension	8 bytes per element
COMPLEX array	14 bytes + name overhead + 4 bytes per dimension	16 bytes per element
String array	16 bytes + name overhead + 4 bytes per dimension	1 byte per char. up to declared length (padded to even number of chars.)+ 2 bytes (length information) per element

Data Storage on Mass Memory Devices

The amount of storage that data takes on mass storage media is similar to the amount of memory that data takes internally, except that no overhead is required (on BDAT files). Arrays and single values are interchangeable on mass storage—no distinguishing information is kept on the media.

INTEGERs (and INTEGER arrays)	2 bytes (per element)
REALs (and REAL arrays)	8 bytes (per element)
COMPLEXs (and COMPLEX arrays)	16 bytes (per element)
Strings (and string arrays)	4 bytes + 1 byte per char up to current length, padded to even number of chars. (per element)

For ASCII files, all information is converted to string (or ASCII) form, and a two-byte length field is tacked onto the front of every field.

INTEGERs (and INTEGER arrays)	2 bytes + 1 byte per digit (per element)
REALs (and REAL arrays)	2 bytes + 1 byte per digit (per element)
COMPLEXs (and COMPLEX arrays)	2 bytes + 1 byte per digit (per element)
Strings (and string arrays)	2 bytes + 1 byte per char (per element)

Comments and Multi-character Identifiers

Self-documenting features such as in-line comments and multi-character variables and line labels are useful because of the benefits to be reaped in terms of developing, testing, debugging, and maintaining programs. They do take extra memory, but this shouldn't be a problem if you keep the following points in mind.

Comments take 1 byte of memory for every character in the comment. If memory space becomes a problem, many people resort to keeping two copies of their programs around—one fully commented to use as reference material, and the other uncommented to use as the "production version," which is the one that is actually used.

Multi-character identifiers are only spelled out in their entirety once—not every time they are used. The program actually stores pointers whenever a reference to the identifier is used, so using short identifiers won't result in any appreciable savings in memory used.

Variable and Array Initialization

Care should be taken to initialize any variables before using them in an expression (on the right hand side of an =, as a left-hand subscript in a function or subprogram parameter list, as an argument to a built-in function, or in a PRINT/OUTPUT/DISP list). The system will set variables to zero, strings to null, and I/O path names to undefined at program prerun, but depending upon default settings is considered bad programming practice and could lead to subtle errors. For instance, the first time a certain line is executed, the variables used may be assumed to be zero because of the prerun operations. Once this assumption has been made, the danger is that the programmer will branch back to the same section of code and forget that the zeroing process has not been performed—an error may result that didn't occur previously.

Mass Memory Performance

Program Files

There are two ways to store programs—they can be saved either as ASCII source strings using the SAVE command, or they can be stored in an intermediate form that the BASIC language system understands using the STORE command.

If the time it takes to load the program is important, always use the STORE command to store the program instead of the SAVE command. The LOAD command, which reads in files created by the STORE command, will execute about fifty times faster than the GET command. This is because the LOAD command does not require that the information on the file be processed in any way. Since the program is already in the form the system needs it in, all that is necessary is to funnel the program directly into memory as fast as the disc can spin (assuming an interleave of one).

SAVE files, on the other hand, require that the system parse and check the lines as they are read, just the same as if a user had typed them in from the keyboard. Consequently, the speed at which the program gets loaded into memory with the GET command will be drastically slower than the LOAD command. Using the Model 226 and 236 internal drives as an example of the relative speeds, a typical 8-Kbyte program will take about 30 seconds to GET, but only about one second to LOAD.

One advantage of the GET/SAVE commands is that it is possible to deal with programs as string data.

Data Files

As with program files, there are two types of data files: ASCII and BDAT. ASCII files require that all data be in string form, while BDAT files are interpreted as internal data representations.

When reading or writing data to an ASCII file, the number formatter is required to convert the data in between its internal representation and its ASCII form. When reading or writing data to a BDAT file, the data may stream directly back and forth with no conversion required. Using the Model 226 and 236 internal drives as an example, an 8K-element REAL array (64K bytes) may take around 200 seconds to write in an ASCII file, while the same array will only take about 5 seconds to write to a BDAT file.

The primary benefit of the ASCII data file is the transportation of data between different models of Hewlett-Packard computers and terminals and between discs used with different language systems.

Benchmarking Techniques

This section discusses the techniques used to determine how fast various operations execute. Ideally, you should separate the measurement time from elapsed time:

```
10   T1=TIMEDATE
20   T2=TIMEDATE
30   PRINT T1-T2;"seconds used to read clock"
40   END
```

In actuality, the clock only has a resolution of 10 ms, so you won't usually be able to time this operation.

Next, most operations are performed inside a loop in order to be able to time operations that are faster than the resolution of the clock (clock resolution is 10 ms.). This also tends to "smooth out" varying system overhead characteristics.

```
10   INTEGER I
20   T1=TIMEDATE
30   FOR I=1 TO 10000
40   NEXT I
50   T2=TIMEDATE
60   PRINT T2-T1;"seconds of loop overhead"
70   END
```

A certain amount of time used in computational operations will involve moving information around. The time will be different depending upon the type of the information being moved (string, REAL, COMPLEX or INTEGER), and for strings, the length.

```
10   REAL A,B,C
20   INTEGER I
30   B=PI
40   T1=TIMEDATE
50   FOR I=1 TO 10000
60   A=B
70   NEXT I
80   T2=TIMEDATE
90   PRINT T2-T1;"seconds of loop overhead"
100  END
```

The next step is to actually time the operation of interest. It should be noted that for arithmetic operations, the time spent performing the operation will vary depending upon the two operands (number of digits and relative magnitudes).

```
10   REAL A,B,C
20   INTEGER I
30   B=PI*1.E+53
40   C=EXP(SQR(2)^13.81)
50   PRINT "B=";B,"C=";C
60   T1=TIMEDATE
70   FOR I=1 TO 10000
80     A=B
90   NEXT I
100  T2=TIMEDATE
110  FOR I=1 TO 10000
120    A=B+C
130  NEXT I
140  T3=TIMEDATE
150  Op_time=DROUND(T3-T2-T2+T1,3)
160  PRINT Op_time*100;"us. per operation"
170  END
```

The above program will show anywhere from 148 to 150 microseconds per operation for addition.

Here is a list of a few other operations:

Addition	150 μs
Subtraction	165 μs
Multiplication	301 μs
Division	460 μs
Exponentiation	7590 μs

These times vary for different processor boards. Use these times and others throughout this chapter to compare the speeds of different operations.

INTEGER Variables

We have seen in the first section of this chapter that INTEGER variables don't take as much memory as REAL variables (2 bytes instead of 8). Now we shall discover that some operations with INTEGERs are much faster than the same operations with REALs.

Minimum and Maximum Values

The INTEGER variable type may store any whole number from −32 768 to +32 767 inclusive.

Mathematical Operations

There are two sets of math routines provided for the MOD, DIV, +, −, and * operations: REAL and INTEGER. Depending upon the types of the operands used, the execution times for these operations will vary widely. The tradeoffs are:

INTEGER math is the faster of the two, since it doesn't require as much "work." This is because:

1. There are only two bytes of data to process instead of eight.
2. Operations do not have to deal with a combination of mantissa and exponent.
3. The results don't have to be normalized.
4. INTEGER math can be done directly in the hardware.

REAL math, though slower, is generally more widely used because it allows numbers with fractional parts to be analyzed. REAL numbers carry about 16 decimal digits of precision and have an exponent range of −308 to +308.

Note

All times specified are without the floating point card or the MC68881 math coprocessor. If you have this card, your times will be faster for REAL math.

For instance, suppose you want to compute your monthly pay. Assume that you're making $5.17 an hour, that you work twenty-four days per month and that you work 14 hours per day. The calculation that you would use is `5.17*24*14` or $1737.12. In this problem, you definitely want your computer to use REAL precision math (or you'll lose 17 cents per hour!) even though you're only using 6 of the 16 digits available.

The computer will pick whatever math routines it needs to solve the current problem. However, the programmer can exercise control over which math routines get executed if the following rules are understood.

- INTEGER math is used if both arguments of a MOD, DIV, *, +, or − operation are of type INTEGER. If the results of the operation cannot be stored in an INTEGER, then an error is generated (INTEGER overflow).
- REAL math is used if either or both arguments of a MOD, DIV, *, +, or − operation is of type REAL. If one of the arguments is of type INTEGER, then that argument is first converted to REAL.
- REAL math is always used for exponentiation (^) and division (/).

The following table gives some approximate time comparisons[1] between INTEGER and REAL operations for +, −, and *. The times are approximations because REAL math routines do different things depending upon the values of the operands. All times shown here were found on operations with numbers having no fractional parts. The multiplication times were found for operands in the range of −200 to +200.

Table 13-2. Approximate Execution Times: INTEGER vs. REAL

Operation	REAL	INTEGER
MOD	160 μs	91 μs
DIV	352 μs	88 μs
Addition	142 μs	68 μs
Subtraction	174 μs	68 μs
Multiplication	152 μs	77 μs

Multiplication, like most math operations, will execute faster on INTEGER values. However, bear in mind that it's much easier to get an INTEGER overflow on multiplications than on additions and subtractions. For instance, 200*200 will give an INTEGER overflow. If you are performing multiplication on INTEGERs, you should carefully examine your program to ensure that the range of your answers doesn't force you to use REALs, even if the requirement for fractional precision doesn't.

[1] These times are for a Series 200 computer with an MC68000 processor running at 8 MHz. They will be significantly decreased on machines with higher clock rates or floating-point math hardware (HP 98635 math card or MC68881 co-processor).

Loops

In general, any FOR/NEXT loop using an index which has been declared to be an INTEGER will execute about 2.4 times faster than a loop whose loop counter is a REAL. Type in the two programs below and run them to see the difference.

```
10   REAL I
20   TO=TIMEDATE
30   FOR I=1 TO 10000
40   NEXT I
50   PRINT TIMEDATE-TO;"seconds"
60   END
```

Time is about 1.67 seconds.

```
10   INTEGER I
20   TO=TIMEDATE
30   FOR I=1 TO 10000
40   NEXT I
50   PRINT TIMEDATE-TO;"seconds"
60   END
```

Time is about .69 seconds.

Bear in mind that the 2.4 speed improvement is only on the time devoted to actually incrementing and testing the loop variable (in these examples, I). So, any loop that iterates for 10 000 times will run about a second faster if the index is an INTEGER instead of a REAL. Now, saving a second on a loop that executes 10 000 times may not sound like much by itself, and it's not. But what if that loop is nested inside *another* one that executes 10 000 times? Now your time savings is 10 000 seconds, or two hours and forty-five minutes! Just for declaring the loop counters to be INTEGER.

Naturally, making a loop index an INTEGER will only work if the loop is not stepping in fractions, and if the minimum and maximum values of the loop index do not exceed the range of −32 768 to +32 767.

Array Indexing

Accessing individual array elements is faster if the variables or expressions giving the indices into the array are INTEGER instead of REAL. This is because the system has to convert floating-point numbers into an INTEGER in order to find the offset from the beginning of the array. If the index is already in INTEGER form, the conversion isn't necessary. The following example illustrates this point.

```
10   REAL I
20   DIM A(1:1000)
30   X=17.568
40   TO=TIMEDATE
50   FOR I=1 TO 1000
60      A(I)=X
70   NEXT I
80   PRINT TIMEDATE-TO;"seconds"
90   END
```

```
10   INTEGER I
20   DIM A(1:1000)
30   X=17.568
40   TO=TIMEDATE
50   FOR I=1 TO 1000
60      A(I)=X
70   NEXT I
80   PRINT TIMEDATE-TO;"seconds"
90   END
```

You will find a difference of .14 seconds between the two programs' execution times, due to a combination of the loop counter being INTEGER and the INTEGER indexing of the array. Again, if you're operating on a much larger array, or if you're working on a multi-dimensional array, this number can become noticeable.

REAL and COMPLEX Numbers

This section describes details of using and storing REAL numbers. The information can generally be applied to COMPLEX numbers, since COMPLEX numbers are essentially two REAL numbers.

Minimum and Maximum Values

The minimum REAL number that can be stored on this computer is approximately $\pm 2.225\,073\,858\,507\,202 \times 10^{-308}$ (The MINREAL function returns this value.)

The maximum REAL number that can be stored on this computer is approximately $\pm 1.797\,693\,134\,862\,315 \times 10^{308}$ (The MAXREAL function returns this value.)

A REAL number can also have the value zero.

Type Conversions

Earlier, it was mentioned that any time a MOD, DIV, *, +, or − operation is performed on two numbers of different type (one INTEGER, and one REAL), a type conversion has to take place to convert the INTEGER to a REAL. This section will address other situations where type conversions have to take place.

Any time an INTEGER is used in an exponentiation or division operation, it must first be converted to a REAL.

All of the following functions require a REAL argument (with the exception of VAL and RND), and all of them return a REAL value (with the exception of RANDOMIZE). If an INTEGER is passed in, or if the result is to be stored in an INTEGER, then the appropriate type conversion must be made: EXP, LGT, LOG, RANDOMIZE, SQRT, DROUND, RND, ACS, COS, ASN, SIN, ATN, TAN, VAL. Note that many of the previously mentioned function also take COMPLEX arguments and return COMPLEX arguments. These functions are: EXP, LGT, LOG, SQRT, ACS, COS, ASN, SIN, ATN, TAN.

All of the comparison operators (=, <>, <, >, <=, >=) will return INTEGER values (0 or 1) but will accept either INTEGERs or REALs as arguments. The only comparison operators allowed with COMPLEX values are = and <>. For more information on comparisons of COMPLEX values read the section "REAL and COMPLEX Numbers and Comparisons" in the chapter entitled "Handling Errors." The logical operators AND, EXOR, OR, and NOT will convert any arguments to the INTEGER values 0 or 1 before the operation is performed, and an INTEGER 0 or 1 will be returned.

The binary bit functions (BINAND, SHIFT, ROTATE, BINIOR, BINCMP, BIT, BINEOR) require INTEGER inputs and provide INTEGER outputs. Type conversions will be performed if REALs are supplied as inputs, or if the results are to be stored in a REAL variable.

SGN returns an INTEGER (−1, 0, 1) regardless of the type of the argument passed to it. ABS and INT return the type of the argument that's passed to them.

If two INTEGERs are used to perform a MOD, DIV, *, +, or − operation, but the result is to be stored in a REAL variable instead of an INTEGER, then the result must be converted from INTEGER to REAL.

Here is how long each type conversion takes:

INTEGER to REAL: 42 microseconds
REAL to INTEGER: 34 microseconds

Constants

All constants that are within the range of −32 767 to 32 767 that aren't entered with a decimal point or an "E" (for scientific notation) are stored in the machine as INTEGERs. Integer constants should always be used with INTEGER variables, but if they are used with REAL variables they will have to be converted to REAL first. This operation will slow down the execution of the program, as shown in the previous section. Any numbers entered with decimal points (1.0, 3., .7, etc.), with an E (1E−304, .2E48, 0E0, etc.), or outside the valid INTEGER range (40000, −75986, etc.) will be stored as REAL constants.

Polynomial Evaluations

The polynomial can waste much computer time because programmers tend to pick the most obvious, and also the most time-consuming, method of evaluating them. Polynomials are usually written mathematically as:

$$y=a_nx^n + a_{n-1}x^{n-1} + \ldots + a_1x + a_0$$

or

$$y= \sum_{i=0}^{n} a_ix^i$$

Hence the temptation is strong to evaluate a polynomial on a computer as:

```
2000 DEF FNPoly(X,Coefficient(*),INTEGER N)
2010 INTEGER I
2020 Y=0
2030 FOR I=0 TO N
2040    Y=Y+Coefficient(I)*(X^I)
2050 NEXT I
2060 RETURN Y
2070 FNEND
```

In the above program, there are N+1 additions, N+1 multiplications, N+1 exponentiations, and N+1 INTEGER to REAL conversions (I is converted to a REAL when the exponentiation operation is performed). Now suppose the polynomial is written in the equivalent form:

$$y=a_0 + x(a_1 + x(a_2 + \ldots + x(a_n) \ldots))$$

Then the corresponding program would be:

```
2000 DEF FNPoly(X,Coefficient(*),INTEGER N)
2010 INTEGER I
2020 Y=Coefficient(N)
2030 FOR I=N-1 TO 0 STEP -1
2040    Y=Coefficient(I)+X*Y
2050 NEXT I
2060 RETURN Y
2070 FNEND
```

Now there are only N additions and N multiplies, and NO exponentiations or INTEGER to REAL conversions! The following chart shows the time savings as a function of the order of the polynomial. For example, if the polynomial is of order 10, it is 70 milliseconds faster to use the nested multiply method to evaluate the polynomial than to use exponentiation. If you're plotting a thousand points on a graph, nested multiplication will save you more than a minute!

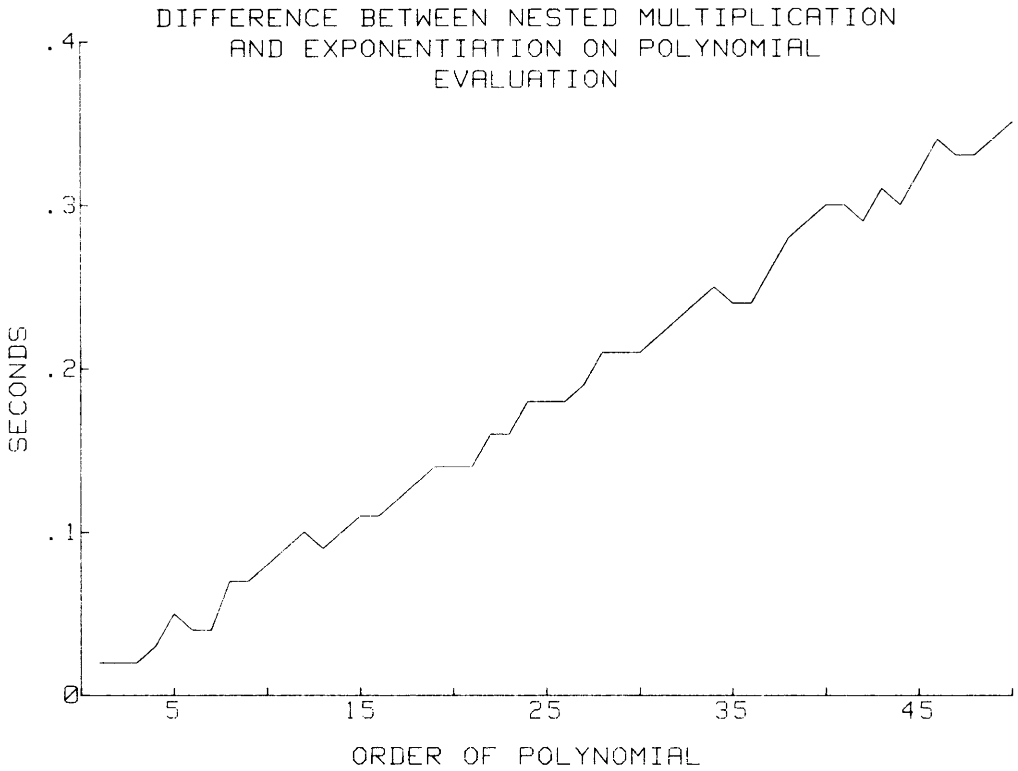


Figure 13-1. Time Savings as a Function of Polynomial Order

Logical Comparisons for Equality on REAL Numbers

Don't do it.

If you are performing mathematical operations on REAL numbers, and then comparing them for equality, the chances are that they will never come up equal. For example, in the previous section on polynomial evaluation, you can pass the same value for X and the same coefficient array to each of the two functions, and although the results will look equal when you print them out, they won't show equality if you compare them. (Try it and see.) A shorter example is to execute this expression from the keyboard:

```
.1+.1+.1+.1+.1+.1+.1=.7
```

The 0 at the bottom of the screen means that the machine doesn't consider the two numbers to be equal. This is characteristic of the way that binary math works.

Any REAL numbers should be rounded first before being tested for equality. This is one of the purposes of the DROUND function.

```
DROUND(.1+.1+.1+.1+.1+.1+.1,12)=DROUND(.7,12)
```

After rounding both numbers to 12 digits, the computer will now accept them as being equal. See the "Numeric Computation" chapter for more discussion on the comparison of REAL numbers.

Saving Time

Multiply vs. Add

It is always faster to add a number to itself than it is to multiply it by 2. For instance, to perform 2*PI a thousand times takes .22 seconds, while to perform PI+PI a thousand times takes .13 seconds.

However, if you want to multiply by 3, that is faster than adding the number three times. 3*PI executed a thousand times takes about the same as 2*PI (.22 seconds), but adding PI+PI+PI a thousand times takes about .28 seconds.

Exponentiation vs. Multiply and SQRT

Exponentiation is very slow when compared to other mathematical operations. The results are worth the wait when the exponent has a fractional part; raising a REAL number to a REAL power is a complex operation. However, time can be saved if you are alert to some special cases. The most common examples are squaring a number or finding a square root. Using SQRT(X) takes only about one-fourth the time required by the expression X^.5. Even more dramatic savings can be gained when raising numbers to an integer power. Using X*X yields a 22-to-1 time savings over the expression X^2. When using powers greater than 2 or 3, the expressions needed to achieve the repeated multiplication can be somewhat cumbersome, and may not even fit on a program line. However, repeated multiplication is so much faster than exponentiation that time savings can be realized (for powers up to 14) even if a FOR...NEXT loop has to be added to repeat the multiplication.

Array Fetches vs. Simple Variables

It takes more time to access an array element than it does a simple variable. This is because the address of the array element has to be computed from the starting address of the array and the offset within the array based on the specified indices. A simple variable's address does not require this computation.

Thus, if you access a given array element often enough, especially within a loop, it will often be faster to store the array element into a simple variable and then operate on the simple variable.

Time to fetch a simple variable and store it:	136 μs
Time to fetch an array variable and store it:	260 μs
Difference:	124 μs

This means that it is faster to fetch three simple variables than it is to fetch two array elements.

Concatenation vs. Substring Placement

The concatenation operator (**&**) allows you to combine two or more strings to construct another string. This operation is discussed in the "String Manipulation" chapter. However, there is a special case that can be accomplished faster without the concatenation operator. This is the case where the new string is built by appending to the end of an existing string. For example, **A$=A$&B$**.

Concatenation works by constructing a temporary workspace in which all the components are assembled. Then the result is moved to its destination. In the example above, A$ is moved to a temporary workspace, B$ is appended to it, and the result is moved back to A$. Thus, the original contents of A$, which weren't changed, have been moved twice unnecessarily. A faster way to accomplish the same thing is:

```
A$[LEN(A$)+1]=B$
```

Another benefit of this approach is that the temporary workspace is not created. If memory is tight and A$ is very large, concatenation could create a memory overflow.

The following chart shows the time savings that result from using substring placement instead of concatenation.

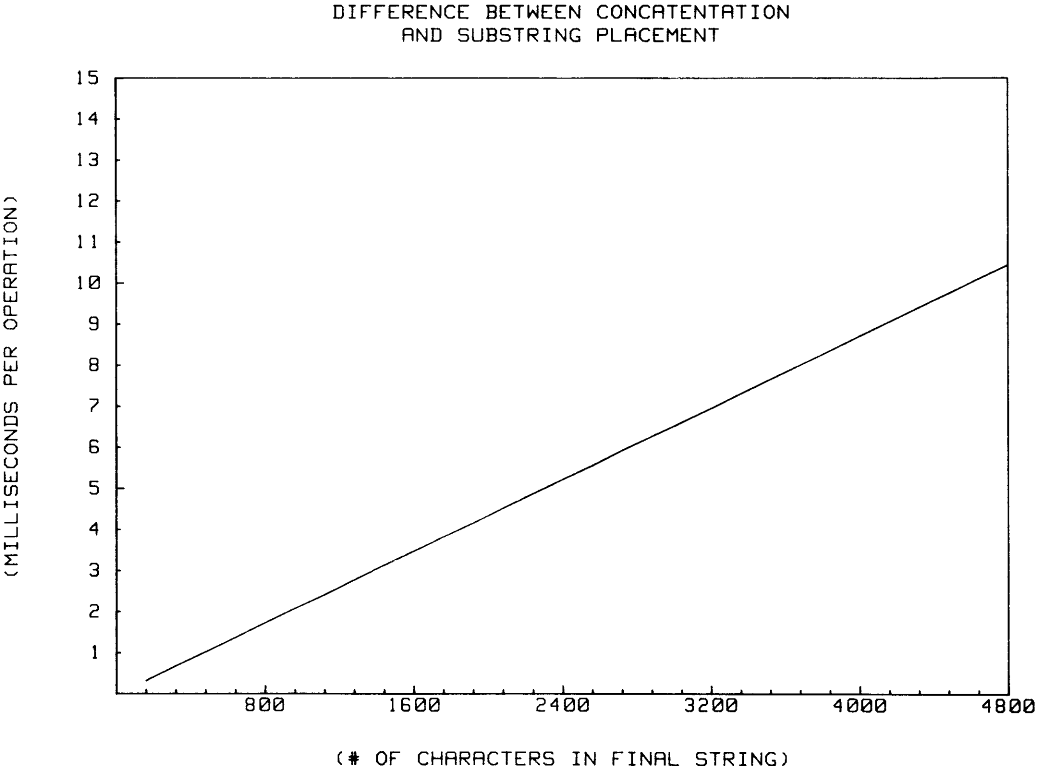


Figure 13-2. Time Savings: Substring Placement vs. Concatenation

HP 98635 Floating-Point Math Card

This card contains a special chip which performs floating-point math computations in hardware rather than in software. It provides significant speed improvements over the "math library" (software) computation method.

The BASIC system uses this card automatically, whenever installed. However, you can disable and enable its use with CONTROL statements just like you can the MC68881 co-processor. See the following section for details.

MC68881 Floating-Point Math Co-Processor

Series 300 computers may optionally be equipped with MC68881 floating-point math co-processors. Not only does the MC68881 provide increased speed of floating-point math calculations, but it also increases the accuracy of these calculations. The MC68881 has 80-bit (binary) precision as opposed to the 64-bit (binary) precision of the BASIC math library and HP 98635 Floating-Point Math Card. In a series of standard math tests, the RMS (root mean square) error in the 10 worst cases for the MC68881 ranged from 0 to 0.37 bit error. For the software math library and Floating-Point Math Card, the RMS error in the worst 10 cases ranged from 0.33 to 4.2 bits of error.

While the BASIC math library and the HP 98635 Floating-Point Math Card produce identical results, these values may not agree with those obtained from using the MC68881. This may only be noticeable when strict equality with the math library or Floating-Point Math Card is required (which is not recommended). For strict compliance, disable the MC68881.

Enabling and Disabling Floating-Point Math Hardware

You can determine whether the MC68881 floating-point math co-processor or HP 98635 Floating-Point Math Card is currently enabled with the following statement:

```
STATUS 32,2;Float_on
```

If the variable called `Float_on` is assigned a value of 1, then the floating-point hardware is currently enabled (this is the default condition). If it is assigned a value of 0, then it is disabled.

If floating-point math hardware is enabled but you want to disable it, execute this statement:

```
CONTROL 32,2;0
```

If you want to re-enable this feature, you can do so with this statement:

```
CONTROL 32,2;1
```

MC68020 Internal Cache Memory

The MC68020 processors available on Series 300 computers have on-chip high-speed cache memory. This memory serves as storage for machine instruction sequences, typically allowing the processor to decrease the amount of off-chip memory accesses and thus speed program execution.

Enabling and Disabling Cache Memory

You can determine whether or not cache memory is currently enabled with this statement:

```
STATUS 32,3;Cache_on
```

If the variable called `Cache_on` is assigned a value of 1, then cache is currently enabled (this is the default condition). If it is assigned a value of 0, then cache is disabled.

If the cache feature is enabled, but you want to disable it, you can do so with this statement:

```
CONTROL 32,3;0
```

If you want to re-enable this feature, execute this statement:

```
CONTROL 32,3;1
```

Saving Memory

The ALLOCATE and DEALLOCATE statements can be used to make efficient use of memory space in certain applications. They are useful in programs that contain a number of large variables that are not all needed simultaneously. For example: during data collection, a large string array is needed; during data processing a large numeric look-up table is needed; and during output formatting, a string array is needed again. Because a mass storage device is used to hold the data between processes, the same memory area can be used for all these arrays.

To use ALLOCATE effectively, it is necessary to understand how the system reclaims areas that have been DEALLOCATED. Space for allocated variables is built using a stack discipline. The DEALLOCATE statement marks a space as unused. Unused space can be reclaimed only if it is the *last* space on the stack. There are two operations that use space in this stack. One is ALLOCATE, and the other is ON <event>.

Keeping other allocated variables from blocking deallocated space is relatively simple. If you have only one allocated variable at any given time, this is not a problem. If you have allocated variables in existence simultaneously, ALLOCATE them in the opposite order of the DEALLOCATE statements. Think of this in the same way you would think about nesting FOR...NEXT loops.

Preventing blockage by ON conditions is more complicated. ON conditions, with one exception, create control blocks that are *permanent* entries on the stack. As soon as you declare an ON (or OFF) condition, all the previous entries on the stack are "locked in" for the duration of the context and cannot be reclaimed. Therefore, all the control blocks should be created *before* any variables are allocated. Once a control block is created, it will be used by all subsequent ON and OFF statements that refer to the same resource. A good technique is to include an OFF statement for each desired event before allocating any variables.

The exception mentioned above is an ON condition declared for an I/O path name. This includes ON END, ON EOT, and ON EOR. For these, subsequent ON and OFF statements behave as previously described. However, if the I/O path is closed, any control blocks associated with the path are marked as unused. This has two implications. One, the reclaiming of the stack will not be blocked after the I/O path is closed. Two, you cannot force the system to leave these control blocks at the beginning of the stack. Here is an example:

```
200  ASSIGN @File to "FRED"
210  ON END @File GOTO Label1
220  ALLOCATE Array(255,4)
 .
 .
 .
600  ASSIGN @File TO "SUSAN"
610  ON END @File GOTO Label2
620  DEALLOCATE Array(*)
```

At first, the array and control block are allocated in the proper order. The ASSIGN statement in line 600 closes the original path and opens a new path with the same name. When the ON END control block for the new path is created, it it placed after the array on the stack. Therefore, no memory space can be recovered by deallocating the array.

Releasing Memory Volumes

BASIC RAM disc memory can be reclaimed without having to do a SCRATCH A which results in the loss of special typing aids and other customizations. However, you must keep in mind that memory can only be reclaimed if no binaries have been loaded after initializing the memory volume. To recover this memory, you would execute a line similar to the following:

```
INITIALIZE ":,0, unit number ",0
```

This, in effect, is equivalent to initializing the volume to 0 sectors to remove it from memory.

The size of a memory volume is 256×n+14 where n is the number of sectors requested in the INITIALIZE statement (unless n is zero).

Memory volumes are allocated in a mark and release stack. What this means is, you only get the memory back if other subsequently created memory volumes are or have been reclaimed. The following program illustrates how this works.

```
100   PRINT SYSTEM$("AVAILABLE MEMORY")
110   INITIALIZE ":,0",4
120   INITIALIZE ":,0,1",4
130   INITIALIZE ":,0,2",4
140   PRINT SYSTEM$("AVAILABLE MEMORY")
150   INITIALIZE ":,0,1",0
160   PRINT SYSTEM$("AVAILABLE MEMORY")
170   INITIALIZE ":,0,2",0
180   PRINT SYSTEM$("AVAILABLE MEMORY")
190   INITIALIZE ":,0",0
200   PRINT SYSTEM$("AVAILABLE MEMORY")
210   END
```

After running this program you would receive results similar to those shown below.

```
434428
431314
431314
433390
434428
```

where:

434428 is the initial size of the free space. This result is displayed after executing line **100**.

431314 is the memory left after creating three memory volumes. This result is displayed after executing lines **110** through **120**.

431314 is the memory left after releasing one memory volume. However, note that this memory volume remains trapped until all subsequent memory volumes have been released. Therefore, the result you see is the same as the previous one. This result is displayed after executing lines **150** and **160**.

433390 is the memory left after releasing two memory volumes. Since there are no subsequent memory volumes following the volume released by executing line **170**, this memory volume and the trapped memory volumes are released.

434428 is the initial size of the free space when you started the program. This result is displayed after executing lines **190** and **200**.

You can re-initialize a removed memory volume in its trapped space provided the newly allocated space is no larger than the original space that was allocated. Otherwise, new space will be allocated for it. (This happens even if enough trapped space exists for the new size.) Here is a program to illustrate this:

```
100   PRINT SYSTEM$("AVAILABLE MEMORY"),"INIT"
110   INITIALIZE ":,0",6
120   INITIALIZE ":,0,1",6
130   INITIALIZE ":,0,2",6
140   INITIALIZE ":,0,3",6
150   PRINT SYSTEM$("AVAILABLE MEMORY"),"0123"
160   INITIALIZE ":,0,1",0
170   PRINT SYSTEM$("AVAILABLE MEMORY"),"-1"
180   INITIALIZE ":,0,2",0
190   PRINT SYSTEM$("AVAILABLE MEMORY"),"-2"
200   INITIALIZE ":,0,1",6 !+2
210   PRINT SYSTEM$("AVAILABLE MEMORY"),"+1"
220   INITIALIZE ":,0,1",0
230   PRINT SYSTEM$("AVAILABLE MEMORY"),"-1"
240   INITIALIZE ":,0,3",0
250   PRINT SYSTEM$("AVAILABLE MEMORY"),"-3"
260   INITIALIZE ":,0",0
270   PRINT SYSTEM$("AVAILABLE MEMORY"),"-0"
280   END
```

When the above program is run results similar to the following are displayed:

```
434186    INIT
427986    0123
427986    -1
427986    -2
427986    +1
427986    -1
432636    -3
434186    -0
```

where:

`434186 INIT` is the initial size of the free space. This result is displayed after executing line **100**. INIT stands for the initial size.

`427986 0123` is the amount of memory left after creating memory volumes zero through three. This result is displayed after executing lines **110** through **150**. The numbers "0123" represent the four memory volumes that were created.

`427986 -1` is the amount of memory left after releasing the unit 1 memory volume. Since subsequent volumes exist, this volume remains trapped until they have been released. Therefore, the available memory displayed is not changed. This result is displayed after executing lines **160** and **170**. The "−1" is displayed for your convenience. A minus (−) indicates that the memory volume has been released, and the 1 is the unit number of the memory volume.

`427986 -2` is the amount of memory left after releasing the unit 2 memory volume. Since subsequent volumes exist, this volume remains trapped until they have been released. Therefore, the available memory displayed is not changed. This result is displayed after executing lines **180** and **190**. The "−2" is displayed for your convenience. A minus (−) indicates that the memory volume has been released, and the 2 is the unit number of the memory volume.

`427986 +1` is the amount of memory left after re-initializing the unit 1 memory volume to its original size. Since the unit 1 memory volume is trapped, you are able to re-initialize it. Note that the available memory displayed is not changed. This result is displayed after executing lines **200** and **210**. The "+1" is displayed for your convenience. A plus (+) indicates that the memory volume has been added, and the 1 is the unit number of the memory volume.

`427986    -1` is the amount of memory left after releasing the unit 1 memory volume. Since subsequent volumes exist, this volume remains trapped until they have been released. Therefore, the available memory displayed is not changed. This result is displayed after executing lines **220** and **230**. The "−1" is displayed for your convenience. A minus (−) indicates that the memory volume has been released, and the 1 is the unit number of the memory volume.

`432636    -3` is the amount of memory left after releasing the unit 3 memory volume. Since subsequent volumes do not exist, this memory volume is released. The memory for the unit 1 and unit 2 memory volumes is also released. Therefore, the available memory displayed is increased by 4650 bytes. This result is displayed after executing lines **240** and **250**. The "−3" is displayed for your convenience. A minus (−) indicates that the memory volume has been released, and the 3 is the unit number of the memory volume.

`434186    -0` is the amount of memory left after releasing the unit 0 memory volume. Since all subsequent volumes are released, this memory volume is also released and the available memory displayed is the initial value. This result is displayed after executing lines **260** and **270**. The "−0" is displayed for your convenience. A minus (−) indicates that the memory volume has been released, and the 0 is the unit number of the memory volume.

Using the same program, remove the exclamation point "!" that is in front of the "+2" comment on line **200**. When you run the program, you will receive results similar to the following:

```
434190    INIT
427990    0123
427990    -1
427990    -2
425928    +1
427990    -1
432640    -3
434190    -0
```

`434190` `INIT` is the initial size of the free space. This result is displayed after executing line **100**. INIT stands for the initial size.

`427990` `0123` is the amount of memory left after creating memory volumes zero through three. This result is displayed after executing lines **110** through **150**. The numbers "0123" represent the four memory volumes that were created.

`427990` `-1` is the amount of memory left after releasing the unit 1 memory volume. Since subsequent volumes exist, this volume remains trapped until they have been released. Therefore, the available memory displayed is not changed. This result is displayed after executing lines **160** and **170**. The "−1" is displayed for your convenience. A minus (−) indicates that the memory volume has been released, and the 1 is the unit number of the memory volume.

`427990` `-2` is the amount of memory left after releasing the unit 2 memory volume. Since subsequent volumes exist, this volume remains trapped until they have been released. Therefore, the available memory displayed is not changed. This result is displayed after executing lines **180** and **190**. The "−2" is displayed for your convenience. A minus (−) indicates that the memory volume has been released, and the 2 is the unit number of the memory volume.

425928 +1 is the amount of memory left after re-initializing the unit 1 memory volume to its original size plus two more sectors. Note that the available memory displayed is changed and it has been decreased by the size of the memory volume. This result is displayed after executing lines **200** and **210**. The "+1" is displayed for your convenience. A plus (+) indicates that the memory volume has been added, and the 1 is the unit number of the memory volume.

427990 -1 is the amount of memory left after releasing the unit 1 memory volume. The available memory displayed is changed to the size of the available memory prior to re-initializing the unit 1 memory volume. This result is displayed after executing lines **220** and **230**. The "−1" is displayed for your convenience. A minus (−) indicates that the memory volume has been released, and the 1 is the unit number of the memory volume.

432640 -3 is the amount of memory left after releasing the unit 3 memory volume. Since subsequent volumes do not exist, this memory volume is released. The memory for the unit 2 memory volume and the original unit 1 memory volume is also released. Therefore, the available memory displayed is changed. This result is displayed after executing lines **240** and **250**. The "−3" is displayed for your convenience. A minus (−) indicates that the memory volume has been released, and the 3 is the unit number of the memory volume.

434190 -0 is the amount of memory left after releasing the unit 0 memory volume. Since all subsequent volumes are released, this memory volume and subsequent memory volumes are released and the available memory displayed is the initial value. This result is displayed after executing lines **260** and **270**. The "−0" is displayed for your convenience. A minus (−) indicates that the memory volume has been released, and the 0 is the unit number of the memory volume.

A result of being able to remove memory volumes is you are able to reclaim memory volume space without losing special typing aids or other customizations which SCRATCH A would undo.

Index

MANUAL COMMENT CARD

BASIC 5.0/5.1
Programming Techniques

HP Part Number 98613-90012 11/87

Please help us improve this manual. Circle the numbers in the following statement that best indicate how useful you found this manual. Then add any further comments in the spaces below. **In appreciation of your time, we will enter your name in a quarterly drawing for an HP calculator.** Thank you.

The information in this manual:

Is poorly organized	1	2	3	4	5	Is well organized
Is hard to find	1	2	3	4	5	Is easy to find
Doesn't cover enough	1	2	3	4	5	Covers everything
Has too many errors	1	2	3	4	5	Is very accurate

---

fold ——

Particular pages with errors? ______________________________

____

Comments: ____________________________________

____

____

____

____

____

____

Name: ____

Job Title: ____________________________________

Company: _____________________________________

Address: ______________________________________

____

□ Check here if you wish a reply.

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 37 LOVELAND, COLORADO

POSTAGE WILL BE PAID BY ADDRESSEE

Hewlett-Packard Company
Attn: Customer Documentation
3404 East Harmony Road
Fort Collins, Colorado 80525

HEWLETT
PACKARD

HP Part Number
98613-90012

Microfiche No. 98613-99012
Printed in U.S.A. 11/87

98613-90665
For Internal Use Only