HEWLETT-PACKARD

HP-UX Technical BASIC Programming Guide

HP-UX Technical BASIC Programming Guide

HEWLETT PACKARD

Edition 1 May 1985

Notice

Portable Computer Division
1000 N.E. Circle Blvd.
Corvallis, OR 97330, U.S.A.

Printing History

Edition 1 May 1985

Table of Contents

Chapter 3

Program Structure and Flow

Chapter 4

Numeric Computation

Chapter 7

Error Handling

Chapter 8

Debugging Programs

Chapter 9

Chapter 10

Chapter 11

Chapter 13

1 Overview

The HP-UX Technical BASIC system runs on the HP-UX operating system. It is a robust BASIC language, containing a generous complement of capabilities.This guide describes how to develop Technical BASIC programs. It covers the range of topics from designing algorithms through writing advanced BASIC programs. Before getting into the technical details of the system, however, you can benefit from looking at what is in this chapter and in this guide.

Chapter Contents

This chapter covers these topics:

- Necessary prerequisites for using this guide.
- A description of what's in this guide.

Prerequisites

Here are the prerequisites that you must meet **before** using the HP-UX Technical BASIC system:

- Hardware must *already* be installed.
- *Both* HP-UX and Technical BASIC software must *already* be installed.
- Previous experience with HP-UX (or UNIX[1]) and BASIC will be helpful, although not a formal prerequisite.

Hardware Installation

All hardware must have already been installed. If not, refer to the installation manual for your model of computer.

Software Installation

The HP-UX system and the Technical BASIC system must have already been installed onto your disc. If the BASIC system is **not** already installed, refer to the HP-UX manuals for your particular system.

HP-UX Knowledge

Although this guide assumes that you have had some previous programming experience and a knowledge of UNIX or HP-UX, you need not have a high level of skill in either area.

In order to use Technical BASIC on the HP-UX operating system, you should be familiar with the following topics:

- How to log in and out of HP-UX (relevant only with multi-user HP-UX systems).
- How the HP-UX file system is structured.

Background information on using HP-UX can be found in the the manuals supplied with your HP-UX system.

1 UNIX is a trademark of AT&T Bell Laboratories

BASIC Knowledge

This *Programming Guide* discusses only the Technical BASIC *language*; it does not describe using the Technical BASIC *system*. For instance, it describes writing programs using the language's features; however, it does not discuss using the system to store the program in a file, get a printed listing of the program, or run the program. Such operations are described in the *Getting Started* manual for Technical BASIC on your particular HP-UX system. **If you have not yet read that manual, you should do so before getting very far into the details of this guide.**

If you have never programmed a computer before, you will probably be more comfortable starting with one of the many beginner's BASIC text books available from various publishing companies. However, some beginners may find that they are able to start in this guide 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 desktop computers, you may start faster by going directly to the *HP-UX Technical BASIC Language Reference* and checking the keywords you normally use. If you don't find the keywords you expect to find, then refer to the Table of Contents or Index for the appropriate topic.

Once you have satisfied the above prerequisites, you are ready to being using Technical BASIC on the HP-UX operating system.

What's In This Guide?

This section describes the contents of this guide. It discusses these topics:

- What this guide contains
- How it is organized
- What it **does not** contain
- How to read the guide
- Previews of each chapter

What this Guide Contains

This guide provides programming techniques, helpful hints, and explanations of capabilities. It mainly consists of examples of BASIC algorithms used to perform programming tasks. Any BASIC statements appropriate to the topic being discussed are included in each chapter, whether they have been previously introduced or not.

How Is It Organized

The explanations and programming hints in this guide are organized **topically**. It reflects the organization of a well-written program: A program performs various "sub-tasks" as it completes its overall job, so many of these tasks can (and should) be viewed separately to be understood more easily and used more effectively. Here are two examples:

- Perhaps you have reduced your favorite formula to program form and now want a graph of the results. You will find a chapter called "Graphics" that explains many ways to generate plots and graphs.
- 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 *HP-UX Technical 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.

What this Guide Does Not Contain

This guide is not a rigorous description of the BASIC language; that is provided by the *HP-UX Technical BASIC Language Reference*. 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 syntactic 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 language reference. It is specifically intended for that purpose.

How to Read This Guide

All readers should read this "Overview" chapter. Since you are now reading this discussion, you will have already learned much about the Technical BASIC documentation set. Whether or not you decide to read *any* of the other chapters of this manual, you will *definitely* benefit from reading the rest of this chapter.

Once you have finished reading the subsequent "Chapter Previews" section, you should realize that many of the keys to designing Technical BASIC programs are revealed in the chapter called "Program Development." All readers should read that chapter also.

Once you feel comfortable using the system, you may choose to read only the chapter(s) and section(s) that are relevant to your programming tasks. Since most users will not read this guide from cover to cover anyway, this approach 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 about that topic.

Use the Overviews We have attempted to provide many "overviews" and "previews" so that you can determine the contents of a chapter or section and then decide whether or not to read them. For instance, most introductions to chapters and sections provide a "bulleted list" of what the subsequent text describes. Here is an example:

Bulleted lists provide the following features:

- They "stand out" visually, allowing you to scan text and easily find them.
- They provide a way of quickly delimiting several key points.
- They allow you to maintain a "global" view, rather than getting bogged down in details.

These lists will help you to more quickly learn the material provided in the text, as well as steer you around sections that are irrelevant to you at a particular time.

Chapter Previews

The following is a preview of each chapter in this guide.

Chapter 1: Overview

This chapter (the one you are now reading) covers the necessary prerequisites for using this guide and a description of each chapter.

Chapter 2: Program Development

This chapter discusses the overall process of developing programs. It briefly discusses the steps in designing, coding, debugging, and supporting programs.

Chapter 3: Program Structure and Flow

This chapter describes how you can tell a program to make decisions and then execute the corresponding part(s) of your program.

Chapter 4: 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 memory.

Chapter 5: String Manipulation

This chapter explains the programming tools available for processing strings. Strings are 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.

Chapter 6: User-Defined Functions and Subprograms

This chapter discusses an outstanding feature of this language – its ability to call other program contexts and the speed with which it can do so. Alternate contexts, or environments, are available as user-defined functions or subprograms.

Chapter 7: Error Handling

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 well written program. Error trapping keeps the program running and provides valuable assistance to the computer operator.

Chapter 8: Program Debugging

This chapter explains the powerful debugging features available on the Technical BASIC system. 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 dream. The next best thing is to get the computer to do most of the debugging work for you.

Chapter 9: Communicating with the Operator

This chapter provides suggestions and techniques for providing information to and receiving it from a person who is using the computer. For instance, it discusses techniques useful for creating organized, highly readable printouts on printers and display screens.

Chapter 10: 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. The two main means for storing and retrieving data are program files and mass storage data files.

Chapter 11: Program Debugging

This chapter explains the powerful debugging features available on the Technical BASIC system. 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 dream. The next best thing is to get the computer to do most of the debugging work for you.

Chapter 12: Graphics

This chapter discusses the techniques of programming with graphics and how they are very useful for displaying data in a form that humans easily understand.

Chapter 13: Binary Programs

This chapter explains how to make calls to programs written in other languages from Technical BASIC programs.

2 Program Development

There are several stages in the design and development of programs: determining what is required; outlining algorithms and data structures; translating the algorithms and data structures into BASIC statements (coding); checking to see that the program works without errors (debugging); documenting the program; and supporting and enhancing it.

There are also several steps in the mechanics of program development once you begin coding the program: entering, storing, listing, and editing it.

Chapter Contents

This chapter contains the following areas of program development:

General Steps in Program Development

This section describes the general steps that you will take as you develop programs. The following sections in this chapter and manual show how to apply them.

1. **Understand and describe the problem.** You cannot design a solution to a problem without clear understanding of the problem. You must get a clear grasp of these two elements:

   a. What **action** the program is required to perform.

   b. What **data** it will be given or will compute.

1. Even though this step sometimes seems trivial or obvious, taking time to methodically describe it may simplify the design process immensely.
2. **Outline the solution.** This phase initially consists of verbally describing what *steps* your program will take to solve the problem. At this point, splitting up the program's responsibilities into various tasks is very important, especially if each task is to be written by a different programmer. Interactions between various tasks are also important.
3. **Design algorithms and data structures, and then refine.** The algorithms that you design (or choose from computing literature) will consist of the individual steps that your program will take. The data structures required may be global or applicable to only one or two steps of the overall solution.

   The "refine" part of this step suggests the highly effective approach of beginning with large tasks and then breaking each one up into smaller tasks. You can keep repeating this approach on each subtask, until you have broken the problem up into a set of rudimentary steps.
4. **Code the data structure and algorithms.** This step involves translating the data structure and algorithms defined in earlier steps into the programming language that you will be using – in this case, BASIC. This guide gives several examples of programs and code segments which are implementations of various algorithms.

5. **Debug and test the program.** Debugging a program involves getting the program to run without crashing and giving you BASIC error messages like `Error 56: STRING OVF`. Testing a program involves making sure that it does what you want it to do. Ideally, you should be able to test each separate "module" of your program independently, which is sometimes called "bottom-up" testing. The chapter called "Debugging Programs" discusses these topics in greater detail.
6. **Document and support it.** Documenting the program involves telling both the program's users and its future supporters what the program does. See the section of this chapter called "Documenting Your Programs" for further details. Support involves fixing bugs and enhancing the program.

Sample Development Session

The remaining sections of this chapter, and some subsequent chapters, show how to apply the suggestions in the preceding section in developing a simple "budget" program. Seeing that the suggestions really work will help you to have faith in them and begin to apply them.

Step 1: Understand and Describe the Problem

As you begin to conceptualize your budget program, one of the first question you might ask is, "What is a budget for?" The answer is that it is an attempt to evaluate present income and spending for the purpose of planning and controlling future income and spending. OK. But how does it help you accomplish these two tasks? Here some things that will help you to evaluate present income and expenses.

1. Determine target income and expenses.
2. Determine actual income and expenses.
3. Calculate the differences between the targets and actuals.

For simplicity, using this information in planning and controlling future income and spending will not be part of the program's responsibility.

Now that the program's action has been generally defined, you are ready to ask the second question: "What data is required?" Here is an example of the **data that the user will supply to the program** (also known as input data):

```
Income
           Category            Target      Actual
         ---------------     --------    --------
         Payroll             1680.56     1680.56
         Investments          345.67      290.32
Expenses
           Category            Target      Actual
         ---------------     --------    --------
         Mortgage             654.32      654.32
         Taxes                432.10      432.10
         Insurance            123.45      123.45
         Food                 432.00      501.81
         Medical               75.00      125.90
         Transportation       165.00      134.32
         Education            100.00       95.00
         Entertainment         75.00       98.55
```

Alphanumeric data will be used in the "Category" column; with BASIC, this data type is also known as string data. The data in the "Target" and "Actual" columns will be real numbers.

Here is an example of the **data that the program will compute and display for the user** (also known as output):

```
Income                                Difference
          Category            $          %
          ---------------  --------   --------
          Payroll            0.00          0
          Investments      - 55.35       - 16
                           --------
          Total difference - 55.35

Expenses
                Difference
          Category            $          %
          ---------------  --------   --------
          Mortgage           0.00          0
          Taxes              0.00          0
          Insurance          0.00          0
          Food             + 69.81       + 16
          Medical          + 50.90       + 68
          Transportation   - 30.68       - 19
          Education        -  5.00       -  5
          Entertainment    + 23.55       + 31
                           --------
           Total difference +108.58
  Net Savings  Deposit
               (Withdrawal)      (163.93)
```

The "$ differences" will be represented as real numbers, since it may be important to keep track of cents as well as dollars. The "% differences" can be integers, since 1% resolution is probably adequate.

Now that you understand and can describe the problem, both in terms of what it does and what data is required, you are ready to begin creating the solution.

Step 2: Outline the Solution

Here are the general steps that a program can take to solve the problem:

1. Show you the target income and expenses (for all categories).
2. Ask you for the actual income and expenses (for each category).
3. Compute the differences between target and actual (for each category).
4. Show you the results (for all categories).
5. Compute the difference between total income and total expenses.
6. Show you the net deposit to (or withdrawal from) your savings.

Maintain Proper Perspective

At this point, you should **not** get into the details of any step; just stick to the **broad perspective**. You will be getting into more details in the next step. By **hiding the details** of each step in a procedure, you can more easily understand what is happening in the procedure (and consequently maintain better control of it).

Let the Data Structure the Algorithms

You can now begin to see that the steps are more or less structured according to the data that you have. This is in fact one of the most important principles that you can use in designing your programs: Let the structure of the data determine the structure of the algorithms.

Step 3: Design, and Then Refine

You have shown what general steps the program will take to solve the problem, but you have not yet described the procedure explicitly enough for a computer to understand what you want it to do (at least not for the Technical BASIC language). The next step is to take each of these general steps and begin to refine it, or break it down into smaller and smaller tasks.

For brevity, let's take the first step:

1. Show you the target income and expenses (for all categories), and break it down into smaller steps.
   - **a.** For each income category:
     - i. Determine the target income value.
     - ii. Display the category name and target value.
   - **b.** For each expense category:
     - i. Determine the target expense value.
     - ii. Display the category name and target value.

With this level of detail you can begin translating the algorithm into BASIC code.

Step 4: Code the Program

This section consists of two parts:

- The first part gives a brief description of the fundamental building blocks of a BASIC program.
- The second part, called "Back to Step 4," presents a BASIC program.

If you already know another version of BASIC, or other programming language, then you may want to skim the first section, called "Elements of a BASIC Program."

Elements of a BASIC Program

This section describes the fundamental building blocks of Technical BASIC programs. These terms and concepts will prepare you for translating your data structure and algorithms into BASIC code.

Keywords A keyword is a group of characters that is understood by the BASIC language system to invoke some predefined action. Examples of keywords are BEEP, DISP, INPUT, and LET.

Statements A statement is a keyword (sometimes optional) followed by any parameters, lists, specifiers, and secondary keywords that are allowed with that keyword. These are examples of statements:

`BEEP`	Tells the computer to produce a short beep.
`DISP "THIS IS A STATEMENT"`	Tells the computer to display the message "THIS IS A STATEMENT" on the screen.
`INPUT Income`	Tells the computer to allow the user to enter a value (from the keyboard) into the numeric variable named Income.
`LET Expense=10`	Tells the computer to assign a value of 10 to the numeric variable named Expense.

Note that the notation used in this guide is to print the statements that you can actually type into the computer in a special dot-matrix font.

Program Lines A program line contains a line number followed by at least one BASIC statement. Here are two legal program lines.

```
10 PRINT "THIS IS A PROGRAM LINE"
20 END !  SO IS THIS
```

Line 10 prints the characters between the quotes, while line 20 indicates the end of the program. The text following the END statement on line 20 is a comment; it is separated from the END statement with the exclamation point.

You can place several statements on a single program line by separating them with @ character.

```
10 PRINT "THIS IS A " @ PRINT "PROGRAM LINE"
```

A line number may be optionally followed by a line label. A line label is a name that is placed after the line number and is terminated by a colon (:).

```
20  Done: END !  SO IS THIS
```

The subsequent section called "Documenting Your Programs" further describes using comments and line labels.

Programs In Technical BASIC, a program is a list of program lines, usually with an END statement on the last line. The two following program lines define a program.

```
10 DISP "THIS IS A PROGRAM LINE"
20 END ! SO IS THIS
```

The maximum length of a program line entered from the keyboard is 159 characters[1]. Note that you should check the *Implementation Specifics* appendix for your particular HP-UX Technical BASIC system.

Data Types With Technical BASIC, there are two general pre-defined types of data: numeric and alphanumeric (or "string"). And within the numeric data category, there are two divisions: real[2] numbers and integers. Here are examples of each.

Here are examples of creating storage locations, called variables, for these fundamental types of data.

String	Real	Integer	Short
a Word MORE letters	1.2 1E+300	16 32 767	−1.987 1E+10

`DIM StringVar$[20]`	Declares a simple variable named StringVar$ to be of type string, and allocates a storage space of size 20 characters for it.
`INTEGER WholeNumber`	Declares a simple variable named WholeNumber to be of type INTEGER, and allocates the corresponding amount of memory for it.
`SHORT ShortReal`	Declares a simple variable named ShortReal to be of type SHORT, and allocates the corresponding amount of memory for it. Note that real variables of type SHORT are stored in half the memory that it takes for a variable of type REAL.
`REAL LongReal`	Declares a simple variable named LongReal to be of type REAL, and allocates the corresponding amount of memory for it.

1 BASIC program lines longer than 159 characters can be created using another editor and then retrieved with the GET statement; however, only the first 159 characters will be stored on the line.

2 With HP-UX Technical BASIC, there are actually two pre-defined representations of real numbers: REAL and SHORT. The difference between the two is the range of values that they can represent. See the chapter called "Numeric Computation" for further details.

`REAL RealArray(10)` Declares an array variable named RealArray to be of type REAL, and allocates the corresponding amount of memory for it. The array structure in BASIC is a group of variables, each of which has the same data type and variable name. Each variable in the array is specified by an index value; for instance, the 4th element is RealArray(4), if OPTION BASE 1 is in effect. Note that OPTION BASE determines the lower bound(s) of a numeric or string array's subscript(s); the default is OPTION BASE 0.

You can also use these fundamental types to implement your own data types, if you wish.

Functions Functions perform operations that always return a value. The Technical BASIC system provides two types of functions:

- Resident – provided by the system.
- User-defined – you can implement these yourself.

Resident functions are part of the BASIC language. For instance, `SIN(PI/2)` and `CHR$(10)` are examples of calling the resident functions SIN and CHR$, respectively. Resident functions are discussed in the "Numeric Computations" and "String Manipulations" chapters.

You can implement your own user-defined functions to provide any function you desire. These types of functions are described in the "User-Defined Functions and Subprograms" chapter.

Subprograms Subprograms also perform operations, but they do not necessarily return a value. Like programs, subprograms are also lists of program lines, but they can be stored independently and "called" from a main program or another subprogram. Each is also stored in its own portion of BASIC memory, which is *separate* from the main program. Here is a simple example subprogram:

```
100  SUB "FirstSub"
110  DISP "This is displayed by 'FirstSub'."
120  SUBEND
```

Here is how you can call it from a main program.

```
100 CALL "FirstSub"
```

Subprograms are a useful programming tool, but the computer is capable of running just fine without them. Subprograms are covered in depth in the chapter called "User-Defined Functions and Subprograms".

Binary Programs The Technical BASIC system has the capability of loading and calling "binary programs". The term "binary programs" is used to identify programs that are stored in the "machine" language used by the computer's central processor, rather than in a high-level language like BASIC. Thus, "binary" programs can be run directly by the processor, rather than having to be translated from the high-level language into machine language.

The usual purpose of a binary program is to add capabilities to the language of the computer. In this respect, the computer's operating system and the Technical BASIC system might be considered "binary programs". However, they cannot be accessed using the CALLBIN statement. For further details read the "Binary Programs" chapter.

Commands (Not Part of Programs) Commands are like statements in that they consist of a keyword, and sometimes appropriate parameters; however, they **cannot** be stored in a program line – you can only execute them from the keyboard. Examples of commands are DELETE and SCRATCH.

Back to Step 4

Now that you have seen the building blocks of a program, you are ready to begin translating it into BASIC language code. For convenience, here are copies of the data structure and algorithm from the solution presented earlier. The translation into BASIC code follows:

Data structure:

```
Income
           Category           Target
         ---------------    --------

           Payroll           1680.56

           Investments        345.67
```

Algorithm:

a. For each income category:

i. Determine the target income value.

ii. Display the category name and target value.

A Coded Program Segment Here is one way of implementing the data structure and algorithm. (Don't be concerned if you don't understand every line of the program right now; each line is explained after the program listing.)

```
100  ! Allocate memory for data storage.
110  OPTION BASE 1
120  DIM IncomeName$(2)
130  REAL TargetIncome(2)
140  !
150  ! Assign values to variables.
160  LET IncomeName$(1)="Payroll"
170  LET IncomeName$(2)="Investments"
180  LET TargetIncome(1)=1680.00
190  LET TargetIncome(2)=345.67
200  !
210  DISP " Category                 Target"
220  DISP "----------               --------"
230  DISP IncomeName$(1),TargetIncome(1)
240  DISP IncomeName$(2),TargetIncome(2)
```

Here are the results of running the program:

```
 Category               Target
----------             --------
Payroll                 1680
Investments             345.67
```

Here is a line-by-line description of what the program does. You can skip the explanation if you already understand what is happening.

Line 100 is a comment. It is a way for the programmer to describe what he is doing in the program; they help him to understand what the program is doing the next time that he edits it. It is especially useful when he, or someone else, must modify the program a long time later.

Line 110 defines the lower bound of array element indexes. In this case, a value of 1 dictates that the first element of an array is specified with an index value of 1; e.g., IncomeName$(1). OPTION BASE 0 indicates that the first element would be specified with an index value of 0; e.g., IncomeName$(0).

The DIM statement on line 120 declares the string variable named IncomeName$, and allocates memory for it. In this case, the *(2)* specifies that it is an array variable with 2 elements (1st, and 2nd) which are both string variables.

The REAL statement on line 130 performs the same function for the real array called ActualIncome.

Lines 140 and 150 are also comments.

The LET statement on line 160 stores the characters "Payroll" in element 1 of the string array variable named IncomeName$. Line 180 performs a similar function for element 1 of the REAL array named TargetIncome. Lines 170 and 190 perform similar operations for element 2 of the respective arrays.

Line 200 is another comment.

The DISP statements on lines 210 through 240 display messages on the CRT screen. The first two DISP statements display the table headings, while the second two display values in the table. On line 230 the index value of 1 specifies that 1st element of the IncomeName$ string array is to be displayed. This string variable's value is "Payroll". Similarly, the value of the 1st element of the TargetIncome array is 1680.

Mechanics of Program Development

Now that you have a real program to work with, the next step is to learn the mechanics of entering, storing, listing, and running programs. Some of these operations are system-dependent; in other words, they vary slightly according to the HP-UX system you are using. Therefore, they have been covered in the *Getting Started* manual for your particular HP-UX Technical BASIC system.

Here is the list of operations covered therein:

- Initial program entry using the Technical BASIC editor, including specifics of using your keyboard.
- Storing the program in a file (STORE or SAVE).
- Checking to see if the file was stored (CAT).
- Getting a listing of the program (LIST and PLIST).
- Running the program (RUN).
- Getting a hardcopy of the screen (DUMP ALPHA and DUMP GRAPHICS).
- Dealing with error messages.

If you are not familiar with these operations, please refer to your *Getting Started* manual now.

2

Global Program Editing

The *Getting Started* manual for your particular Technical BASIC system describes entering programs using the BASIC editor; it also describes editing programs on a line-by-line basis. This section describes the following "global" program editing operations which are provided by Technical BASIC keywords:

- Inserting new program lines between existing lines.
- Deleting existing lines.
- Renumbering existing lines.
- Scanning for string literals.
- Renaming variables.
- Copying and moving program segments.

Inserting Lines Lines can be easily inserted into a program. As an example, assume that you want to insert some lines between line 200 and line 210 of our example program.

```
 .
 .
 .
180  LET TargetIncome(1)=1680.00
190  LET TargetIncome(2)=345.67
200  !
210  DISP " Category                 Target"
220  DISP "----------               --------"
 .
 .
 .
```

You can begin by numbering the first line 201, the second one 202, and so forth (up through 209 without overwriting existing lines).

```
201  CLEAR ! Clear the alpha screen.
202  !
```

Note that while inserting lines, you should keep track of the line numbers you have inserted so that you do not inadvertantly:

- Overwrite existing lines.
- Insert lines into the wrong place.

You can generate a program listing with LIST or PLIST to keep track of where lines have been placed.

```
   .
   .
   .
 180  LET TargetIncome(1)=1680.00
 190  LET TargetIncome(2)=345.67
 200  !
 201  CLEAR ! Clear the alpha screen.
 202  !
 210  DISP " Category                  Target"
 220  DISP "----------               --------"
   .
   .
   .
```

Deleting Lines The DELETE command can be used to delete single or groups of program lines. When the keyword DELETE is followed by a single line number, only a single line is deleted. For example, executing:

```
DELETE 201
```

deletes only line 201 of your program.

Blocks of program lines can be deleted by using two line numbers in the DELETE command. The first number identifies the start of the segment to be deleted, and the second number identifies the end of the segment to be deleted. Here are some examples.

`DELETE 100,200`	deletes lines 100 thru 200, inclusively.
`DELETE 150,65535`	deletes all the lines from line 150 through the end of the program.
`DELETE 100,10`	would do nothing except generate an error if a program is currently in memory.

Renumbering a Program No matter how careful you have been while entering lines, there will inevitably be a time when you need to renumber a program. And it is also good practice to renumber occasionally to improve readability.

You can renumber a program by using the REN command. When no parameters are specified, the first line number is renumbered to 10 and the line-number increment is 10.

Both the **starting line number** and the **interval between lines** can be specified. For example, this command renumbers the entire program, using 100 for the first line number and an increment of 5.

```
REN 100,5

100  ! Allocate memory for data storage.
105  OPTION BASE 1
110  DIM IncomeName$(2)
115  REAL TargetIncome(2)
120  !
125  ! Assign values to variables.
130  LET IncomeName$(1)="Payroll"
135  LET IncomeName$(2)="Investments"
140  LET TargetIncome(1)=1680.00
145  LET TargetIncome(2)=345.67
150  !
155  CLEAR ! Clear alpha display.
160  !
165  DISP " Category                Target"
170  DISP "----------              --------"
175  DISP IncomeName$(1),TargetIncome(1)
180  DISP IncomeName$(2),TargetIncome(2)
```

If **only the beginning line number** is specified, a line-number increment 10 is assumed. For example, this command renumbers the entire program using 1000 for the first line number and an increment of 10:

```
REN 1000

1000  ! Allocate memory for data storage.
1010  OPTION BASE 1
1020  DIM IncomeName$(2)
1030  REAL TargetIncome(2)
```

```
1040  !
1050  ! Assign values to variables.
1060  LET IncomeName$(1)="Payroll"
1070  LET IncomeName$(2)="Investments"
1080  LET TargetIncome(1)=1680.00
1090  LET TargetIncome(2)=345.67
1100  !
1110  CLEAR ! Clear alpha display.
1120  !
1130  DISP " Category                Target"
1140  DISP "----------              --------"
1150  DISP IncomeName$(1),TargetIncome(1)
1160  DISP IncomeName$(2),TargetIncome(2)
```

You can also **renumber a portion of a program**. For instance, this command renumbers only line numbers 1000 through 1080 to lines 10 through 90.

```
REN 10,10,1000,1080

10   ! Allocate memory for data storage.
20   OPTION BASE 1
30   DIM IncomeName$(2)
40   REAL TargetIncome(2)
50   !
60   ! Assign values to variables.
70   LET IncomeName$(1)="Payroll"
80   LET IncomeName$(2)="Investments"
90   LET TargetIncome(1)=1680.00
1090  LET TargetIncome(2)=345.67
1100  !
1110  CLEAR ! Clear alpha display.
1120  !
1130  DISP " Category                Target"
1140  DISP "----------              --------"
1150  DISP IncomeName$(1),TargetIncome(1)
1160  DISP IncomeName$(2),TargetIncome(2)
```

Note that the REN command **cannot** be used to move lines. Moving and copying program lines is the topic of a subsequent section.

To get back to the original program, you can use this sequence:

```
DELETE 1110,1120
REN 100
```

Scanning for Literals The SCAN command is used for finding all the occurrences of a particular string literal or variable name in a program. In our continuing example program, let's look for the literal "Income":

```
SCAN "Income"
```

Here is the system's response:

```
Scanning ...
130  REAL TargetIncome(2)
180  LET TargetIncome(1)=1680.00
190  LET TargetIncome(2)=345.67
210  DISP " Category                  Target"
230  DISP IncomeName$(1),TargetIncome(1)
240  DISP IncomeName$(2),TargetIncome(2)
...end of scan
```

Here is a more useful example. Suppose that you have the string literal "Tax" in several places in your program, and you want to change it to either "State Tax" or "Federal Tax" – and which one you change it to depends on the context of the statement. Use the following command to find and list all occurrences of the string "Tax":

```
SCAN "Tax"
```

You can then look at each line and decide whether it should be changed to "State Tax" or "Federal Tax".

To verify that all change(s) have been made, execute another SCAN command specifying the string for which you were originally searching. (Using this command avoids a long listing of the program.) The command lists all program lines containing the string "State Tax" or "Federal Tax", since the string "Tax" is a subset of those strings.

Renaming Variables You can rename variables with the the REPLACEVAR...BY command. Here is an example:

```
REPLACEVAR TargetIncome BY TargetExpense
```

REPLACEVAR...BY is like SCAN in that it looks for specific patterns of characters; however, REPLACEVAR is different in two ways:

- It can *only* find occurrences of the specified *variable name,* not any combination of characters in the program.
- It *automatically replaces* the first variable name with the second one.

Here is an example of replacing one variable name with another. Suppose that you have the following program in memory:

```
10 A=20
20 B=30
30 T=A+B
40 DISP T
50 T=A*B
60 DISP T
70 END
```

You decide after entering the program that you want to replace the variable "T" with the variable name "RESULT" , but you do not want to go through the program and replace "T" with "RESULT" everywhere you see it as it would take a long time to do so. This is particularly true for large programs. The following command allows you to do this:

```
REPLACEVAR T BY RESULT
```

When you do a listing of you program it now looks like this:

```
10 A=20
20 B=30
30 RESULT=A+B
40 DISP RESULT
50 RESULT=A*B
60 DISP RESULT
70 END
```

Copying and Moving Program Segments

During program development you often enter a section of code that performs some function, thinking that this function will be needed at that place. Sure enough, a short time later you find that you need to move it to another location. But how on earth do you move those thirty-five lines of code? You certainly do not want to retype the whole thing.

The following paragraphs show you how to move program segments using three different methods:

- Using the Technical BASIC editor (if you have a terminal or console that supports screen-oriented editing).
- Using Technical BASIC's STORE and MERGE commands.
- Using the HP-UX system's "vi" editor

Moving Lines with the Technical BASIC Editor

You can only use this method if you have a terminal or console that supports screen-oriented editing. See the *Getting Started* manual for your particular Technical BASIC system to determine whether your terminal or console supports this feature.

Here are the steps that you will be taking with this method:

1. Perform a LIST operation on the lines to be moved.
2. Change a statement's line number by moving the cursor onto the line and typing over the existing number.
3. Store the line by pressing the carriage return key.
4. Repeat steps 2 and 3 until you have changed the line numbers of all program lines to be moved. (You may have to perform another LIST if you are moving more than a screenful of lines.)
5. Verify that the lines have been copied into the desired location. (Note that the original lines are still present.)
6. DELETE the original copies of the lines. (If you are duplicating the lines into another location, then you will skip this step.)

Moving Lines with the MERGE Command The following procedure allows you to move a program segment using the Technical BASIC's MERGE command.

1. Store the program which you have just entered under a file name of your choosing by executing a STORE command that specifies the desired file name.
2. To assist in determining where the lines are you wish to move, list the program using the LIST statement. Make a note of those lines for later reference.
3. Delete *all* the lines in your program *except* the those which you wish to move to another location in the program. Use the DELETE command, which was explained earlier in this section.

   If, for instance, you want to move lines 300 through 390 to another location, you could execute:

   ```
   DELETE 1,299
   ```

   and then execute

   `DELETE 391,`*last_line*

4. Store the remaining lines in a temporary file. Use the STORE command and specify the name of the temporary file; this file's name must be **different** from the one you stored in step 1.
5. Reload the original program.
6. Delete the lines in this file that you want to move. (If you are just making a second copy of these lines, you can skip this step.)

   If, as in the preceding example, you were moving lines 300 through 390, you would now delete these lines:

   ```
   DELETE 300,390.
   ```

7. Finally, merge the lines stored in the "temporary" file into the new location in original program.

   Use the MERGE command and specify the temporary file's name. Following the file name, specify the line number where you want the insertion to occur and the increment for each line. Note that the increment of 1 is used so that lines of the existing program are not overwritten and that lines are not "interleaved" between existing lines of code. For example, if you were merging the contents of the file named TEMP into this program, beginning at line 850, you would specify that line number and the increment of 1 after the file name. Your statement would be specified as follows:

```
MERGE "TEMP" 850,1
```

After executing the above procedure, purge the temporary file by using PURGE.

Moving Lines with the HP-UX vi Editor The editors available on your HP-UX system read and write text using ASCII-format files. The Technical BASIC system can also read and write ASCII files using the GET and SAVE commands. The general procedure you will use is as follows:

1. Create a program with the BASIC editor, SAVE it in an ASCII file, and exit the BASIC system.
2. Read this file with an HP-UX editor, and move the desired lines. Note that your program lines and statements refering to them have to be renumbered. Then store this modified file, and exit the HP-UX editor.
3. Load the modified file into BASIC memory with the GET command.

Here are the details of the above procedure using the HP-UX system's *vi* editor.

1. While in BASIC, use the SAVE command to store the program. This command creates an ASCII file and stores the program as ASCII text in the file.
2. Exit the BASIC system.
3. Execute the *vi* command, specifying the name of the file saved in step 1.
4. Locate the program lines you want to move and place the cursor on the first line to be moved. Next, type the number of program lines you wish to move followed by the uppercase letter *Y* (for "yank"). This will place the indicated number of program lines into the *vi* editor's buffer.

5. Next, move the cursor to the line above which you wish to copy the text contained in the buffer. Type an uppercase *P* (for "put"). Your program lines have been moved; however, the same program lines still exist in their previous location and need to be removed from the program. To remove these line place the cursor on each of the lines to be removed and type lowercase *dd*.
6. Renumber the program lines and statements refering to them.
7. Store the file by typing uppercase *ZZ*. Typing this command also exits the *vi* editor.
8. Enter the BASIC system. Execute a GET command, specifying the name of the modified file.

Step 5: Debug and Test

This phase of the development process involves two main things:

- Getting the program to run (without program-execution errors).
- Making sure the program does what is expected.

Since these are rather large topics, they are discussed in the separate chapter called "Debugging". Both debugging and testing programs are mentioned there, but the focus is on the Technical BASIC features available for debugging. An exhaustive treatise of software testing is beyond the scope of this manual.

Step 6: Document and Support

Documentation for a program describes relevant facts about the program, such as what the program does and what kind of data it requires. Support involves both fixing errors and adding enhancements. Documentation is described in this chapter, but a detailed treatment of software support is beyond the scope of this text.

There are basically two types of documentation that you can produce for your programs:

- Internal documents – those available to someone supporting or enhancing the program.
- External documents – those available to the program's users.

Internal Documents

There are basically two ways to document programs for those who will be supporting or revising them:

- Within the program itself.
- In a separate document.

The focus of this section is on self-documenting programs. The topic of producing separate documents, while extremely useful, is not discussed in this manual.

Internally Self-Documenting Programs

When first learning how to program, many people may view the use of comments, long variable names, descriptive printouts, and other documentation tools as merely extra typing that is not really necessary in their short programs. However, as old programs are expanded or become more widely used, or new programs are written, software support activities eventually become necessary. For example, when some obscure bug is found, someone must address the problem.

A programmer (often the original designer) picks up a copy of the program written a year ago and can't begin to see what "X1" was or why you would ever want to divide it by "X2". Program documentation can make the difference between a supportable tool that adapts to the needs of the users and a support nightmare that never really does exactly what the current user wants. Keep in mind that the local software support person just might be you.

Relevant Features The Technical BASIC language on HP-UX makes it easy to write self-documenting programs. In addition to BASIC's standard REM (remark) capability, its primary documentation features are as follows:

- Line labels (up to 32 characters)
- Variable names (up to 32 characters)
- Remark (REM) statements
- End-of-line comments (that follow statements on a program line)
- Indentation of statements on program lines

Although this section deals primarily with commenting methods, all of these features work together to make a readable program.

A Comparison The following example shows two versions of the same program.

- The first version is uncommented and uses "traditional" BASIC variable names.
- The second version uses the features of the Technical BASIC language to make the program more easily understood.

After reading both programs, answer this question: **Which version would you rather work with?**

```
100  A=0.03
110  B=0.02
120  C=A+B
130  D=0
140  DISP "Input item price" @ INPUT D
150  IF D<0 THEN GOTO 210
160  E=D*C
170  F=D+E
180  DISP "Tax =";E,"Item cost =";F
190  DISP
200  GOTO 130
210  END
```

```
100  ! ********************************************
110  ! This program computes the sales tax
120  ! for a list of prices.
130  !
140  ! Input:  Item prices are input individually.
150  !
160  ! Output: The tax and total cost for
170  !         each item are displayed.
180  !
190  ! Program terminated with negative cost.
200  !
210  ! Sales tax rates are assigned on lines 270
220  ! and 280. The rates used in this version
230  ! of the program were in effect 9/1/84.
240  !
250  ! ********************************************
260  !
270  State_tax=.03 !     Local tax rates
280  City_tax=.02
290  !
300  Tax_rate=State_tax+City_tax
290  !
300 Get_price: !         Start of main loop
310    Price=0 !         Don't change totals if no entry
320    DISP "Input item price (negative price to quit)."
330    INPUT Price
340    IF Price < 0 THEN GOTO Finished
350    Tax=Price*Tax_rate
360    Item_cost=Price+Tax
370    DISP "Tax =";Tax," Item cost =";Item_cost
380    DISP
390  GOTO Get_price !    Repeat loop for next item
400  !
410 Finished: END
```

There are two methods for including comments in your programs. The use of an exclamation point is demonstrated in the second example program. The exclamation point marks the boundary between an executable statement and comment text. There does not have to be an executable statement on a line containing a comment. Therefore, the exclamation point can be used to introduce a line of comments, to add comments to a statement, or simply to create a "blank" line to separate program segments. Exclamation points may be indented as necessary to help keep the comments neat.

Note that when the exclamation point is used on the same line as syntax, the exclamation point and comment are moved a space away from the syntax by the interpreter. If you wish to make your comment stand out from the syntax on that line, you need to use blank spaces between the exclamation point and the comment. An example of this was shown in the previous program.

The REM statement can also be used for comments. The exclamation point is neater and more flexible, but the REM statement provides compatability with other BASIC languages. The REM keyword must be the first entry after the line identifier and must be followed by at least one blank. Here are some examples of comments.

Obscure	Better
`20  REM bal. array`	`20  REM Check Book Balance`
`50  X=PI*R^2`	`50  X=PI*R^2 @ REM Area of circle`

General Suggestions on Comments Each programmer has an individual style in the use of comments. Therefore, the following are not formal rules – they are simply some general suggestions on the effective use of comments.

- Include a heading on programs that tells the purpose of the program: Why was the program written? What does it do? Who will probably be using it?
- Give any helpful support information, such as the author of the program, the revision date, where to call or write for help, and instructions for any modifications that might be made by a normal user.
- Identify all significant variables, especially global variables. A descriptive variable name may do the job, or a more detailed explanation may be needed.
- Describe any input or output devices that are required for the proper running of the program. This may even include an explanation of how to modify the program to accommodate alternate devices (when such changes are reasonable).

- Make major segments and entry points visible. Many tools are available for this, including descriptive line labels, indenting (described in the next section), spacing, and comments describing program flow.
- Use comments freely to describe the action of complex lines, equations, fancy manipulations, and "low-level" operations like escape code sequences. These heavily coded operations can be very important to the computer and very mysterious to the human trying to read the program.

Indenting Indenting is used to make the structure of a program more intuitively obvious by placing program lines in their "appropriate places". "Appropriate places" means indenting whenever there is a beginning or end of a program statement which:

- causes looping,
- is conditionally executed,
- is a separated program segment (such as a function),
- is the first character of each program line contained in that segment – excluding the line number.

The following program is an example of indenting and commenting (but a poor example of variable names):

```
10 PRINT " I  J" !                       Prints heading.
20 FOR I = 1 TO 3 !                      Begins "I" loop.
30    FOR J = 4 TO 6 !                   Begins "J" loop.
40       PRINT I;J
50    NEXT J !                           Ends "J" loop.
60 NEXT I !                              Ends "I" loop.
70 END
```

External Documentation

Like internal program documents, there are two ways that external documentation can be presented to the user:

- By the program itself.
- In a separate manual.

Externally Self-Documenting Programs Ultimately, a program should require **no** external documents for its users. It should communicate what it does and how to use it through these two principles:

- If it operates "as the user would expect," then it doesn't need any documentation. The chapter called "Communicating with the Operator" presents some examples of this principle.
- If an interaction is so complex as to require description, then it should do so while it is running. For instance, it should prompt the user for data when required, giving as much description as necessary.

Since most programs do not have this level of "human interface," you will probably need to produce an external document for the human who is to use it.

External User Documents Although the topic of producing documentation for the program user is a large one, here are a few general suggestions:

- Give users a global view of all the things that they can do with the program.
- Describe how to complete each task. Present these discussions in logical progression.
- Provide relevant examples, and **be sure that they work.**
- If a task requires some level of expertise or knowledge, then state what is required. If possible, present relevant concept(s) at that point. If that is not possible, then give them a way to fulfill the requirement (such as by consulting another document or local expert).
- Summarize tasks at the end of tutorials.

3 Program Structure and Flow

Two of the most significant characteristics of a computer lie in its abilities to:

- Perform quick and accurate computations.
- Execute decisions within programs.

If the execution sequence could never be changed within a program, the computer could do little more than plug numbers into a formula. Technical BASIC has powerful computational features, but the heart of its usefulness is its ability to make decisions.

The computational power of Technical BASIC is exercised as it evaluates the expressions contained in the program lines. The chapters entitled "Numeric Computation" and "String Manipulation" present the various tools available for data manipulation.

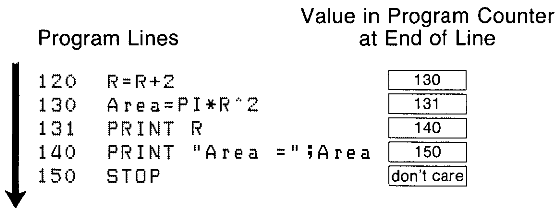
The decision-making power is used to determine the order in which lines will be executed. This chapter discusses the ways of controlling the "flow" of program execution.

Chapter Contents

Here are the general topics covered in this chapter.

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.



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

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
```

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 due to the program linearly flowing into a portion of the program that was not supposed to be executed.

Types of Program Flow

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. Here are the three general categories of program flow:

- Sequentially executed program segments (one after the other)
- Selection of program segments (conditional execution)
- Repetition of program segments (loops)

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.

Sequences of Program Segments

There are several types of sequences that the computer can use in executing program segments:

- Linear flow (no changes to normal sequence)
- Halting program execution
- Simple branching (modifying the normal sequence)

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 you 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:

- END
- STOP
- PAUSE

Each of these statements has a specific purpose, as explained in the following paragraphs.

STOP and END The "Program Development" chapter defined a main program as a list of program lines with an optional END or STOP statement on the last line. Marking the end of the main program is the primary purpose of the END statement; its secondary purpose is to stop 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 another purpose of the END or STOP statements. 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 not restored. (Subprograms and context switching are explained in the chapter "User-Defined Functions and Subprograms".)

As an example of the use of STOP and END, enter 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 you execute RUN, the computer prints `31` on the display. This first execution of the RUN command caused linear execution of lines 100 thru 130, with line 130 stopping that execution. If you execute RUN 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 you can execute a command like RUN 140. The computer prints `0` and then stops. This command caused linear execution of lines 140 thru 160, with line 160 stopping that execution. However, a RUN command also causes a pre-run initialization, which zeroed the value of the variable Radius[1].

1 See the subsequent section of this chapter called "A Closer Look at Program Execution" for an explanation of pre-run.

PAUSE A stopped program is not continuable. This leads up to the third statement for halting program flow. Replace the STOP statement on line 130 of the preceding program 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 execute RUN. The computer prints 31 on the display. Then execute CONT. The computer prints 78 on the display. 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 chapter "Program Debugging." Another use for PAUSE is to allow time for the computer user to read messages or follow instructions. Here is one example of using the PAUSE statement.

```
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 "type CONT and press RETURN."
170  PAUSE
180  !  Program execution resumes here after CONT.
```

Lines 100 thru 160 are instructions to the program user. Since a user will often just load a program and then execute RUN, the you cannot assume that the user's disc is in place at the start of the program. The instructions on the display 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. Note that the WAIT statement causes a delay in program execution until the specified number of milliseconds has elapsed. The PAUSE statement gives freedom to the user to take as little or as much time as necessary.

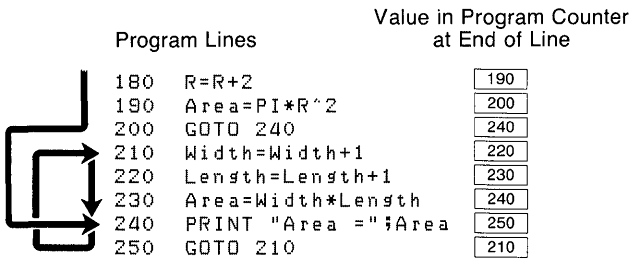
When CONT is executed, the program resumes with any necessary input of file names and assignments. Questions such as "Have you inserted the proper disc?" are unnecessary now. The user has already indicated compliance with the instructions by executing CONT.

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. The term *context* refers to the fact that each subprogram and user-defined function has its own independent set of variables and line labels. This is a complex subject that is the topic of an entire chapter ("User-Defined Functions and Subprograms"). This section discusses the use of GOSUB and GOTO for *local* branching.

Using GOTO First, you should be aware that it desirable to avoid the *excessive* or *unnecessary* use of the unconditional GOTO. The problem is not anything inherent in the GOTO statement. The problem lies in some programmers' tendencies to "patch together" pieces of a poorly planned 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". A program that contains sloppy and excessive use of GOTO has been appropriately named *spaghetti code*. Keep this very descriptive term in mind when you are deciding whether to "just throw something together" or to take a little more time to organize and plan a project.

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.



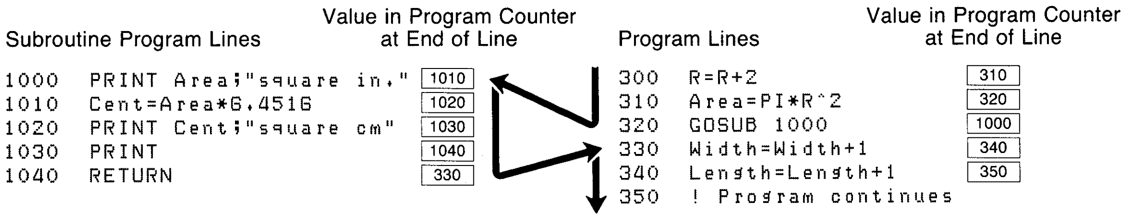
```
Program Lines                         Value in Program Counter
                                          at End of Line

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
```

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 lines 210 thru 250.

An infinite loop by itself is not 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 Technical BASIC. Calling a *subprogram* invokes a new context; subprograms can declare formal parameters and local variables. A *subroutine* is simply another segment of the current program context that is entered with a GOSUB and exited with a RETURN. There are no parameters passed and no local variables. If you are a newcomer to Technical BASIC, be careful to distinguish between these two terms. They have been used differently in some other programming languages.

The GOSUB statement is very useful in structuring and controlling program flow. GOSUB executes a branch to a subroutine, which performs a certain task or tasks. When those task are complete, control returns to the main body of the program. 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
1010   Cent=Area*6.4516          1020
1020   PRINT Cent;"square cm"    1030
1030   PRINT                     1040
1040   RETURN                    330
```

```
300   R=R+2                  310
310   Area=PI*R^2            320
320   GOSUB 1000             1000
330   Width=Width+1          340
340   Length=Length+1        350
350   ! Program continues
```

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 what program line should be returned to at the end of a service routine. Note that softkeys are keys on your keyboard which are defined by their corresponding label on the display. By using GOSUB and RETURN, the computer does the work for you. There are more details on this use of GOSUB later in this chapter.

3

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 follows:

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 well defined rules to dictate when a program task should be a subroutine and when it should be in the linear flow. The following suggestions may help you decide.

- **A subroutine should have some identifiable task**, such as opening a file, normalizing a variable, executing an end-of-page algorithm, decoding a keypress, parsing a string, and so forth. It is handy for a subroutine to "hide the details" of performing a task so that these details do not obscure the readability (and supportability) of the routine.
- **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[1] to select all the variations that are needed from one usage of the subroutine to the next, then a subprogram is probably a cleaner solution. Lines of code that just happen to be repeated in several places are not necessarily good candidates for a subroutine.
- 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.

1 System flags are system variables that you can use to keep track of information. For instance, you can use a flag to keep track of an operating mode by storing it as a numeric value: `IF FLAG(1) THEN End_page`. See the "Using System Flags" section of the "User-Defined Functions and Subprograms" for further information.

Selection of Program Segments

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.

An Example

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 "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.

Types of Conditional Execution

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*[1], note that there is no BOOLEAN data type in BASIC. 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 PRINT "Ph is > 7.7."
```

Notice the test (Ph>7.7) and the conditional statement (PRINT...) 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, then the expression evaluates to 0 (false) and the line is not executed. If the value contained in the variable Ph is greater than 7.7, then the expression evaluates as 1 (true) and the PRINT statement is executed. If you don't already understand logical and relational operators, refer to the chapter entitled "Numeric Computation" or the chapter entitled "String Computation".

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.

1 A Boolean expression can have one of two values: true (1), or false (0).

3

Prohibited Statements Certain statements are not allowed as the conditional statement in an IF...THEN statement. The disallowed statements are used for various purposes, but the "common demoninator" 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 an IF...THEN statement.

Keywords used in the declaration of variables:

OPTION BASE	COM
DIM	SHORT
INTEGER	REAL

Keywords that define context boundaries:

DEF FN	FNEND
SUB	SUBEND

Keywords that define program structures:

FOR
NEXT

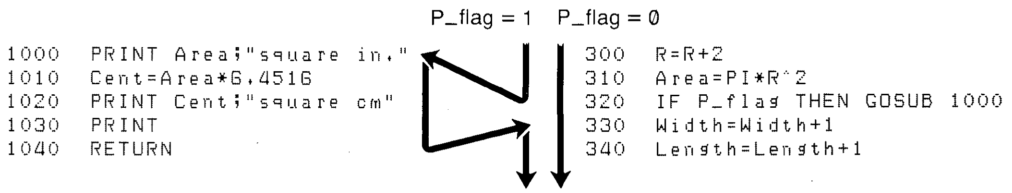
Keywords used to identify lines that are literals:

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.



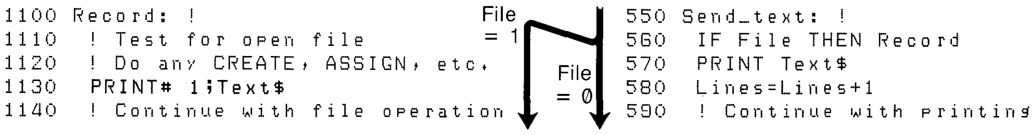
```
1000   PRINT Area;"square in."
1010   Cent=Area*6.4516
1020   PRINT Cent;"square cm"
1030   PRINT
1040   RETURN
```

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

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 implied GOTO , then 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   PRINT# 1;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
```

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 ! This is a multiple-line IF...THEN structure.
110 IF Ph<=7.7 THEN GOTO Skip
120       PRINT "Ph is > 7.7"
130       PRINT "Final Ph = ";Ph
140       PRINT "Conditional Test Ends"
150 Skip: ! Execution resumes here.
```

Any number of program lines can be placed between the line containing the IF...THEN statement (line 110 here) and the line number specified in the GOTO (line 150 here). In executing this example, the computer evaluates the expression Ph< =7.7 following the IF clause. If the result is true, then the program counter is set to 150 (i.e., the GOTO is executed), and execution resumes with that line. If the condition is false, the program counter is set to 120 (i.e., the GOTO is not executed), and the "conditional" statements (lines 120, 130, and 140) are executed. Line 150 is then where "normal" execution resumes.

If an other branching construct is used within a multiple-line IF...THEN structure, the entire structure should be contained in the conditional segment. This is called *nesting* constructs. The following example shows some properly nested constructs. Notice that the use of indenting improves the readability of the otherwise messy code.

```
100  PRINT "Enter an integer value between 1 and 5."
110  INPUT Value
120  IF Value<=1 THEN GOTO NotGrThan1
130             ! Begin outer IF.
140               PRINT "Value is greater than 1."
150               IF Value>=5 THEN NotLsThan5
160                    ! Begin nested IF.
170                    PRINT "Value is less than 5."
180 NotLsThan5:        ! End of nested IF.
190 !
200 NotGrThan1: ! End of outer IF.
210 END
```

Choosing One of Two Segments

This language has an IF...THEN...ELSE construct which makes the one-of-two choices 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 Isc=Select DIV 100 ELSE Isc=Select
```

Notice that this structure requires you to type the IF...THEN...ELSE structure on one contiguous line, which is not easy to read. Note that you may place multiple statements after the THEN and ELSE in this construct, as long as they are concatenated by the @ character. For example:

```
IF X > 5 THEN X=X+5 @ DISP X ELSE X=X^2 @ DISP X
```

This is one way of implementing multiple statements within the IF...THEN...ELSE construct. However, one contiguous line of 159 characters is not easy to read. A more readable way to implement the choice between one of two segments is as follows:

```
100 ! Choosing one of two segments.
110 IF X>5 THEN GOTO Seg1 ELSE GOTO Seg2
120 Seg1:
130    X=X+5
140    DISP X
150 GOTO CommonExit
160 Seg2:
170    X=X^2
180    DISP X
190 CommonExit: ! Both segments "continue" here.
200 END
```

Choosing One of Many Segments

This requires choosing from one of several possibilities, and is like executing a sequence of IF...THEN statements. This type of program flow can be generated with the ON statement and some additional processing. 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.

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

Using the ON...GOSUB statement, 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$="DVAVDIAIOM"
110  !
120  !
500  Pointer=POS(Match$,UPC$(Funct$))
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. The POS function returns the position of the first character of a substring within another string. 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, but 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 are not usually that lucky. To match arbitrary values, the following trick can be used. This example tests the three cases: <0, 1, and >1.

```
100 DISP "Enter an integer X."
110 INPUT X
120 Pointer=1*(X<0)+2*(X=1)+3*(X>1)
130 ON Pointer GOSUB Negative,One,Greater
140 Negative:
150    DISP "The value entered is negative."
160    GOTO Quit
170 One:
180    DISP "The value entered is a positive 1."
190    GOTO Quit
200 Greater:
210    DISP "The value entered is positive."
220    GOTO Quit
230 Quit: END
```

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 useful trick for 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. The code lines for this are left as an exercise for the reader.

Repetition

Humans usually prefer tasks with variety that avoid tedious repetition. A computer does not have this shortcoming. Although a computer is usually a miserable failure at creative thought, it is in full glory when called upon to accurately repeat the same boring task millions of times.

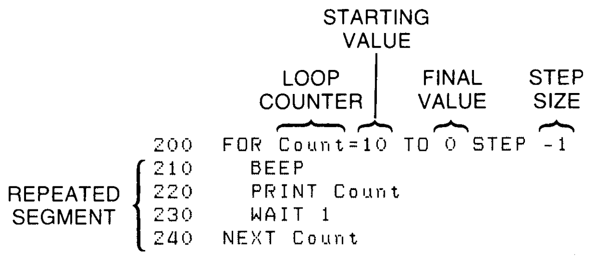
With Technical BASIC, you have only one structure available for creating repetition. However, the others can be built using the GOTO statement.

This section covers the repetitive capabilities common to all versions of BASIC and explains how you can implement them using Technical BASIC. These capabilities are:

- Repeating a program segment a predetermined number of times (using the FOR...NEXT construct)
- Repeating a program segment indefinitely (using the GOTO statement), waiting for a specified condition to occur
- Creating an iterative structure that allows multiple exit points at arbitrary locations (using the GOTO statement)

Fixed Number of Iterations

The general concept of repetitive program flow can be shown with the FOR...NEXT structure, available in all BASIC language systems. 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
```

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. Note also that the step size is a default of 1 when it is not included in the FOR...NEXT statement.

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 of 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    Counter=0
210   FOR X=10 TO 20
220      Counter=Counter+1
230      PRINT Counter
240   NEXT X
250   END
```

According to the formula, this loop should execute 11 times: INT((1 + 20 − 10)/1 = 11). The result on the display 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 small accumulated error that results from the successive addition of one-tenth. The error is less significant than the 15th digit, but discernible 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 shift the final value very slightly.

The following line does give the 11 iterations predicted by the formula.

```
210 FOR X=1 TO 2.0001 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.

Some 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.

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 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 DISP "Enter a positive number."
820 INPUT Number
830 !    INPUT "Enter a positive number",Number
840 IF Number<0 THEN Repeat ! Until Number>=0
1000 !
1010 DISP "Now this wasn't so bad."
1020 END
```

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    DISP "Enter a positive greater than 100."
1230    INPUT Note
1240    PRINT Note;
1250    Note=Note*2^(1/12)
1260  IF Note<880 THEN Repeat ! End at high A
2000 !
2010 DISP "It's getting better; not much."
2020 END
```

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 implemented with the IF...THEN and GOTO statements is more flexible than FOR...NEXT when working with exponential data in general.

The While...End While loop structure, which executes from one to N number of statements several times until the loop condition is met, is used for the same purpose as the Repeat...Until loop structure and is implemented using the GOTO statement[1].

The only difference between the two is the location of the test for exiting the loop. The Repeat structure has its test at the bottom (post-test). This means that the loop is always executed *at least once*, regardless of the value of the condition. The While structure has its test at the top (pre-text). Therefore, it is possible for the loop to be skipped entirely (if the conditions so dictate).

The Repeat...Until and While...End 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 structure 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.

1 Keep in mind that the Repeat...Until and While...End While *keywords* are **not** implemented in Technical BASIC.

```
100 I=1
110 While: ! I<LEN(Str$)
120   IF I>LEN(Str$) THEN End_While
130   IF Str$[I,I]<CHR$(32) THEN Remove ELSE I=I+1 @ GOTO While
140 Remove: LastChar=LEN(Str$)
150         Str$[I,LastChar-1]=Str$[I+1,LastChar] ! Remove ctrl. char.
160         Str$=Str$[1,LastChar-1] !  Remove trailing character.
170         GOTO While
180 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. One alternative is to place a conditional GOTO in the middle of the loop that directs program flow to a point beyond the bottom of the loop. In fact, with Technical BASIC, this is the way it is accomplished.

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 is 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 generalized Loop structure is well suited to this task, as shown in the following example.

```
2000 Loop1:
2010    Position=POS(A$,CHR$(14))
2020  IF NOT Position THEN GOTO End_Loop1 ! "Exit Loop1
2030    A$[Position]=CHR$(27)&"&dD"&A$[Position+1]
2040    GOTO Loop1
2050 End_Loop1:
2060 !
2070 Loop2:
2080    Position=POS(A$,CHR$(15))
2090  IF NOT Position THEN GOTO End_Loop2
2100    A$[Position]=CHR$(27)&"&d@"&A$[Position+1]
2110    GOTO Loop2
2120 End_Loop2:
```

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 structure 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     DISP "Enter a positive number."
220     INPUT Number
230   IF Number>=0 THEN End_Loop
240     BEEP
250     PRINT
260     PRINT "Negative numbers are not allowed."
270     PRINT "Repeat entry with a positive number."
280     GOTO Loop
290 End_Loop: END
```

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 exiting can occur at the top or bottom of the loop. This means that the Loop structure can serve the same purposes as the Repeat and While structures, if that suits your programming style.

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. Interrupts are conditions declared in a program that are constantly being monitored by the computer. When these particular conditions occur a branch is made from the normal program flow.

The process can be visualized as a special case of selection. Every time program flow leaves a line, the BASIC system executes an "event-checking" subroutine. The process of "event checking" is represented in the following lines.

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

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. These potential branching points are marked in the above BASIC program by the words "gosub event_check". These event checks are "if...then" statements that the BASIC system executes. If an event is enabled to initiate a branch (such as with ON KEY#) and the event occurs, then this "event-checking" routine initiates a branch to the *service routine* for the event (which you have designated in BASIC).

Notice that in the sample program above if the operating system finds a "true" event, a branch is taken at the end of the current line. If not, program execution resumes with the "normal" program flow.

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:

ON EOT	ON ERROR
ON KYBD	ON KEY#
ON INTR	ON TIMEOUT
ON TIMER#	

The ON EOT defines and enables end-of-line branching when the last byte of data is transferred by a TRANSFER statement. This topic is discussed in the *HP-UX Technical BASIC I/O Programming Guide.*

The ON ERROR statement is used to trap run-time errors by specifying a branch to an error-handling routine. This subject is is discussed in the chapter called "Error Handling".

The ON KYBD and ON KEY# events pertain to various parts of the keyboard and are used to enhance the "human interface" of programs. ON KYBD enables an event-initiated branch to be taken when the specified key(s) is(are) pressed during program execution. The term enable means to turn on the particular interrupt condition so that the computer can start monitoring that condition. The ON KEY# statement is used to define and label the softkeys on your keyboard, and enables an event-initiated branch for them.

The ON INTR and ON TIMEOUT events pertain to interfaces and I/O operations. ON INTR defines an end-of-line branch to be taken when an interface generates an interrupt. ON TIME-OUT enables end-of-line branching when an interface timeout occurs on the specified interface. These topics are discussed in the *HP-UX Technical BASIC I/O Programming Guide.*

ON TIMER# defines an end-of-line branch to be taken when the specified time interval has elapsed. Note that the OFF command of all keywords mentioned above cancels that command. Timers are discussed in the "Clock and Timers" chapter.

An Example of Using Softkeys

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.

```
100  ON KEY# 1,"Inc" GOSUB Plus
110  ON KEY# 5,"Dec" GOSUB Minus
115  ON KEY# 4,"Quit" GOSUB Quit
120  KEY LABEL
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 Quit: 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 lines 140 and 150 because of 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 BASIC system.

```
140  Spin: DISP X
     If Key# 1 pressed, then gosub Plus.
     If Key# 5 pressed, then gosub Minus.
150  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 busy" while it is waiting for an interrupt.

Now run the sample program you just entered. Notice that the the screen displays an inverse-video label area. 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. Any label which your program has not defined is blank. The label areas are defined in the ON KEY# statement by typing a comma after the key number and the key label name inside of quotes.

The starting value in the display line is zero, since numeric variables are initialized to zero at prerun. Each time you press **k1**[1], the displayed value of X is incremented. Each time you press **k5**[1], the displayed value of X is decremented. This simple demonstration should acquaint you with the basic action of the softkeys.

1 Key labels differ slightly for the various computers that run Technical BASIC. For information on key labels, refer to your particular HP-UX Technical BASIC system's *Getting Started* manual.

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 priorites are not of any real significance, such as the example program running now. However, priorities will sometimes cause unexpected flow problems. For more information on priorities, read the "Branch Precedence Table" found in the *HP-UX Technical BASIC Language Reference.*

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 keypress 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.

A key is *deactivated,* if it no longer has any influence on program flow. You can press a deactivated key all day long and nothing will happen.

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

- OFF EOT deactivates end-of-line branching for termination of a TRANSFER operation on the specified interface.
- OFF ERROR deactivates interrupts resulting from run-time errors. If these events occur while deactivated, the program pauses and an error message is displayed.
- OFF INTR deactivates end-of-line branching for interface interrupts previously established by ON INTR.

- OFF KYBD deactivates end-of-line branching previously enabled by an ON KYBD statement.
- OFF KEY# deactivates interrupts from the softkeys. If a softkey is pressed while deactivated, it does nothing.
- OFF TIMEOUT deactivates interrupts from interface timeouts. There is no such thing as a "timeout" when ON TIMEOUT is deactivated.
- OFF TIMER# deactivates end-of-line branching for the specified timer.

The following example shows one use of OFF KEY# to disable the softkeys.

```
100 Begin:
110   ON KEY# 1,"StpSz" GOSUB Step_size
120   ON KEY# 4,"Start" GOTO Process
130   ON KEY# 5,"Quit " GOTO Quit
140   KEY LABEL
150   !
160   Inc=1
170   DISP "Step Size = 1"
180   !
190 Spin: GOTO Spin !        Wait for keypress
200   !
210 Step_size:
220   Inc=Inc+1 !            Change increment
230   DISP "Step Size = ";Inc
240   RETURN
250   !
260 Process:
270 ! OFF KEY# !             Deactivate first choices
280 ! ON KEY# 8," ABORT" GOTO Leave
290   KEY LABEL
300   Number=0
310   FOR I=1 TO 10
320     Number=Number+Inc
330     PRINT Number
340     WAIT 600
350   NEXT I
360 Leave:
370   OFF KEY# 8 !           Deactivate ABORT
380   PRINT !                End line
390   GOTO Begin !           Start over
400 !
410 Quit: END
```

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 270 and 280 commented out, the *softkey menu*, or label area, never changes.

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 270. This disables all the softkeys when the printing process starts. Notice that the softkey menu goes away when no sofkeys are active. This is a very handy feature while you are experimenting with interupts. It provides immediated feedback to indicate when interrupts are active and when they are not.

Finally, remove the exclamation point from line 280. 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.

The OFF KEY# statement can include a key number to deactivate a selected key. This was done in line 370.

Chaining Programs

You may have had in the past a program which was quite large, and you wanted to reduce the amount of memory required to run the program. The Technical BASIC system allows a running program to load and run another program. This section explains this type of operation.

3

General Features

Chaining allows you to break up a program into smaller segments, loading and running only one segment at a time.

If you need to pass information from the program currently running to the program being chained, you can use the COM statement to place the shared variables in a "common" storage area. All that is necessary is to insure that the COM declarations in both programs match, and use the CHAIN command to call the other program.

A Simple Example

The following three short programs illustrate chaining. (If you are going to type these programs into your computer for this example, note that you will need to STORE them, not SAVE them. CHAIN only works with files created by the STORE command.)

```
10  REM ****************** Program#1 ****************************
20  PRINT "Program#1"
30  CHAIN "Program#2"
40  END

10  REM ****************** Program#2 ****************************
20  PRINT "Program#2"
30  CHAIN "Program#3"
40  END

10  REM ****************** Program#3 ****************************
20  PRINT "Program#3"
30  END
```

When Program#1 is run, the following output is printed:

```
Program#1
Program#2
Program#3
```

Program-to-Program Communications

All variables not placed in "common storage" (i.e., declared by a COM statement) are scratched when the chained program is loaded. So if you want chained programs to communicate with one another, then you will need to declare variables with the COM statement.

The preceding three programs have been modified to place four variables in COM, thereby providing a means for the programs to communicate. Note that the variables can be accessed with different names as they are passed between programs.

```
10  REM ***************** Program#1C ****************************
20  COM A,B$[1],C,D
30  A=1 @ B$="x" @ C=3 @ D=4
40  PRINT "Program#1C";A;B$;C;D
50  CHAIN "Program#2C"
60  END
```

```
10  REM ***************** Program#2C ****************************
20  COM T,Y$[1]
30  COM C,D
40  PRINT "Program#2C";T;Y$;C;D
50  CHAIN "Program#3C"
60  END
```

```
10  REM ***************** Program#3C ****************************
20  COM Q,R$[1]
30  COM W,X
40  PRINT "Program#3C";Q;R$;W;X
50  END
```

When Program#1C is run, the following output is printed:

```
Program#1C 1 x 3  4
Program#2C 1 x 3  4
Program#3C 1 x 3  4
```

This is a simplistic example; however, it does show the general tasks involved in chaining programs. For further details about numeric and string variables, read the "Numeric Computations" and "String Manipulation" chapters, respectively. For further information about COM, read the "User-Defined Functions and Subprograms" chapter.

A Closer Look at Program Execution

3

The normal running of a program is started by the RUN command. Before being able to run a program, however, the computer must make a "pre-run," during which it performs such tasks as allocating memory for variables and verifying that the line numbers specified in branch instructions (GOTO and GOSUB) actually exist. The computer then begins normal program execution, starting with the lowest numbered line in the main program.

This section describes some of the things that are happening during prerun and while the program is being executed. You may skip this section with no loss of continuity if you are not interested in this level of detail.

Prerun (RUN and INIT)

Prerun is executed automatically by the RUN command. It is also executed by the INIT command, which allows you to perform a prerun without starting program execution – a handy operation to have for use with the SINGLESTEP command (see the "Program Debugging" chapter for details).

There are three primary reasons for the prerun.

- To reserve sufficient memory for all the variables in the program. This includes all variables in COM, DIM, INTEGER, SHORT, and REAL statements, and all implicitly declared variables. (The chapter entitled "Numeric Computation" explains the declaration of numeric variables, and the chapter "String Manipulation" covers the dimensioning of string variables.)
- To detect errors that involve interaction between lines. The computer checks for syntax errors before it stores a program line. However, there are some errors that can't be detected by looking at a single line. For example, a program line that uses properly placed subscripts can appear to be correct when it is stored. However, if that line references two dimensions in an array that had previously been declared to have only one dimension, it is in error. To detect an error of that kind, the computer needs to "search" the entire program to see all the dimension statements as well as the variables used in each line. Another example of this kind of error is a GOTO or GOSUB that specifies a line that does not exist.

- To locate all the user-defined function boundaries. These are defined by the DEF FN statement and the FNEND statement with multiple-line, user-defined functions. (See the "User-Defined Functions and Subprograms" chapter for a complete description of user-defined functions.)

Note that these types of "prerun errors" are **not** caught by "ON ERROR" (discussed in the "Error Handling" chapter).

Normal Program Execution

The term *execution* is used to describe the process used by the computer while it is completing the tasks described in its program. The process of program execution is summarized below.

1. Determine which program line is to be acted upon next.
2. Identify the statement that follows the line number and label (if any) on that line.
3. If the statement has a run-time action, then perform that action.
4. Repeat steps 1 thru 3 until an END, STOP, PAUSE or an error occurs.

The continuing process of determining which line is to be executed next is discussed in detail in preceding sections of this chapter. The RUN command determines which line is acted on first. Executing `RUN` with no parameters, causes the execution process to begin at the first (lowest-numbered) line of the main program. Execution can be started anywhere in the main program by using the RUN command with a line identifier. For example:

```
RUN 220
```

This command causes execution to begin at line 220, if there is such a line. If there is no line 220 in the main program, execution begins with the line whose number is closest to and greater than 220.

Note that the prerun phase is always the same, whether the actual execution begins at the program start or somewhere in the middle. Also, if a starting line is specified, that line must be in the main program. An error 3 results when RETURN is executed if you attempt to start a program in a user-defined function or subprogram. Even if the starting point is correctly specified, be alert to the effects of starting a program in the middle. Skipping over a section of the program may result in null values for some of the variables.

Non-Executed Statements

In the preceding summary of normal execution, step 3 mentioned that only statements with run-time actions are executed. The term *run-time* refers to the state that exists after the prerun, when the computer is actually performing the sequence of actions described by the program. Some statements are not executed in the course of normal program flow, but are merely "looked at" and then bypassed.

The following is a list of some statements that do not cause an action as a result of run-time execution.

- Comments and REM statements: these never cause an action. (See the "Program Development" chapter for more complete details.)
- Variable declarations: COM, DIM, INTEGER, SHORT, and REAL. These are executed during prerun but skipped at run time. The OPTION BASE statement is also part of the declaration process. (See the "Numeric Computation"s and "String Manipulations" chapters for further descriptions of these statements.)
- DEF FN and FNEND statements. These are used during prerun to establish the program structure and are skipped over at run time. (See the "User-Define Functions and Subprograms" chapter for a complete description.)
- DATA statements: these are accessed by the READ statement, but are not executed. (See the "Data Storage and Retrieval" chapter for further information.)

4 Numeric Computation

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. For instance, adding two numbers and calculating a sine or a logarithm are all numeric operations. Making numeric computations from the keyboard and within a program are covered in this chapter.

Even though numeric computation includes converting a number to a string, and vice versa, these tasks are not described in this chapter; they are covered in the "String Manipulations" chapter.

Chapter Contents

Here are the major topics covered in this chapter.

- Assigning values to numeric variables
- Numeric data types
- Evaluating scalar expressions
- Making comparisons work
- Range limits
- Rounding
- Binary operations
- Number-base conversions
- Trigonometric functions
- Random numbers
- Miscellaneous numeric functions
- Array operations

Assigning Values to Variables

One of the most fundamental numeric operations is the assignment operation, achieved with the LET statement. The LET statement originally required the keyword LET for BASIC interpreters, but Technical BASIC makes it optional. Thus, the following *program lines* are equivalent:

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

However, when executing these statements *from the keyboard,* there is a difference:

- A = A + 1 is evaluated as a *boolean* expression.
- LET A = A + 1 is an *assignment* to the variable A.

Unless you have declared otherwise, the *data type* of numeric variables in this example is REAL. This is the default data type of numeric variables. The next section discusses other numeric data types and shows how to declare the type of a variable.

Numeric Variable Names The rules for naming simple numeric variables are as follows: the name can be up to 32 characters in length, and it may contain alphabetic (uppercase and lowercase) characters, decimal digits, and the underscore (_) character. The only restriction is that the first character must be a letter.

Here are some examples:

```
A
AVeryDescriptiveVariableName
Const22
NumericResult
a15
z_coordinate
```

Numeric Data Types

There are three pre-defined numeric data types in Technical BASIC:

- INTEGER
- SHORT
- REAL

Any numeric variable that is not explicitly declared an INTEGER or SHORT is implicitly declared to be of type REAL.

REAL Numbers

The range of REAL numeric variables is the largest range of numeric values. The largest value of a REAL variable is returned by the numeric INF (infinity) function, and the smallest positive value is returned by the EPS (epsilon) function. For a more complete description of the range on your system, see the *Implementation Specifics* appendix for your particular BASIC system.

SHORT Real Numbers

The range of SHORT numeric variables is less than that of REAL numbers. For an exact description of the range of REALs on your system, see your *Implementation Specifics* appendix.

INTEGERs

The range of INTEGER numeric variables is less than that of REAL and SHORT numbers. Also, INTEGERS are *whole* numbers, and cannot contain any fractional part.

For an exact description of the range of INTEGERS on your system, see your *Implementation Specifics* appendix.

Declaring a Variable's Data Type

The DIM, REAL, SHORT, and INTEGER statements are provided for explicitly declaring numeric variables:

```
DIM SimpleReal,RealArray(4,5)
REAL XCoord,YCoord,Voltage(4,13)
SHORT LogBase10,Hours(52,7)
INTEGER I,J,Days(5),Weeks(5,17)
```

Each of the above statements declares both simple and array variables.

- A simple variable can, at any given time, contain only a single value.
- An array variable can contain multiple values, each of which is accessed by subscripts.

With Technical BASIC, you can *only* define the upper bound of array subscripts; the current OPTION BASE is *always* defined to be the lower bound. Details on declarations of arrays and how to use them are provided in the subsequent "Arrays" section of this chapter.

Implicit Type Declarations When a variable is used in a program without its type being previously declared (such as with SHORT or INTEGER), it is implicitly declared to be of type REAL. Even though you can use this feature to implicitly declare a REAL variable's type, it is better programming practice to *explicitly* declare all variables. As shown in the preceding example, the DIM statement may also be used to declare REAL variables.

Evaluating Scalar Expressions

This section describes some additional details of how the computer evaluates scalar arithmetic expressions (as opposed to array expressions, which is discussed in the subsequent "Arrays" section).

Arithmetic Hierarchy

If you look at the expression 2+4/2+6, it can be evaluated in 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. These rules were made 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:

- Constants – represent numbers or strings with fixed value.
- Variables – represent the value stored in the variable.
- Operators – modify or perform operations on other elements in the expression.
- Intrinsic numeric functions – represent numeric values.
- User-defined numeric functions – represent numeric values.
- Parentheses – used to modify the default arithmetic hierarchy.

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

Math Hierarchy

Precedence	Operator
Highest	Parentheses: `(  )` used to force the order of evaluation
	Functions: both user-defined and machine-resident
	Exponentiation: `^`
	The logical "Not" monadic operator: `NOT`
	Multiplication and division: `*` `/` `MOD` `DIV`
	Addition and subtraction, and monadic operators: `+` `-`
	Relational operators: `<` `<=` `=` `>=` `>` `<>`
	Logical "And" operator: `AND`
Lowest	Logical "Or" operators: `OR` `EXOR`

When an expression is evaluated, it is read from left to right; operations are performed as encountered, unless one of the following is encountered:

- A higher precedence operation is encountered immediately to the right of the operation being evaluated.
- The hierarchy is modified by parentheses.

If the computer cannot deal immediately with the operation, it is stacked and the expression evaluator continues to read the expression until it encounters an operation it can perform. It is easier to understand if you see how an expression is actually handled. 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 background information. We will assume that DEG has been executed, that X=90, and that FNNegl returns −1. Evaluation proceeds as follows:

$$5+3*(\underbrace{4+2})/\mathrm{SIN}(X)+X*(1>X)+\mathrm{FNNeg1}*(X<5\ \mathrm{AND}\ X>0)$$

$$5+\underbrace{3*6}/\mathrm{SIN}(X)+X*(1>X)+\mathrm{FNNeg1}*(X<5\ \mathrm{AND}\ X>0)$$

$$5+18/\underbrace{\mathrm{SIN}(X)}+X*(1>X)+\mathrm{FNNeg1}*(X<5\ \mathrm{AND}\ X>0)$$

$$5+\underbrace{18/1}+X*(1>X)+\mathrm{FNNeg1}*(X<5\ \mathrm{AND}\ X>0)$$

$$\underbrace{5+18}+X*(1>X)+\mathrm{FNNeg1}*(X<5\ \mathrm{AND}\ X>0)$$

$$23+X*\underbrace{(1>X)}+\mathrm{FNNeg1}*(X<5\ \mathrm{AND}\ X>0)$$

$$23+\underbrace{X*0}+\mathrm{FNNeg1}*(X<5\ \mathrm{AND}\ X>0)$$

$$\underbrace{23+0}+\mathrm{FNNeg1}*(X<5\ \mathrm{AND}\ X>0)$$

$$23+\underbrace{\mathrm{FNNeg1}}*(X<5\ \mathrm{AND}\ X>0)$$

$$23+-1*(X<5\ \mathrm{AND}\ \underbrace{X>0})$$

$$23+-1*(0\ \mathrm{AND}\ \underbrace{X>0})$$

$$23+-1*\underbrace{(0\ \mathrm{AND}\ 1)}$$

$$23+-\underbrace{1*0}$$

$$\underbrace{23+0}$$

$$23$$

Operators

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

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

  Examples of usage:

  ```
     -5
  NOT True
  ```

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

Examples of usage:

```
5+3
```
Var1 MOD Var2 Var1 = 44

While the use of most operators is obvious from the descriptions in the *Technical BASIC Language Reference*, some of the operators have uses and side effects that are not always apparent. Examples of such subtleties for DIV and MOD are shown in the next section.

DIV and MOD Two additional arithmetic operators are DIV (integer division) and MOD (modulo). These operators can be used exactly like the arithmetic operators previously discussed.

4

The DIV operator uses this formula:

```
A DIV B = IP(A/B)
```

where IP returns the integer portion of the quotient of A/B.

The MOD operator returns the remainder resulting from a normal division. Given two numbers, A and B, A MOD B is defined by the equation:

```
A MOD B = A - B * INT(A/B)
```

where INT(A/B) is the greatest integer less than or equal to the quotient of A/B. It turns out that:

`0<=(A MOD B)<B` if `B>0`

and

`B<(A MOD B)<=0` if `B<0.`

By definition, A MOD 0 is A.

Try the following arithmetic operations:

Expression	Result
`16 DIV 5`	`3`
`19 DIV 5`	`3`
`5 DIV 16`	`0`
`5 DIV 6`	`0`
`16 MOD 5`	`1`
`17 MOD 5`	`2`
`-8 MOD 3`	`1`
`-(8 MOD 3)`	`-2`

The expression −8 MOD 3 is not evaluated the same way as −(8 MOD 3), because the monadic −operator has a higher priority than the MOD operator.

Expressions, Calls, and Functions

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 functions and subprograms read the chapter "User-Defined Functions and Subprograms".

Strings in Numeric Expressions

String expressions can be directly included in numeric expressions if they are separated by comparison operators. For instance:

```
A$="ABC"
```

The comparison operators always yield "boolean" results, which are numeric values in BASIC. This numeric expression is 1 if the string variable names A$ is equal to the string ABC, and 0 otherwise.

Step Functions The comparison operators are obviously useful for conditional branching (IF...THEN statements), but are also valuable for creating numeric expressions representing step functions. For example, you can easily represent this function with a numeric expression:

- IF Select<0 THEN Result = 0
- IF 0< = Select<1 THEN Result = $(A^2 + B^2)$^½.
- IF Select> = 1 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)+(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" (comparison) 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 Result. This technique can be used to represent other functions, as long as the program statement does not exceed the maximum allowable line length.

Making Comparisons Work

If you are comparing INTEGER numbers, no special precautions are necessary. However, if you are comparing REAL or SHORT 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 comparison operators in IF..THEN statments to check for equality in any situation resembling the following:

```
120  DEG
130  A=25.3765477
140  IF SIN(A)^2+COS(A)^2=1 THEN PRINT "OK" ELSE PRINT "Not OK"
```

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.

A good example of equality error occurs when multiplying or dividing data values. A 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, there are three techniques that can be used to eliminate equality errors:

- Use the value of the *absolute difference* between the two values, and test for the difference less than a specified limit.

  ```
  IF ABS(A-B)<.001 THEN DISP "Equal"
  ELSE DISP "Not Equal"
  ```

- Use the absolute value of the *relative difference* between two values, and test for the difference less than a specified limit:

  ```
  IF ABS((A-B)/B)<.001 THEN DISP "Equal"
  ELSE DISP "Not equal"
  ```

- Eliminate unwanted resolution *before* comparing results. For instance, you could use a specified number of significant digits in the comparison.

Range Limits

It is sometimes necessary to limit the range of values that are assigned to a variable. You can do that with IF...THEN statements, as shown in these statements:

```
200 IF X>MaxX THEN X = MaxX
210 IF X<MinX THEN X = MinX
```

However, it is more convenient to use the MAX and MIN functions.

```
200 X = MAX(X,MinX)
210 X = MIN(X,MaxX)
```

Note that MAX is used to establish the lower bound, and MIN is used to establish the upper bound. If you think about it a minute, it makes sense.

Here is a faster version of the above computation:

```
190  X = MIN(MAX(X,MinX),MaxX)
```

Rounding

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 an output operation. For instance:

```
DISP USING "DD.DDD";Number
```

The value in the variable Number is displayed using (up to) two decimal digits preceding the decimal point radix, and three digits following the decimal point. For further details, see the "Formatting Information" section of the "Communicating with the Operator" chapter. This feature works in statements such as DISP, PRINT, LABEL, and OUTPUT.

Sometimes it is necessary to round a number in a calculation in order to eliminate unwanted resolution. There are two basic types of rounding:

- Rounding to a number of significant decimal *digits*
- Rounding to a number of significant decimal *places* (limiting fractional information)

Both types of rounding have their own application in programming.

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.

In rounding to a number of decimal places, the idea is to eliminate decimal representation beyond a specific power of ten. A simple approach to it is to shift the desired decimal information to the left of the radix, round up (add 0.5 to the resulting quantity), use INT to get rid of the undesired decimal information, then shift the number (to the right) back to its original position.

```
180  X=123.456
190  Places=2 !  Round to two digits after decimal point.
200  ScaleFactor=10^Places
210  XRounded=INT(X*ScaleFactor+0.5)/ScaleFactor
220  DISP XRounded
```

Here are the program's results:

```
   123.46
```

ScaleFactor and Places should both be INTEGERs. The example shows rounding to 2 decimal places (to the right of the decimal point). Places should be set to a negative number to round to a number of digits to the left of the decimal point.

Binary Operations

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 the computer performs use the binary number representation. You usually don't see this, because the computer 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 it displaying or printing them.

There are some operations available with Technical BASIC that deal with binary numbers. For example, the BINIOR function performs a bit-by-bit "inclusive or" operation on the two arguments, and returns the result:

```
BINIOR(2,5)
 7
```

When any of these operations are used, the arguments are first converted to INTEGER (if they are not already of this type) and then the specified operation is performed. Therefore it is best to restrict bit-oriented binary operations to declared INTEGERs. However, if it is necessary to operate on a REAL or SHORT, then you should make sure that the argument is not beyond the range of INTEGERs (to avoid an error).

Resident Binary Functions

In the following descriptions, the variable(s) shown in parentheses (such as `Arg1`, `NthBit`, and `Shift`) signify that the function requires numeric argument(s), which can be *any* numeric expression.

Function	Description
`BINAND(Arg1,Arg2)`	Returns the bit-by-bit logical "AND" of the two arguments.
`BINCMP(Arg)`	Returns the bit-by-bit "complement" of the argument.
`BINEOR(Arg1,Arg2)`	Returns the bit-by-bit *exclusive* "OR" of the two arguments.
`BINIOR(Arg1,Arg2)`	Returns the bit-by-bit *inclusive* OR of the two arguments.
`BIT(Arg,NthBit)`	Returns the state of bit `NthBit` of `Arg`.
`ROTATE$(String$,Shift)`	Returns a string obtained by shifting the characters in the string argument `String$` the number of positions specified by `Shift`, with wraparound. (Even though this is a string function, you may also find it useful for binary operations. See the "String Manipulations" chapter for details.)

Number-Base Conversions

The computer treats all numeric values as decimal (base 10) quantities. However, it is often necessary to handle numbers represented in these number bases:

- Binary (base 2)
- Octal (base 8)
- Hexadecimal (base 16)

This section describes the functions that allow you to convert between these number bases.

Converting from Decimal

Technical BASIC provides resident functions for converting from decimal to binary, octal, and hexadecimal number bases. These functions are as follows:

`DTB$`	Converts a decimal number to a binary string
`DTO$`	Converts a decimal number to an octal string
`DTH$`	Converts a decimal number to a hexadecimal string

The DTB$ (Decimal-To-Binary) string function returns a string containing the base-2 representation of the decimal argument. For example, to find the binary representation of 15, use this function:

```
DTB$(15)
```

The following "32-bit" *string* value is returned:

```
00000000000000000000000000001111
```

The DTO$ (Decimal-To-Octal) string function returns a string containing the octal representation of the decimal argument. The following function call returns the octal string representation of decimal 15:

```
DTO$(15)
```

This *string* value is returned:

```
00000000017
```

The DTH$ (Decimal-To-Hexadecimal) string function returns a string containing the hexadecimal representation of the decimal argument. To find the hexadecimal value for 15, use the following function call:

```
DTH$(15)
```

The following *string* value is returned:

```
0000000F
```

Converting to Decimal

Technical BASIC also provides functions to convert numbers from binary, octal, and hexadecimal number bases to a decimal numeric representation. These functions are as follows:

`BTD`	Converts a binary string to a decimal number
`OTD`	Converts an octal string to a decimal number
`HTD`	Converts a hexadecimal string to a decimal number

The BTD(Binary-To-Decimal) numeric function returns the decimal equivalent of the specified binary number (which is represented by a string expression). For instance, this function converts an "8-bit" string-binary number to a decimal numeric value:

```
BTD("11111111")
```

Here is the *numeric* value it returns:

```
255
```

The OTD(Octal-To-Decimal) numeric function returns the decimal equivalent of the specified octal number (which is represented by a string expression). For instance, this function converts a string-octal number to a decimal numeric value:

```
OTD("377")
```

Here is the *numeric* value it returns:

```
255
```

The HTD(Hexadecimal-To-Decimal) numeric function returns the decimal equivalent of the specified hexadecimal number (which is represented by a string expression). For example, this function converts an "16-bit" string-hexadecimal number to a decimal numeric value:

```
HTD("ff")
```

Here is the *numeric* value it returns:

```
255
```

4 Trigonometric Functions

Technical BASIC provides several functions for dealing with angles and angular measure: SIN, COS, TAN; CSC, SEC, COT; ASN, ACS, ATN, ATN2; DTR and RTD. Each function has a different purpose, as described subsequently; however, all deal with angles. The interpretation of the argument, and consequent value returned, is dependent on the angular unit of measure currently being used.

- The *default* unit for all angular measure is radians. You can use the RAD statement to set this mode. (There are 2 π radians in a circle.)
- Degrees can be selected with the DEG statement. (There are 360 degrees in a circle.)
- Grads can be selected with the GRAD statement. (There are 400 grads in a circle.)

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. Subprograms inherit the angular mode from the context that calls it. And if the subprogram changes the mode, then the mode used in the calling context is **not** restored.

Resident Trigonometric Functions

In the following descriptions, the variable(s) shown in parentheses (such as `Angle`, and `X`) signify that the function requires numeric argument(s), which can be *any* numeric expression.

Function	Description
`ACS(Cosine)`	Returns the arccosine of an expression ($-1 \leq \text{Cosine} \leq 1$) as an angle in first or second quadrant. The resultant angle returned is dependent upon the current DEG/RAD/GRAD mode.
`ASN(Sine)`	Returns the arcsine of an expression ($-1 \leq \text{Sine} \leq 1$) as an angle in first or fourth quadrant. The resultant angle returned is dependent upon the current DEG/RAD/GRAD mode.
`ATN(Tangent)`	Returns the arctangent of an expression as an angle in first or fourth quadrant. The resultant angle returned is dependent upon the current DEG/RAD/GRAD mode.
`ATN2(Y,X)`	Returns the arctangent of Y divided by X (Y/X) in the proper quadrant. (X,Y) is the rectangular coordinate position of a point. The resultant angle returned is dependent upon the current DEG/RAD/GRAD mode.
`COS(Angle)`	Returns the cosine of the angle specified by the expression. The interpretation of the specified angle is dependent upon the current DEG/RAD/GRAD mode.
`COT(Angle)`	Returns the cotangent of the angle specified by the expression. The interpretation of the specified angle is dependent upon the current DEG/RAD/GRAD mode.
`CSC(Angle)`	Returns the cosecant of the angle specified by the expression. The interpretation of the specified angle is dependent upon the current DEG/RAD/GRAD mode.
`DTR(DegreeAngle)`	Converts the specified angle expression from degrees to radians. The result returned is *independent* of the current DEG/RAD/GRAD mode.
`RTD(RadianAngle)`	Converts an angle expression from radians to degrees. (The value this function returns is *independent* of the current DEG/RAD/GRAD mode.)
`SEC(Angle)`	Returns the secant of the angle represented by an expression. The interpretation of the specified angle is dependent upon the current DEG/RAD/GRAD mode.
`SIN(Angle)`	Returns the sine of the angle specified by the numeric expression. The interpretation of the specified angle is dependent upon the current DEG/RAD/GRAD mode.
`TAN(Angle)`	Returns the tangent of the angle represented by an expression. The interpretation of the specified angle is dependent upon the current DEG/RAD/GRAD mode.

Random Numbers

The RND numeric function returns a pseudo-random random number greater than or equal to 0 and less than 1.

```
RND
 0.0459608752708518
```

Scaling

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

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

The above statement will return an integer between Offset and Offset + Range.

A New Seed

The random number generator is seeded with the a default value at system reset, power-on, SCRATCH, and pre-run. You can change the seed with the RANDOMIZE statement, which will give a new pattern of numbers.

```
RANDOMIZE NewSeed
```

Miscellaneous Numeric Functions

Technical BASIC provides a generous complement of numeric functions. For instance, the ABS, MIN, and MAX functions have been used in preceding examples. This section lists some of the general numeric functions which are not described elsewhere in this chapter.

Resident General Numeric Functions

In the following descriptions, the variable(s) shown in parentheses (such as `Number` and `Arg`) signify that the function requires numeric argument(s), which can be *any* numeric expression.

Function	Description
`ABS(Number)`	Returns the absolute value of the expression `Number`.
`CEIL(Number)`	Returns the smallest integer greater than or equal to the expression `Number`.
`EPS`	Returns the machine's "epsilon" – the smallest positive number greater than zero that the system can handle. Note that this is system-specific; refer to the *Specifics* appendix for you particular system.
`EXP(PowerOfE)`	Raise the natural *e* to the power specified by the expression `PowerOfE`. ($e \approx 2.718\ 281\ 828\ 459\ 05$).
`FLOOR(Number)`	Returns the greatest integer that is less than or equal to the specified expression `Number`. (Same as INT function.)
`FP(RealNumber)`	Returns the "fractional" part of the specified expression `RealNumber`.
`INF`	Returns the machine's "infinity" – the largest positive number that the system can handle. This number is system-specific; refer to the *Specifics* appendix for you particular system.
`INT(Number)`	Returns the greatest integer that is less than or equal to the specified expression `Number`. The result is of the same type (INTEGER or REAL) as the original number. It differs from IP with negative numbers. For example:

`INT(-45.66)`	returns the value -46.
`IP(-45.66)`	returns the value -45.

Function	Description
`IP(Number)`	Returns the integer part of the expression `Number`. (Similar to INT above.)
`LGT(Number)`	Returns the base 10 logarithm of the specified decimal expression `Number`.
`LOG(Number)`	Returns the natural logarithm (base *e*) of the specified decimal expression `Number`.
`MAX(Arg1,Arg2)`	Compares `Arg1` and `Arg2`, and returns the larger of the two values.
`MIN(Arg1,Arg2)`	Compares `Arg1` and `Arg2`, and returns the smaller of the two values.
`NUM(String$)`	Returns the decimal code of the *first* character in the specified `String$` expression.
`PI`	A constant function which returns a 15-digit approximation of π; 3.141 592 653 589 79.
`POS(Source$,Target$)`	Returns the position (index) of the `Target$` string expression in the Source$ string expression. (See the "String Manipulations" chapter for examples.)
`RMD(Dividend,Divisor)`	Divides `Dividend` by `Divisor` (Dividend/Divisor) and returns the *remainder* of the division (not the quotient, as in DIV).
`SGN(Number)`	Returns the arithmetic sign of the expression `Number`: 1 if positive, 0 if 0, −1 if negative.
`SQR(PositiveN)`	Returns the positive square root of the non-negative expression `PositiveN`.
`VAL(String$)`	Returns the decimal number represented by the string expression.
`VAL$(Number)`	Returns the string representation of the specified `Number`.

Arrays

This section describes the broad topic of arrays. Here are the topics discussed in the remainder of this chapter:

- Concepts – what is an array?
- Creating an array variable – dimensioning.
- Assigning value to individual elements.
- Displaying arrays.
- Redimensioning arrays.
- Assigning values to all elements of an array.
- Constant and zero matrices.
- Identity matrices.
- Copying subarrays.
- Scalar arithmetic array operations.
- Summing rows and columns.
- Array transpose.
- Matrix multiplication.
- Vector cross products.
- Matrix inversion.
- Solving systems of linear equations.

Array Concepts

An array variable (or simply, an array) is a group of data items of one type, collectively referred to by one variable name. Subscripts enclosed in parentheses after the array name reference individual items in the collection.

Technical BASIC allows one- and two-dimensional arrays. A one-dimensional array (also called a vector) can be thought of as a list of items consisting of several rows but only one column; items are referenced by one integer subscript. A two-dimensional array is like a table of items. The table has multiple rows and columns, and elements in the table are accessed by two integer subscripts separated by commas.

The number of items, or elements, in an array is determined by the lower and upper bounds of its subscripts. The lower bound of an array subscript is the lowest value that the subscript can be assigned; the upper bound of an array subscript is the highest value subscript that the subscript can be assigned. For example, the following group of ten numbers can be organized several different ways: as one ten-item list; as two five-item lists; or as five two-item lists.

```
1        2
3        4
5        6
7        8
9        10
```

The following array is organized as one ten-item array.

Numbers(0) = 1	Numbers(1) = 2
Numbers(2) = 3	Numbers(3) = 4
Numbers(4) = 5	Numbers(5) = 6
Numbers(6) = 7	Numbers(7) = 8
Numbers(8) = 9	Numbers(9) = 10

The lower bound of the array subscript is 0, and the upper bound is 9. Non-integer subscripts are rounded to the nearest integer. Negative subscripts are not allowed.

The following assignments treat the data as two five-item lists (i.e., two one-dimensional arrays of five elements each).

OddNumbers(0) = 1	EvenNumbers(0) = 2
OddNumbers(1) = 3	EvenNumbers(1) = 4
OddNumbers(2) = 5	EvenNumbers(2) = 6
OddNumbers(3) = 7	EvenNumbers(3) = 8
OddNumbers(4) = 9	EvenNumbers(4) = 10

Now organize the two lists as one two-dimensional array. The two subscripts used to reference the items are separated by commas; the first subscript designates the row, the second subscript designates the column.

Numbers(0,0) = 1	Numbers(0,1) = 2
Numbers(1,0) = 3	Numbers(1,1) = 4
Numbers(2,0) = 5	Numbers(2,1) = 6
Numbers(3,0) = 7	Numbers(3,1) = 8
Numbers(4,0) = 9	Numbers(4,1) = 10

The lower bound of both subscripts in the Numbers array is 0. Note that there are different upper bounds for each of the two subscripts: 4 and 1.

The above examples used numeric arrays. The computer also allows string arrays; see the "String Manipulations" chapter for details.

Dimensioning Arrays

Dimensioning an array establishes the array-subscript upper bound(s) and reserves computer memory for the array elements. After a variable is dimensioned, you can reference the individual elements by using the array name and the appropriate subscript(s). Here is a simple example:

```
100  DIM RealArray(9) !     Dimension the array.
110  !
120  RealArray(0)=123 !     Assign value to element 0.
130  RealArray(7)=3.142 !   Assign value to element 7.
140  !                      Now display an element's value.
150  DISP "Value of element 7 =";RealArray(7)
```

Here are the program's results:

```
Value of element 7 = 3.142
```

Array Subscript Bounds The array in the preceding example had 10 elements, specified by the array subscripts 0 through 9. The upper bound (9) was specified in the DIM statement. The maximum upper bound of any numeric array subscript is 65 530.

The lower bound of an array subscript, always 0 or 1, is established either by default or explicitly. Technical BASIC assumes that all array subscripts have a lower bound of 0, unless you specify otherwise using this statement:

```
OPTION BASE 1
```

Since the computer assumes OPTION BASE 0 unless told otherwise, the OPTION BASE 0 statement is used only for documentation purposes.

An OPTION BASE statement can be included *only* once in a program. Once an option base has been declared (or assumed), that option base is used throughout the program. The OPTION BASE declaration in a program must appear before any array variables are dimensioned or referenced. And you cannot execute an OPTION BASE statement from the keyboard after running a program.

Declaration Statements These declarative statements are available for dimensioning arrays – declaring the type and size of the array:

- REAL (and DIM)
- SHORT
- INTEGER
- COM

The DIM statement is used to declare REAL variables – both simple and arrays. The REAL statement is also used to dimension REAL variables; it is the preferred method, because it documents the variable's type more clearly.

```
10  OPTION BASE 1
20  REAL Light,Energy(20) !  Simple variable Light, and
30  !                        20-element array Energy.
```

All numeric variables in DIM statements, both simple and array, are assumed to be of type REAL. The only way to declare them to be of type INTEGER or SHORT is to explicitly declare them using the corresponding INTEGER or SHORT declaration statement.

The SHORT statement declares simple numeric and numeric array variables of type SHORT.

```
10  OPTION BASE 0
20  SHORT Change(9,15),Delta,PSI ! 160-element array Change,
30  !                              and simple variables Delta and PSI.
```

The INTEGER statement declares simple numeric or numeric array variables of type INTEGER.

```
10  OPTION BASE 0
20  INTEGER Day,Pointer(40) !  Simple variable Day, and
30  !                          41-element array Pointer.
```

The COM statement can declare variables of any type. It is used to reserve memory in *common storage*. Programs and subprograms use common storage for communicating with one another. Program-to-program communication is discussed in the "Chaining Programs" section of the "Program Structure and Flow" chapter. Subprogram-to-program and subprogram-to-subprogram communications are discussed in the "Program/Subprogram Communication" section of the "User-defined Functions and Subprograms" chapter.

Implicit Dimensioning You need not dimension an array if its upper bounds are less than or equal to 10. Any array not explicitly dimensioned (such as with DIM) is assumed to have upper bound(s) of 10. Here are the number of elements that implicitly dimensioned arrays will have:

	One-dimensional Array	**Two-dimensional Array**
`OPTION BASE 0`	11	121 (11*11)
`OPTION BASE 1`	10	100 (10*10)

If you want an array to have fewer elements, you must dimension it explicitly. This also conserves memory by allocating space for fewer elements.

Because of implicit dimensioning, the statement that *explicitly* dimensions an array variable must appear *before* any elements of the array are referenced. Otherwise, the system first dimensions the array implicitly and then reports an error when the second dimension is attempted (the explicit declaration)[1].

1 This error will be caught during program pre-run, which occurs at RUN and INIT. Pre-run is described in the section called "A Closer Look at Program Execution" in the "Program Structure and Flow" chapter.

Here is an example that will generate `Error 35 : DIM EXIST VRBL` ("attempted to dimension an existing variable"):

```
100  OPTION BASE 1
110  Array(3)=44 !   Implicitly dimensions 'Array(10)'.
120  DIM Array(10) ! Causes error 35 (at pre-run).
```

Regardless of the method used to dimension an array, it can be dimensioned *only* once in a program. A second attempt to dimension an array variable generates this pre-run error (35).

Array Variable Names The rules for naming simple numeric variables also apply to numeric arrays. The name can be up to 32 characters in length, and may contain alphabetic (uppercase and lowercase) characters, decimal digits, and the underscore (_) character. The only restriction is that the first character must be a letter.

In addition, a simple variable may be given the same name as an array variable. However, the simple variable is referenced using the name without subscripts, while an array element is referenced using one or two subscripts in parentheses. For example:

`Variable`	Simple numeric variable.
`Variable(2,4)`	Element of a numeric array.

Assigning Values to Individual Elements

Here is an example of dimensioning an array, assigning values to its elements, and displaying the array elements individually:

```
10  OPTION BASE 1
20  DIM Squares(8)
25  !
30  FOR Element=1 TO 8
40      Squares(Element)=Element*Element
50      PRINT Element;"times";Element;"=";Squares(Element)
60  NEXT Element
70  END
```

The program produces the following results:

```
1 times 1 = 1
2 times 2 = 4
3 times 3 = 9
4 times 4 = 16
5 times 5 = 25
6 times 6 = 36
7 times 7 = 49
8 times 8 = 64
```

Displaying and Printing Entire Arrays

Preceding examples have shown how to display and print individual array elements. However, it is often easier to use some resident features of the Technical BASIC system to do that for you. There are two statements for displaying and for printing arrays: MAT DISP and MAT PRINT. MAT DISP displays the array on the current (CRT IS) screen, while MAT PRINT prints arrays on the current (PRINTER IS) system printer.

Here are examples of using the MAT DISP and MAT PRINT statements:

```
100   OPTION BASE 1
110   REAL Array33(3,3) !  3x3 array.
120   !
130   DATA 11,12,13,21,22,23,31,32,33
140   MAT READ Array33
150   !
160   MAT DISP Array33, !  Trailing "," means 21-col. fields.
170   DISP
180   !
190   MAT DISP Array33; !  Trailing ";" means compact.
200   DISP
210   !
220   MAT DISP Array33/ !  Trailing "/" means line-feed.
230   !
240   END
```

Here are the program's results:

```
 11                   12                   13
 21                   22                   23
 31                   32                   33

 11  12  13
 21  22  23
 31  32  33

 11
 12
 13
 21
 22
 23
 31
 32
 33
```

The terminator following the array name (semicolon, comma, or slash) is used to specify the spacing between elements of the array.

Terminator	Spacing Between Elements
,	Fields – elements will be placed at the beginning of 21-column fields. (This is also the default terminator if another is not specified.)
;	Compact – elements will have one leading and one trailing space. If the number is negative, then the leading space is replaced by a minus sign.
/	One element per line.

You can also specify whether an array is to be displayed by rows (default) or by columns. Normally, vectors (one-dimensional arrays) are displayed or printed with one element per line. If you specify COL before the vector name, however, elements of the vector are displayed or printed across a line. Here is an example:

```
100  OPTION BASE 1
110  DIM Vector(9)
120  !
130  DATA 1,2,3,4,5,6,7,8,9
140  MAT READ Vector
150  !
160  MAT DISP Vector; !     One element per line.
170  !
180  MAT DISP COL Vector; ! All elements on same line.
190  !
200  END
```

Upon execution of this program, the following is displayed:

```
1
2
3
4
5
6
7
8
9
1  2  3  4  5  6  7  8  9
```

This statement displays Array33 by rows with compact spacing and then by columns with compact spacing.

```
100  OPTION BASE 1
110  REAL Array33(3,3) !  3x3 array.
120  !
130  DATA 11,12,13,21,22,23,31,32,33
140  MAT READ Array33
150  !
160  MAT DISP Array33; !      Default (by rows).
170  DISP
180  !
190  MAT DISP ROW Array33; !  By rows.
200  DISP
210  !
220  MAT DISP COL Array33; !  By columns.
230  DISP
240  !
250  END
```

Here is the program's output:

```
11  12  13
21  22  23
31  32  33

11  12  13
21  22  23
31  32  33

11  21  31
12  22  32
13  23  33
```

If you do not specify either ROW or COL, then the default (ROW) display order is used. If you specify ROW before an array name, elements are displayed or printed on each line by rows, beginning with the first row (0 or 1, depending on the current OPTION BASE in effect). Each row begins on a new line, and the elements in each row are listed in order from the first column to the last. More than one line may be required to list the elements in each row, depending on the terminator following the array name, the number of elements in each row, the number of digits in the values of the elements, and the display's screenwidth or printer's linewidth.

If you specify COL before an array name, elements are displayed or printed on each line by columns, beginning with the first column ("column-major" order). Each column begins on a new line; and the elements in each column are listed in order from the first row to the last. Again, more than one line may be required to list the elements in each column; this depends on the terminator following the array name, the number of elements in each column, the number of digits in the values of the elements, and the printer line width.

Specifying neither ROW nor COL before an array name has the same effect as specifying ROW.

If more than one array is specified, a blank line appears between the display or printout of each array.

Using Images You can achieve more complete control of the spacing between array elements with the MAT DISP USING and MAT PRINT USING statements.

One form of these statements includes an image string that specifies how array elements are displayed or printed.

```
MAT DISP USING "5D.D";Numbers,
```

Another form specifies the line number of an IMAGE statement.

```
100  IMAGE 5D.D
110  MAT DISP USING 100;Numbers,
```

As with the MAT DISP and MAT PRINT statements, specifying COL before the array name causes elements to be displayed or printed one column per display (or printer) line: the array elements are sent in column-major order, from first row to last row of a column, from the first column to the last column. Otherwise, elements are displayed or printed in row-major order. Here is an example:

```
100  OPTION BASE 1
110  DIM Array43(4,3),Vector10(10),Array45(4,5)
120  !
130  DATA 126.4,5.4,243.3,364.4,248.2,215.7
140  DATA 548.9,548.6,18.5,75,10.3,518.1
150  MAT READ Array43
160  PRINT "Array43:"
170  MAT PRINT USING "2X,3D.2D" ; Array43
180  PRINT
190  !
200  DATA 48,21,94,4,18,44,27,98,72,69
210  MAT READ Vector10
220  IMAGE "Vector10:"/10(DDD)
230  MAT PRINT USING 220 ; COL Vector10
240  PRINT
250  !
260  DATA 25,23,17,12,77,17,13,11,7,48
270  DATA 21,18,12,13,64,63,54,40,32,189
280  MAT READ Array45
290  PRINT "Array45:"
300  MAT PRINT USING 310 ; Array45
310  IMAGE 4(2D,X),X,3D/
320  !
330  END
```

This program prints the contents of three arrays using three different implementations of the MAT PRINT USING statement. Here are the results.

```
Array43:
  126.40    5.40  243.30
  364.40  248.20  215.70
  548.90  548.60   18.50
   75.00   10.30  518.10
```

```
Vector10:
 48 21 94  4 18 44 27 98 72 69

Array43:
25 23 17 12   77

17 13 11  7   48

21 18 12 13   64

63 54 40 32  189
```

Redimensioning Arrays

Once an array has been dimensioned, you can reorganize it into a different size by redimensioning it. This example dimensions Array4 with 4 elements, and then redimensions it to a working size of 3 elements.

```
100  OPTION BASE 1
110  DIM Array4(4) !   4-element array.
 .
 .
 .
400  REDIM Array4(3) ! Change working size to 3 elements.
```

Subsequent statements affect only the elements included in the new working size. In this example, you cannot access the 4th element of the array. Although this 4th element still exists in memory, the value of this element cannot be changed until the array is appropriately redimensioned (again).

The redimensioning subscripts are numeric expressions that specify a new upper bound for each dimension; they can be variables, constants, or arithmetic expressions. The number of subscripts must be the same as the number specified in the original DIM, REAL, SHORT, or INTEGER statement. For example, you cannot redimension a two-dimensional array into a one-dimensional array. Furthermore, the total number of elements in the new working size cannot exceed the number originally dimensioned. For example, you cannot redimension a 3×5 array into a 4×5 array, but you can redimension it into a 5×3 or a 7×2 array.

This example redimensions Array2 from a 3×5 array (15 elements) into a 4×2 array (8 elements).

```
100  OPTION BASE 1
110  DIM Array35(3,5) !     3x5 array (15 elements).
 .
 .
 .
400  REDIM Array35(4,2) !   4x2 array (8 elements).
```

This statement redimensions Array4 and Array35 to their original sizes.

```
   REDIM Array4(4),Array35(3,5)
```

4

This example redimensions Array29 from a 2×9 array into a 3×6 array.

```
100  OPTION BASE 0
110  DIM Array29(1,8)
 .
 .
 .
395  X=2 @ REDIM Array29(X,10/X)
```

When the array is redimensioned, the *values* in the array variable in memory are **not** changed. The only difference is that the *correspondence* between subscript(s) and elements are changed. The following example shows how values in an array variable are accessed when an array originally declared to be 3×3 is redimensioned into a 2×2 array.

```
100  OPTION BASE 1
110  DIM Array33(3,3)
120  !
130  DATA 11,12,13,21,22,23,31,32,33
140  MAT READ Array33 !   Reads in "row major" order.
150  !
160  MAT DISP Array33; !  3x3 array.
170  DISP
180  !
190  REDIM Array33(2,2) ! 2x2 array.
200  MAT DISP Array33;
210  DISP
220  !
230  REDIM Array33(3,3) ! 3x3 array.
240  MAT DISP Array33;
250  !
260  END
```

The results of running the program look like this:

```
11  12  13
21  22  23
31  32  33

11  12
13  21

11  12  13
21  22  23
31  32  33
```

The program first dimensions Array33 to be a 3×3 array. The MAT READ statement fills Array33 with values specified in the DATA statement. The DATA values are placed into the array in "row-major" order: all column elements of a row are read, beginning with the lowest-numbered column, and then the next higher-numbered row is read, and so forth through the highest-numbered row. Note that each element's value corresponds to its subscripts; for instance, Array33(1,1) = 11 and Array33(2,2) = 22.

The program displays Array33, and then redimensions it to a 2×2 array. The 2×2 array is then displayed, and redimensioned back to the original array size and displayed once again.

Note that redimensioning an array does **not** isolate a subarray. In other words, if you redimension a 3×3 array into a 2×2 array, the resulting array is not the 2×2 subarray in the upper left corner of the original array. Such operations are covered in the subsequent section called "Copying Subarrays."

REDIM is not the only statement that redimensions an array. The MAT...CON, MAT...ZER, and MAT...IDN statements allow you to optionally specify redimensioning subscripts. These statements assign certain values to the array specified. If redimensioning subscripts are specified, the array is redimensioned *before* the assignments are performed. See the corresponding sections later in this chapter for further information.

Implicit redimensioning may also be performed with statements that specify both a result array and an operand array. Here is an example:

```
100  OPTION BASE 1
110  DIM Array22(2,2),Array33(3,3)
120  !
130  DISP "Before assigning values from a smaller array."
140  MAT DISP Array33; !   Will contain all 0's.
150  DISP @ DISP
160  !
170  DISP "After assigning values from a smaller array."
180  MAT Array22=(2) !     Assign '2' to all elements.
190  MAT Array33=Array22 ! Now assign 2x2 to 3x3 array.
200  MAT DISP Array33; !   Now show new values.
210  !
220  END
```

Here are the results of running the program:

```
Before assigning values from a smaller array.
 0                    0                    0
 0                    0                    0
 0                    0                    0

After assigning values from a smaller array.
 2                    2
 2                    2
```

The program first dimensions two arrays: Array22 is a 2×2 array, while Array33 is a 3×3 array.

The contents of Array33 are then displayed. From the three rows of three columns of 0's, you can see that it is a 3×3 array.

The contents of the smaller Array22 (2×2) are assigned to the larger Array33 (3×3). The result of the assignment is that Array33 is first implicitly redimensioned to a 2×2 array, and then the contents of elements of Array22 are assigned to corresponding elements of Array33.

Here is a description of the general case. The result array (such as Array33 above) is redimensioned to accommodate the elements of the operand array (such as Array22 above) before the new values are assigned. The number of rows in the result array will then equal the number of rows in the operand array, and the same is true for the number of columns. If the size of the result array is greater than that of the operand array, then the result array is first redimensioned to match the size of the smaller operand array. Conversely, if the current size of the result array is smaller than that of the operand array, then the result array is first redimensioned to match the size of the larger operand array. Note, however, that this second case requires that the size of the result array (when originally dimensioned) was at least as large as the current size of the operand array; if not, an error is reported.

When an array has been redimensioned – either explicitly or implicitly – the array remains redimensioned even when the program that originally dimensioned it is run again. The array is **not** dimensioned back to the original size declared in the program's DIM, REAL, SHORT, or INTEGER statement. If a program is rerun, and it contains an array that is redimensioned (either in the program or from the keyboard), then a REDIM statement that specifies the *original* size should be included in the program between the DIM, REAL, SHORT, or INTEGER statement and the first statement or function that accesses the array or one of its elements.

Assigning Values to an Entire Array

Earlier sections showed examples of assigning values to individual array elements. This section describes several methods of assigning values to every element of an array with a single BASIC statement.

Assigning Values From the Keyboard The MAT INPUT statement allows you to assign values to elements of a numeric array from the keyboard. The MAT INPUT statement is programmable only; it cannot be executed from the keyboard. Here is an example:

```
100 OPTION BASE 1
110 DIM Vector(3),FirstMatrix(5,4),SecondMatrix(2
120 MAT INPUT Vector,FirstMatrix
130 RAD @ X=PI/6 @ Y=PI/3
140 MAT INPUT SecondMatrix
150 !
160 MAT PRINT Vector;
170 MAT PRINT FirstMatrix;
180 MAT PRINT SecondMatrix;
190 !
200 END
```

When the computer prompts you for the first element of Vector:

```
Vector(1)?
```

Enter all 3 elements of Vector as follows:

`1,2,3` **Return**

Since all three elements of Vector have been entered, the program then prompts you to enter the first element of the 5×4 array named FirstMatrix.

```
FirstMatrix(1,1)?
```

Respond to the prompt by entering the first 10 of twenty elements:

`1,2,3,4,5,6,7,8,9,10` **Return**

All elements are assigned values in order from lowest-numbered column to highest-numbered column within a row, beginning with the lowest-numbered row and finishing with the highest-numbered row. After you press the carriage return key, the computer displays the name of the next element to be assigned a value. In this case, the next prompt requests that you enter the eleventh element of FirstMatrix.

```
FirstMatrix(3,3)?
```

Respond to this prompt by entering the remaining 10 elements as follows:

```
11,12,13,14,15,16,17,18,19,20 Return
```

The next request that you enter the first element of the 2×3 array named SecondMatrix.

```
SecondMatrix(1,1)
```

Values can be entered as numbers, as numeric variables, or as numeric expressions. Input into the array continues until all elements have been assigned values. If an array becomes full in the middle of an input line, then the remaining elements on the line are ignored.

You can also enter expressions that contain variables and system-resident functions. Note that the values assigned to the variables and the values computed for the variable functions are entered into the corresponding elements.

Enter the following for SecondMatrix(1,1):

```
X,SIN(X),1-COS(2*X) Return
```

The final prompt given is:

```
SecondMatrix(2,1)
```

Now enter the remaining 3 elements into array SecondMatrix.

```
Y,COS(Y),1-SIN(2*Y) Return
```

The execution of your program is now complete. Here is what the program prints (assuming that you entered the values shown in preceding paragraphs):

```
1
2
3
1  2  3  4
5  6  7  8
9  10  11  12
13  14  15  16
17  18  19  20
.523598775598299  .5  .5
1.0471975511966  .5  .133974596215561
```

This program's purpose was to show you how the MAT INPUT statement prompts you to enter all values into an array. It also showed how several elements could be entered at one time. However, if you prefer entering elements individually, then you can enter one numeric expression at a time, pressing the carriage-return key after each one.

Assigning Values from a DATA Statement Like the READ statement, the MAT READ statement can be used in conjunction with one or more DATA statements. When MAT READ is executed, elements of the array are assigned values from the list of numbers in a DATA statement. Array elements are assigned values in row-major order, just as they are in the MAT INPUT statement. The items in a DATA statement that correspond to numeric array elements must be valid numeric values, not string values.

The MAT READ statement is programmable only; it cannot be executed from the keyboard. The following program is an example of using this statement.

```
100  OPTION BASE 1
110  INTEGER Numbers(2,5)
120  DIM Title1$[11],Title2$[12]
130  !
140  ! Years
150  DATA 1920,1930,1940,1950,1960
160  !
170  ! Numbers of U.S. drivers.
180  DATA 14,38,48,62,87
190  DATA "Millions of","U.S. Drivers"
200  !
210  MAT PRINT Numbers
220  READ Title1$,Title2$
230  !
240  PRINT Title1$
250  PRINT Title2$
260  PRINT "------------"
270  FOR Line=1 TO 5
280    PRINT Numbers(1,Line);Numbers(2,Line)
290  NEXT Line
300  !
310  END
```

The results of executing this program are as follows:

```
Millions of
U.S. Drivers
------------
 1920  14
 1930  38
 1940  48
 1950  62
 1960  87
```

Assigning the Same Value to Every Element The MAT statement also allows you to assign the value of a numeric expression to all elements of an array. For instance, this statement assigns the value 30.48 to all elements of array X.

```
MAT X = (30.48)
```

This statement assigns the value of the variable M to all elements of array Y.

```
MAT Y = (M)
```

This statement assigns the result of the expression 2*PI*Raduis^2 to all elements of array Z.

```
MAT Z = (2*PI*Radius^2)
```

Constant and Zero Matrices

The MAT...CON statement assigns the value 1 to *all* elements of an array. Here is an example:

```
MAT Array1 = CON
```

Every element in the array will now contain a value of 1.

You can also use this statement to redimension an array:

```
MAT Array3 = CON(2,2)
```

Assuming OPTION BASE 1, if array A was originally a 3×3 array, then it would be redimensioned to a 2×2 array. All elements of the redimensioned array would then be assigned a value of 1.

The MAT...ZER statement assigns the value 0 to *all* elements of the result array. An array in which all elements are zero is called a *zero matrix*. Likewise, a vector of which all elements are zero is called a *zero vector*. Here is an example:

```
MAT Array3 = ZER
```

You can also redimension the array:

```
MAT A = ZER(5,2)
```

Assuming OPTION BASE 1, if array A was originally a 5×5 array, then it would be redimensioned to a 5×2 array. All elements of the redimensioned array would then be assigned a value of 0.

Here is another example.

```
100  OPTION BASE 1
110  DIM Array43(4,3)
120  !
130  MAT Array43=ZER !    Zero 4x3 array.
140  MAT DISP Array43;
150  DISP
160  !
170  MAT Array43=CON(3,2) ! Redim, and assign 1 to first 6 elements.
180  MAT DISP Array43;
190  DISP
200  !
210  REDIM Array43(4,3) !  Restore 4x3 array subscripts.
220  MAT DISP Array43;
230  !
240  END
```

This program displays the following results:

```
0  0  0
0  0  0
0  0  0
0  0  0

1  1
1  1
1  1

1  1  1
1  1  1
0  0  0
0  0  0
```

The Identity Matrix

An identity matrix is created using the MAT...IDN statement. MAT...IDN assigns the value 1 to all *diagonal* elements of the result matrix and assigns the value 0 to all other elements. (Diagonal elements are those for which the row subscript is equal to the column subscript.) A matrix created using the MAT...IDN statement is also called a *unit matrix*. An example program using MAT...IDN is as follows:

```
100  OPTION BASE 1
110  DIM Array33(3,3),Array55(5,5)
120  !
130  MAT Array33=IDN
140  MAT DISP Array33;
150  DISP
160  !
170  MAT Array55=IDN(4,4)
180  MAT DISP Array55;
190  !
200  END
```

Execution of the above program produces the following result:

```
1  0  0
0  1  0
0  0  1

1  0  0  0
0  1  0  0
0  0  1  0
0  0  0  1
```

This program dimensions Array33 to be a 3×3 array and dimensions Array55 to be a 5 × 5 array. It also makes them both *identity matrices* and displays them. Note that Array55 is redimensioned to a 4×4 identity matrix by specifying the corresponding subscript values in the MAT...IDN statement. Note also, however, that you cannot redimension arrays to be larger than the dimensions specified in the declaration statement that originally specified the array's size.

The array specified in a MAT...IDN statement **must** be a square matrix; that is, it must have two dimensions, and the number of rows must be the same as the number of columns.

Copying Subarrays

An earlier section discussed copying the contents of an entire array into another array. For instance, this statement is used to copy every element of Array33 into Array55.

```
   MAT Array55=Array33
```

If Array33 and Array55 are of the same size, then each element of Array33 is copied into the corresponding element of Array55. However, if Array33 is a 3×3 array and Array55 is a 5×5 array, then this statement redimensions Array55 into a 3×3 array and then copies the nine elements of Array33 into corresponding elements of Array55.

With Technical BASIC, you can also copy a subset of an array (herein called a "subarray") into another array. Here is a simple example:

```
100  OPTION BASE 1
110  DIM Array55(5,5),Array33(3,3)
120  !
130  MAT Array55=(5) !  Fill with all 5's.
140  MAT DISP Array55;
150  DISP
160  !
```

```
170  MAT Array33=(3) !  Fill with all 3's.
180  MAT DISP Array33;
190  DISP
200  !
210  MAT Array55(2:4,2:4)=Array33 !  Put Array33 into subarray.
220  MAT DISP Array55;
230  !
240  END
```

Here are the program's results:

```
5  5  5  5  5
5  5  5  5  5
5  5  5  5  5
5  5  5  5  5
5  5  5  5  5

3  3  3
3  3  3
3  3  3

5  5  5  5  5
5  3  3  3  5
5  3  3  3  5
5  3  3  3  5
5  5  5  5  5
```

The example copies all of Array33 into columns 2 through 4 of rows 2 through 4 of Array55. The rest of Array55 is not changed, and the array is not redimensioned.

In general, you can specify both starting and ending row and starting and ending column for both the *operand* (Array33 above) and *result* (Array55 above). Subsequent examples explain this more clearly.

Several Examples of Copying Subarrays The following examples show several usages of the MAT statement in copying subarrays. For these examples, assume that OPTION BASE 1 is in effect and that all values in the 5×5 array named Result55 are set to zero before each statement is executed. The values shown for the Result arrays are the values that it will contain *after* the corresponding MAT statement has been executed.

This statement copies the value from each element of Operand55 into the corresponding element of Result55.

```
MAT Result55 = Operand55
```

Operand55					**Result55**				
11	12	13	14	15	11	12	13	14	15
21	22	23	24	25	21	22	23	24	25
31	32	33	34	35	31	32	33	34	35
41	42	43	44	45	41	42	43	44	45
51	52	53	54	55	51	52	53	54	55

This statement redimensions Result55 to a 3×3 matrix (since no subscripts were specified for the Result55 array); then it copies the values from columns 1 through 3 of rows 1 through 3 of Operand55 into the redimensioned Result55.

```
MAT Result55 = Operand55(1:3,1:3)
```

Operand55					**Result55**		
11	12	13	14	15	11	12	13
21	22	23	24	25	21	22	23
31	32	33	34	35	31	32	33
41	42	43	44	45			
51	52	53	54	55			

This statement copies the third element of Vector into the element in row 3 of column 2 of Result55.

```
MAT Result55(3,2) = Vector(3)
```

Vector	**Result55**				
1	0	0	0	0	0
2	0	0	0	0	0
3	0	3	0	0	0
4	0	0	0	0	0
5	0	0	0	0	0

This statement copies the values from row 1 columns 2 through 4 of Operand55 into row 3 columns 1 through 3 of Result55.

```
MAT Result55(3,1:3) = Operand55(1,2:4)
```

Operand55					Result55				
11	12	13	14	15	0	0	0	0	0
21	22	23	24	25	0	0	0	0	0
31	32	33	34	35	12	13	14	0	0
41	42	43	44	45	0	0	0	0	0
51	52	53	54	55	0	0	0	0	0

This statement copies the values from rows 4 and 5 of column 1 of Operand55 into rows 2 and 3 of column 5 of Result55.

```
MAT Result55(2:3,5) = Operand55(4:5,1)
```

Operand55					Result55				
11	12	13	14	15	0	0	0	0	0
21	22	23	24	25	0	0	0	0	41
31	32	33	34	35	0	0	0	0	51
41	42	43	44	45	0	0	0	0	0
51	52	53	54	55	0	0	0	0	0

This statement copies the entire Vector into the entire third row of Result55.

```
MAT Result55(3,) = Vector
```

Vector	Result55				
1	0	0	0	0	0
2	0	0	0	0	0
3	1	2	3	4	5
4	0	0	0	0	0
5	0	0	0	0	0

This statement copies the entire second column of Operand55 into Vector.

```
MAT Vector = Operand55(,2)
```

Operand55					Vector
11	12	13	14	15	12
21	22	23	24	25	22
31	32	33	34	35	32
41	42	43	44	45	42
51	52	53	54	55	52

This statement copies the values from rows 1 and 2 of columns 2 through 5 of Operand55 into rows 2 and 3 of columns 1 through 4 of Result55.

```
MAT Result55(2:3,1:4) = Operand55(1:2,2:5)
```

```
        Operand55                 Result55

11   12   13   14   15       0    0    0    0   0
21   22   23   24   25      12   13   14   15   0
31   32   33   34   35      22   23   24   25   0
41   42   43   44   45       0    0    0    0   0
51   52   53   54   55       0    0    0    0   0
```

This statement redimensions Result55 into a 2 × 3 matrix, and copies the values from rows 1 and 2 of columns 3 through 5 of Operand55 into Result55.

```
MAT Result55 = Operand55(1:2,3:5)
```

```
        Operand55                 Result55

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

Summary of General Rules

- The array elements are always copied and assigned in row-major order – from the first column to the last column of each row, beginning with the first row and proceeding through the last row.
- If all elements of the result array are to be assigned values, then do not specify row numbers or column numbers in the result array.
- If all elements of the operand array are to be copied into the result array, then do not specify row numbers or column numbers in the operand array.
- If row and/or column numbers are specified, they must be enclosed in parentheses and separated by a comma.

- If no row or column numbers are specified after the result array, then it is redimensioned (if necessary) before values are assigned to it. If row or column numbers are specified after the result array, then values are assigned to the corresponding elements and the array is not redimensioned.
- If the array is a vector, then specify only the row number(s).
- If only one row is to be copied or assigned values, then you need only specify that one row number; if more than one row is to be copied or assigned values, then specify the first row number and the last row number, separated by a colon. Similarly, if only one column is to be copied or assigned values, then you need only specify that one column number; if more than one column is to be copied or assigned values, then specify the first column number and the last column number, separated by a colon.
- If entire row(s) are to be copied or assigned values, then you may omit the column numbers but include a comma after the row number(s). Similarly, if entire column(s) are to be copied or assigned values, then you may omit the row numbers but include a comma before the column number(s).
- Unless either the operand array or the result array is a vector, the number of rows specified after the result array must be the same as the number of rows to be copied from the operand array. The number of columns specified after the result array must be the same as the number of columns to be copied from the operand array.
- Unless the operand array is a vector, a column from the operand array cannot be copied, using just one statement, into a row in the result array. Similarly, unless the result array is a vector, a row from the operand array cannot be copied, using just one statement, into a column of the result array. These types of copy operations can, however, be made using two statements, as shown in the next example.

In this example, row 1 of Array33 is copied into column 3 of array Result55, then column 3 of Array33 is copied into row 2 of array Result55.

```
100  OPTION BASE 1
110  DIM Operand33(3,3),Result33(3,3),Vector3(3)
120  !
130  DATA 1,2,3,4,5,6,7,8,9
140  MAT READ Operand33 !   Fill 3x3 matrix.
150  MAT DISP Operand33; !  Show contents.
160  DISP
170  !
180  MAT Result33=ZER !           Fill result with 0's.
190  MAT Vector3=Operand33(1,) !  Copy row 1 into vector.
200  MAT Result33(,3)=Vector3 !   Copy vector into column 3
210  MAT DISP Result33;
220  DISP
230  !
240  MAT Result33=ZER !           Fill result with 0's.
250  MAT Vector3=Operand33(,3) !  Copy column 3 into vector
260  MAT Result33(2,)=Vector3 !   Copy vector into row 2.
270  MAT DISP Result33;
280  !
290  END
```

Here are the results obtained from the program:

```
1  2  3
4  5  6
7  8  9

0  0  1
0  0  2
0  0  3

0  0  0
3  6  9
0  0  0
```

The first matrix displayed is Operand33. The next matrix displayed is matrix Result33, with column 3 containing values from row 1 of matrix Operand33. The final matrix displayed is Result33 again, but this time with row 2 containing values from column 3 of Operand33.

The row and column number(s) can be specified not only as constants, like those in the preceding examples, but also as variables or expressions. Here is an example:

```
245  Column=2
250  MAT Vector3=Operand33(,Column) !     Copy column 2 into vector.
255  BottomRow=3
260  MAT Result33(BottomRow-2,)=Vector3 ! Copy vector into row 1.
```

The first row (or column) number specified is usually less than the second row (or column) number. However, if the first row number is *greater* than the second (or if the first column number is greater than the second), then elements will be copied or assigned values in **reverse** order[1]. The first row or column number actually copied or assigned values is *one less than* the specified beginning row/column number. Similarly, the last row or column copied or assigned values is *one greater than* the specified last row/column number. For instance, the following statement copies rows 4 through 1 of the operand into rows 1 through 4 of the result array.

```
180 MAT Result(1:4,1:4)=Operand(5:0,1:4)
```

Here is an example program containing this statement:

```
100  OPTION BASE 1
110  DIM Result(4,4),Operand(4,4)
120  !
130  DATA 1,1,1,1,2,2,2,2,3,3,3,3,4,4,4,4
140  MAT READ Operand
150  MAT DISP Operand;
160  DISP
170  !
180  MAT Result(1:4,1:4)=Operand(5:0,1:4) ! Copy Operand rows 4 thru 1
190  !                                       into Result  rows 1 thru 4.
200  MAT DISP Result;
210  !
220  END
```

1 An important special case occurs when the first row number specified is *just one greater* than the second row number, or when the first column number specified is just one greater than the second column number. This subject will be discussed in the subsequent section called "A Special Case: Empty Arrays."

Here is the program's output:

```
 1  1  1  1
 2  2  2  2
 3  3  3  3
 4  4  4  4

 4  4  4  4
 3  3  3  3
 2  2  2  2
 1  1  1  1
```

The program reverses rows 1 through 4 of the Operand array as it copies them into the Result array.

A Special Case: Empty Arrays Here is the special case mentioned earlier: When the first row specified is *just one greater* than the second row, and with the corresponding case for columns, then *no elements will be copied or assigned values.* Furthermore, if no row or column numbers are specified after the result array (and OPTION BASE 1 is in effect), then the result array is redimensioned to have *zero rows or zero columns*[2]. The value of these features will be more apparent after we discuss this special case a bit more.

Examples of this special case are contained in this program:

```
100 OPTION BASE 1
110 DIM Operand(4,4),Result(4,4)
120 MAT Operand=CON !  Assign all 1's.
130 DISP
140 MAT DISP Operand;
150 DISP
160 K=1
170 MAT Result=Operand(1:K-1,2)
180 DISP "After copying Operand rows '1:0'."
190 MAT DISP Result;
200 DISP
210 MAT Result=Operand(1:4,1:K-1)
220 DISP "After copying Operand columns '1:0'."
230 MAT DISP Result;
240 !
250 END
```

2 If OPTION BASE 0 is in effect, the result array is not redimensioned, and a DIM SIZE error message is reported.

The program yields this output:

```
 1   1   1   1
 1   1   1   1
 1   1   1   1
 1   1   1   1

After copying Operand rows '1:0'.

After copying Operand columns '1:0'.
```

The display of Operand shows that it is indeed a constant matrix. The first display of array Result showed nothing because it was redimensioned to a 0×2 array, which is by definition an "empty" array. The second display of array Result was the same as the first, except that the number of columns is 0 and the array is redimensioned to a 4×0 empty array. Empty arrays should not be confused with zero arrays, which contain all 0's. If you should display or print an empty array, there will be no output since there are no elements in the array (according to current dimensions).

Empty arrays can be specified in subsequent statements and functions with meaningful results; the usual rules of redimensioning and row/column matching apply (in statements with two operand arrays). The following situations are of particular interest:

- Statements specifying only one operand array will, if that array is empty, redimension the destination array to be empty. For example, if the Operand array has been redimensioned to a 0×3 array, then this statement redimensions the Result array to 0×3:

  ```
  MAT Result = Operand
  ```

- If both operand arrays are empty, then performing a matrix multiplication[1] can yield a result array that is not empty. However, in such cases the statement assigns the value 0 to all elements of the destination array, regardless of the current values in the operand arrays.

1 Matrix multiplication is discussed in a subsequent section.

For example, if matrix Operand1 has been redimensioned to be 3 × 0, and matrix Operand2 has been redimensioned to be 0 × 1, then this statement:

```
MAT Result = Operand1*Operand2
```

redimensions matrix Result to 3 × 1, since 3 × 0*0 × 1 = 3 × 1 according to the rules of matrix multiplication. The Result array is not empty, since neither its number of rows nor number of columns is zero. However, it is a zero matrix, since the value 0 has been assigned to all (three) elements.The fact that no elements are copied or assigned values when the first row or column number is just one greater than the second, plus the characteristics previously described above for resulting empty arrays, simplifies programs that do certain matrix manipulations.

Scalar Arithmetic Array Operations

You can perform scalar arithmetic operations with a scalar numeric expression (such as a constant, variable, or expression) and each element of an operand array. For example, you could add the constant 4 to each element of an array:

```
MAT ArrayX=(4)+ArrayX
```

The resulting values are assigned to the corresponding elements of the result array (ArrayX).

The scalar arithmetic operations that you can perform with MAT keywords are as follows:

- Addition (+)
- Subtraction (−)
- Scalar multiplication (.), also known as the inner or dot product
- Division (/)

The following program makes Array44 an identity matrix. (An identity matrix is a square matrix which contains all ones in a diagonal which begins at its first element and moves down to its last element; the remainder of the elements in the identity array contain zeros.) The program then multiplies each element of that array by the scalar 2.

```
100  OPTION BASE 1
110  DIM Array44(4,4)
120  !
130  MAT Array44=IDN
140  MAT PRINT Array44;
150  PRINT
160  !
170  MAT Array44=(2)*Array44
180  MAT PRINT Array44;
190  !
200  END
```

The result of executing the above program looks like this:

```
1  0  0  0
0  1  0  0
0  0  1  0
0  0  0  1

2  0  0  0
0  2  0  0
0  0  2  0
0  0  0  2
```

If you need to change the signs of all elements in a matrix, you can do so inserting the following statement in the preceding program:

```
145 MAT Array44 = -Array44
```

The array now contains these values:

```
-2  0  0  0
 0 -2  0  0
 0  0 -2  0
 0  0  0 -2
```

You can also perform these scalar arithmetic operations with corresponding elements of two operand arrays: addition, subtraction, (dot-product) multiplication, and division. For instance, this statement calculates the squares of the elements in an array:

```
MAT Array33=Array33.Array33
```

The statement multiplies Array33(1,1) by Array33(1,1) and places the result in Array33(1,1). It does the same for each corresponding element of the operand arrays. Multiplication of corresponding elements is known as the dot product or inner product of the arrays; the operator for this type of operation is a period (.)[1]. Note that the two operand arrays **must** have the same number of elements in each dimension.

The result of two scalar multiplications can be added in one statement. An example is given below:

```
100  OPTION BASE 1
110  DIM Array1(2,4),Array2(2,4)
120  !
130  DATA 12,52,76,33,81,70,72,14
140  MAT READ Array1
150  !
160  MAT Array2=(50) @ Array2(1,2),Array2(2,1)=0
170  !
180  DEG ! Use degrees mode (for angular function
190  MAT Array1=(0.7)*Array1+(0.3*SIN(60))*Array2
200  MAT DISP USING "2X,DD.D" ; Array1
210  !
220  END
```

The results of executing this program are as follows:

```
21.3  36.4  66.1  36.0
56.7  61.9  63.3  22.7
```

Subtracting the results of two scalar multiplications can be accomplished in one statement by changing the sign of the second scalar. In the preceding example, change the statement 190 to:

```
190  MAT Array1=(0.7)*Array1+(-0.3*SIN(60))*Array
```

1 The asterisk (*) is used to denote matrix multiplication, which is a different kind of operation and is described in the subsequent section called "Matrix Multiplication."

Here are the modified program's results:

```
-4.5  21.3  40.2  10.1
56.7  36.0  37.4  -3.1
```

However, multiplying or dividing the results of two scalar multiplications cannot be performed with one statement.

Summing Rows and Columns

Technical BASIC provides MAT capabilities that allow you to compute the sum of elements in rows and columns of arrays.

- MAT...RSUM calculates the sum of elements in a row.
- MAT...CSUM calculates the sum of elements in a column.

Usages of MAT...RSUM and MAT...CSUM are shown in the following example.

Here is the Whackit Racket Company's monthly forecast data table. It is organized into sales regions (East, Midwest, and West) and racket model (WR01, WR02, and WR03).

Monthly Sales Forecast (Thousands of Units)

	Model			
Sales Region	**WR01**	**WR02**	**WR03**	**WR04**
East	25	23	17	12
Midwest	17	13	11	7
West	21	18	12	13

The program calculates and prints the total forecast for all racket models – one forecast by region, and another by racket model. Since each row contains the forecasts for all models in a region, the total forecast for all models in each region can be found using the MAT...RSUM statement. Likewise, since each column contains the forecasts for one model in all regions, the total forecast for each model in all regions can be found using the MAT...CSUM statement.

```
100  OPTION BASE 1
110  DIM Forecasts(3,4),RegionSums(3),ModelSums(1,4)
120  ! Forecast for the Eastern region.
130  DATA 25,23,17,12
140  ! Midwest region.
150  DATA 17,13,11,7
160  ! West region.
170  DATA 21,18,12,13
180  MAT READ Forecasts
190  !
200  PRINT "Forecasts:"
210  PRINT "----------"
220  MAT PRINT Forecasts;
230  PRINT
240  MAT RegionSums=RSUM(Forecasts) !  Row sums to vector.
250  PRINT "Forecasts by Region:"
260  PRINT "--------------------"
270  MAT PRINT RegionSums;
280  PRINT
290  PRINT "Forecasts by Model:"
300  PRINT "-------------------"
310  MAT ModelSums=CSUM(Forecasts) !  Columns sums to matrix.
320  MAT PRINT ModelSums;
330  !
340  END
```

The results displayed after program execution are:

```
Forecasts:
----------
 25  23  17  12
 17  13  11  7
 21  18  12  13

Forecasts by Region:
--------------------
 77
 48
 64

Forecasts by Model:
-------------------
 63  54  40  32
```

The first matrix displayed is the monthly sales forcast (in thousands of units); the rows correspond to regions, and the columns correspond to racket models. The second matrix has in its rows the total sales for the East, Midwest, and West regions, respectively. Finally the last matrix has in its columns the total sales of models WR01, WR02, WR03, and WR04, respectively.

General Rules Here are the rules that govern this type of operation:

- MAT...RSUM adds the values of the elements in each row of the operand array, and then assigns the sum to the corresponding element of the result array (a vector or single-column matrix). If the result array is a vector, it is first redimensioned (if necessary) to have as many elements as the number of rows as the operand array. If the result array is a matrix, it is first redimensioned (if necessary) to have one column and as many rows as in the operand array.
- Likewise, MAT...CSUM adds the values of the elements in each column of the operand array, and then assigns the sum to the corresponding element of the result array (a vector or single-row matrix). If the result array is a vector, it is first redimensioned (if necessary) to have as many elements as the number of columns as the operand array. If the result array is a matrix, it is first redimensioned (if necessary) to have one row and as many columns as in the operand array.

Array Transpose

The MAT...TRN statement computes the transpose of the operand array – interchanges the rows and columns of the array – and places the values in the result array. The following program shows how this statement can be used within a program:

```
100  OPTION BASE 1
110  DIM Array23(2,3),Array55(5,5)
120  DATA 1,2,3,4,5,6
130  MAT READ Array23
140  MAT PRINT Array23;
150  PRINT
160  MAT Array23=TRN(Array23)  ! Redim, then transpose.
170  MAT PRINT Array23;
180  PRINT
190  MAT Array55=TRN(Array23)  ! Redim, then transpose.
200  MAT PRINT Array55;
210  END
```

Here is the program's output:

```
1  2  3
4  5  6

1  4
2  5
3  6

1  2  3
4  5  6
```

The first matrix displayed is the original Array23 – a 2×3 matrix. The transpose of Array23 is then calculated, after which the Array23 variable is redimensioned (to a 3×2 matrix) and then and assigned the transposed Array23; the second matrix shown above is the result. Next, the transpose of Array23 is computed, then Array55 is redimensioned from a 5×5 matrix to a 2×3 matrix, and is then assigned the values of the transposed matrix Array23.

Matrix Multiplication

The MAT statement can be used to calculate the (outer) product of two arrays. The value of each element of the destination array is determined according to the usual rules of matrix multiplication. Here is an example:

```
A * B = Result
```

$$\begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix} * \begin{vmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{vmatrix} = \begin{vmatrix} a_{11}{}^*b_{11} + a_{12}{}^*b_{21} & a_{11}{}^*b_{12} + a_{12}{}^*b_{22} \\ a_{21}{}^*b_{11} + a_{22}{}^*b_{21} & a_{21}{}^*b_{12} + a_{22}{}^*b_{22} \end{vmatrix}$$

The number of *columns* in the first operand array (A) must be the same as the number of *rows* in the second operand array (B). The Result array has the same number of rows as the first operand array and the same number of columns as the second operand array. Either (but not both) of the operand arrays can be vectors.

An Example Here is an example program that performs the multiplication:

```
100  OPTION BASE 1
110  DIM Array23(2,3),Array32(3,2),Array22(2,2)
120  DATA 1,2,3,4,5,6
130  MAT READ Array23 !  Fill 2x3 matrix.
140  MAT DISP Array23;
150  DISP
160  DATA 1,4,2,5,3,6
170  MAT READ Array32 !  Fill 3x2 matrix.
180  MAT DISP Array32;
190  DISP
200  MAT Array22=Array23*Array32 !  Multiply.
210  MAT DISP Array22; !            Result is a 2x2 matrix.
220  END
```

Here are the program's results:

```
1  2  3
4  5  6

1  4
2  5
3  6

14  32
32  77
```

The first array displayed is the first operand. The second array is the second operand. The third array is the product of matrix multiplication on the two operands.

Another Example The following problem illustrates the use of matrix multiplication. The Whackit Racket Company is considering raising the prices on each of its four models: WR01, WR02, WR03, and WR04. The sales manager wants a program that uses the data in the following table to calculate and print a matrix that shows the total income (in thousands of dollars) in each of the three sales regions at the old and at the new prices. (The price increase is not expected to affect the number of units sold.)

Monthly Sales Forecast (Thousands of Units)

	Model			
Sales Region	WR01	WR02	WR03	WR04
East	25	23	17	12
Midwest	17	13	11	7
West	21	18	12	13

Price (Per Unit)

Model	Old	New
WR01	$10	$15
WR02	$20	$27
WR03	$35	$50
WR04	$60	$80

In each sales region, the total income (either at the old or at the new prices) can be determined by multiplying the quantity of each model by the price of each model, then adding the results. Applying this process to the data in the forecast and price tables above, multiply each entry in a row of the forecast table by the corresponding entry in a column of the price table, and then add the results. The sum could be entered into the same row and column of another table, in which each row shows the total income in a sales region and each column shows the total income at the old or at the new prices.

Since all this is just what happens in matrix multiplication, these calculations can be done compactly with the matrix multiplication Incomes = Forecasts*Prices, in which:

- The Forecasts matrix contains the sales forecasts (in thousands of units). The rows correspond to the three sales regions, and the four columns correspond to the four models.
- The Prices matrix contains the prices (per unit) of each model. The four rows correspond to the four models, and the two columns correspond to the two price lists (old and new).
- The Incomes matrix will contain the total income in each sales region at the old and at the new prices. The three rows will correspond to the three sales regions, and the two columns will correspond to the two price lists.

```
100 OPTION BASE 1
110 DIM Forecasts(3,4),Prices(4,2),Incomes(3,2)
120 ! Sales for East region.
130 DATA 25,23,17,12
140 ! Sales for Midwest region.
150 DATA 17,13,11,7
160 ! Sales for West region.
170 DATA 21,18,12,13
180 ! Prices.
190 DATA 10,15,20,27,35,50,60,80
200 MAT READ Forecasts,Prices
210 !
220 MAT Incomes=Forecasts*Prices
230 PRINT
240 PRINT " Old      New"
250 PRINT "Income   Income"
260 PRINT " (k$)     (k$)"
270 PRINT "------   ------"
280 IMAGE X,DC3D,4X,DC3D
290 MAT PRINT USING 280 ; Incomes
300 END
```

Executing the program produces this result:

```
  Old          New
 Income       Income
  (K$)         (K$)
 ------       ------
 2,025        2,806
 1,235        1,716
 1,770        2,441
```

The first row shows incomes from the East region. The second row shows incomes from the Midwest region. The third row shows incomes from the West region.

4

Transposing before Multiplying If you want to multiply two matrices, but the dimensions need to be transposed, you can multiply the transpose of one array by the other array. The following problem helps illustrate this usage.

Assume the manufacturing capacity of the Whackit Racket Company is limited this quarter; it can produce only a percentage of the rackets demanded. The table below shows the percentage that can be supplied to each region in the next two months. Using the forecast data in the table of the preceding problem, calculate and print a matrix showing how many of each racket model will be produced each month.

Production Quota (Percentage)

Sales Region	June	July
East	80	90
Midwest	75	85
West	85	95

The quantity of each racket model that will be produced (during either month) can be determined by multiplying the quantity of each model by the percentage for that model, and then adding the results. As in the preceding example, these calculations can be performed easily with a matrix multiplication of elements in the sales forecast table by elements in the production quota table. To do so, however, requires that the multiplication use the transpose of either the matrix containing the forecasts or the matrix containing the quotas. The following program multiplies the transpose of the matrix containing the forecasts by a matrix containing the quotas.

```
100 OPTION BASE 1
110 DIM Forecasts(3,4),Quotas(3,2),Units(4,2)
120 ! Sales for East region.
130 DATA 25,23,17,12
140 ! Sales for Midwest region.
150 DATA 17,13,11,7
160 ! Sales for West region.
170 DATA 21,18,12,13
180 ! Production quotas.
190 DATA 80,90,75,85,85,95
200 MAT READ Forecasts,Quotas
210 !
220 MAT Quotas=(0.01)*Quotas !  Converts percentages to decimal values.
180 MAT Units=TRN(Forecasts)*Quotas
190 PRINT
200 PRINT "  June       July"
210 PRINT "(K-Units)  (K-Units)"
220 PRINT "---------  ---------"
230 IMAGE 2X,2D.D,7X,2D.D
240 MAT PRINT USING 230 ; Units
250 END
```

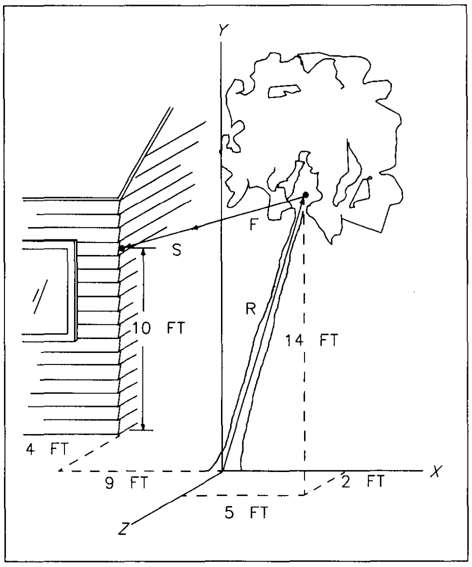
The results from executing this program are shown below:

```
  June       July
(K-Units)  (K-Units)
---------  ---------
  50.5       56.9
  43.4       48.8
  32.0       36.0
  25.9       29.1
```

Vector Cross Product

The MAT...CROSS statement calculates the vector cross product (or vector product) of two 3-element vectors. Mathematically, the cross product of two vectors is expressed as CrossProd = Vector3a × Vector3b. Each of the arrays named in the MAT...CROSS statement must be vectors; that is, they must have only one dimension. Arrays dimensioned like Matrix(3,1) are **not** allowed because they are two dimensional.

The following problem illustrates the use of MAT...CROSS. A leaning tree has a guyed wire connecting it to the corner of a house as shown in the picture. Calculate the moment of the force exerted by the guy wire about the base of the tree for a tension in the wire of 960 pounds.



To calculate the moment, use this formula:

Moment = Radius $\times$ Force

in which:

- Radius is the position vector of the guy wire point (on the tree) with respect to the base of the tree.
- Force is the 960-pound force vector exerted by the guy wire.

Before making the calculation, you will need to resolve the vectors Radius and Force into their components in the x, y, and z directions.

You can determine the components of Radius by looking at the drawing. They are as follows:

Radius(x) = 5
Radius(y) = 14
Radius(z) = 2

The components of Guy, from the illustration, are as follows:

Guy(x) = −9 − 5 = −14
Guy(y) = 10 − 14 = −4
Guy(z) = −4 − 2 = −6

The magnitude of Guy is equal to the square root of the sum of the squares of each component. Here is the equation:

MagGuy = SQR(Guy(x)^2 + Guy(y)^2 + Guy(z)^2)

The program makes this calculation in line 200 below.

You can now calculate each component of Force by multiplying the corresponding component of Guy by the ratio of the magnitude of Force to the magnitude of Guy.

$$\text{Force}(x) = \text{Guy}(x) \ast \frac{||\text{Force}||}{||\text{Guy}||}$$

The program performs this calculation in statement 230.

You will need to calculate the Force vector. Since you know that the 960-pound force is exerted in the direction of the Guy vector, the x, y, and z components of Force and Guy are proportional.

$$\frac{\text{Force}(x)}{\text{Guy}(x)} = \frac{\text{Force}(y)}{\text{Guy}(y)} = \frac{\text{Force}(z)}{\text{Guy}(z)} = \frac{|\text{Force}|}{|\text{Guy}|}$$

The unknowns in this equation are the x, y, and z components of Force; you already know its magnitude, and you can determine the components of Guy and its magnitude.

The components of Moment (in lb-ft) are determined by the following program:

```
100  OPTION BASE 1
110  DIM Radius(3),Force(3),Guy(3),Moment(3)
120  !
130  ! Components of Radius.
140  DATA 5,14,2
150  ! Components of Guy.
160  DATA -14,-4,-6
170  MAT READ Radius,Guy
180  !
190  ! Calculate magnitude of Guy.
200  MagGuy=SQR(Guy(1)^2+Guy(2)^2+Guy(3)^2)
210  !
220  ! Calculate components of Force vector.
230  MAT Force=(960/MagGuy)*Guy
240  !
250  ! Calculate components of Moment.
260  MAT Moment=CROSS(Radius,Force)
270  MAT PRINT USING "5D.DD" ; Moment
280  !
290  END
```

The components of Moment are as follows:

```
-4632.96
  121.92
10728.97
```

The x-component is printed first, then the y-component, and finally the z-component.

Inverting a Matrix

The MAT...INV statement finds the inverse of the operand matrix. The inverse of a matrix is the matrix that, when multiplied by the original matrix, results in an identity matrix. The main restriction on the operand matrix is that it must be square – that is, the number of rows must be the same as the number of columns.

Find the inverse of the matrix shown below. Check that when the inverse is multiplied by the matrix itself, the result is an identity matrix.

$$\text{Original} = \begin{vmatrix} 2 & 3 \\ 4 & 5 \end{vmatrix}$$

The following program provides a solution to the problem:

```
100  OPTION BASE 1
110  DIM Original(2,2),Inverse(2,2),Identity(2,2)
120  !
130  ! Elements of Original matrix.
140  DATA 2,3,4,5
150  MAT READ Original
160  IMAGE 3D.D
170  MAT PRINT USING 160 ; Original
180  PRINT
190  !
200  MAT Inverse=INV(Original)
210  MAT PRINT USING 160 ; Inverse
220  PRINT
230  !
240  MAT Identity=Inverse*Original
250  MAT PRINT USING 160 ; Identity
260  !
270  END
```

Here are the results of executing the program:

```
 2.0   3.0
 4.0   5.0

-2.5   1.5
 2.0  -1.0

 1.0   0.0
 0.0   1.0
```

The first four elements are the contents of matrix Original. The next group of four elements are the inverted contents of matrix Original. The last group of four elements show that the product of Original and inverse of Original is indeed an identity matrix.

You can also multiply the inverse of a matrix by another matrix using just one MAT statement. The syntax for this type of operation is as follows:

```
MAT Result = INV (Operand1) * Operand2
```

When the determinant of a matrix is zero, the matrix does not have an inverse. Therefore, if you attempt to find the inverse of such a matrix using the MAT...INV statement, `Error 112: DETERMINANT IS ZERO` is reported. You can use the DET (determinant) function to check the determinant before attempting to invert a matrix.

Calculating the inverse of a matrix is typically done in the process of solving the matrix equation: Coeffecients * Unknowns = Constants. However, a more accurate solution than the one provided by MAT Unknown = INV(Coefficients)*Constants can be obtained using the MAT...SYS statement, which is described in the next section.

Solving a System of Linear Equations

Suppose that you have a system of *n* linear equations with *n* unknowns. Here is the general system:

$$\begin{array}{ccccccc}
c(1,1)*x(1) & + & c(1,2)*x(2) & \ldots + & c(1,n)*x(n) & = & k(1) \\
c(2,1)*x(1) & + & c(2,2)*x(2) & \ldots + & c(2,n)*x(n) & = & k(2) \\
\vdots & & \vdots & & \vdots & & \vdots \\
c(n,1)*x(1) & + & c(n,2)*x(2) & \ldots + & c(n,n)*x(n) & = & k(n)
\end{array}$$

It can also be expressed using this matrix notation:

$$C * X = K$$

in which:

$$C = \begin{vmatrix} c(1,1) & c(1,2) & \ldots & c(1,n) \\ c(2,1) & c(2,2) & \ldots & c(2,n) \\ . & . & & . \\ . & . & & . \\ . & . & & . \\ c(n,1) & c(n,2) & \ldots & c(n,n) \end{vmatrix} \quad X = \begin{vmatrix} x(1) \\ x(2) \\ . \\ . \\ . \\ x(n) \end{vmatrix} \quad \text{and } K = \begin{vmatrix} k(1) \\ k(2) \\ . \\ . \\ . \\ k(n) \end{vmatrix}$$

C is the coefficient array; K is the constant array. The solution to this system of equations is the set of elements of array X.

The MAT...SYS statement is used to solve the matrix equation:

```
MAT X = SYS(C,K)
```

for the array X.

The following example illustrates the use of MAT...SYS in solving this system of equations:

2x + y - z = 0

x - y + z = 6

x + 2y + z = 3

First express the system of equations in matrix notation AX = B:

$$C = \begin{vmatrix} 2 & 1 & -1 \\ 1 & -1 & 1 \\ 1 & 2 & 1 \end{vmatrix}, \quad X = \begin{vmatrix} x \\ y \\ z \end{vmatrix}, \quad \text{and } K = \begin{vmatrix} 0 \\ 6 \\ 3 \end{vmatrix}$$

The program to solve this system of equations is as follows:

```
100  OPTION BASE 1
110  DIM C(3,3),X(3),K(3)
120  !
130  ! Read coefficient matrix.
140  DATA 2,1,-1,1,-1,1,1,2,1
150  MAT READ C
160  ! Read constant vector.
170  DATA 0,6,3
180  MAT READ K
190  !
200  ! Solve the equations.
210  MAT X=SYS(C,K)
220  !
230  ! Display solution.
240  MAT DISP X
250  !
260  END
```

Here are the results from executing the program:

```
 2
-1
 3
```

The value of x is 2, the value of y is -1, and the value of z is 3.

As mentioned earlier, the solution to the matrix equation C*X = K can also be obtained using the statement MAT X = INV(C)*K. The solution obtained using the statement MAT X = SYS(C,K) is somewhat more accurate; however, to achieve this accuracy two extra blocks of memory are used, each the size of the array X.

Although in typical applications the result array X and constant array K are each vectors or one-column matrices, the MAT...SYS statement does not restrict these arrays to only one column. This allows you, for example, to simultaneously solve two different systems of *n* equations in *n* unknowns, provided that the coefficients in both systems of equations are identical.

A useful example of this is described in the following problem. Your company's publications manager wants to determine the cost factors used by two outside printers. Each printer estimates jobs based on these criteria:

- The number of pages.
- The number of photographs.
- A fixed setup charge.

Here are estimates obtained from two printers for jobs that have varying numbers of pages and photographs.

Job	Number of Pages	Number of Photographs	Total Cost	
			Printer 1	Printer 2
1	273	35	$5835.00	$7362.50
2	150	8	$3240.00	$4085.00
3	124	19	$2775.00	$3517.50

Given the three estimates from each printer shown in the table above, you need to develop a program that calculates the printer's charge per page, cost per photograph, and once-per-job setup charge.

To solve the problem, you can set up the following system of equations for two sets of cost estimates:

```
273*x(1) + 35*x(2) + 1*x(3) = Estimate(1,Printer)

150*x(1) + 8*x(2) + 1*x(3) = Estimate(2,Printer)

124*x(1) + 19*x(2) + 1*x(3) = Estimate(3,Printer)
```

These equations can be represented in matrix notation as follows:

```
Items * CostFactors = Estimates
```

Items is the coefficient matrix containing the number of items for each job.

$$\text{Items} = \begin{vmatrix} 273 & 35 & 1 \\ 150 & 8 & 1 \\ 124 & 19 & 1 \end{vmatrix}$$

Each row contains data for a different job. Column 1 of each row contains the number of pages for the job. Column 2 of each row contains the number of photographs for the job. Column 3 of each row contains the number of setup charges for the job.

Estimates is the constant array that contains the cost estimates of three jobs (from two printers).

$$\text{Estimate} = \begin{vmatrix} 5835.00 & 7362.50 \\ 3240.00 & 4085.00 \\ 2775.00 & 3517.50 \end{vmatrix}$$

Each row contains cost estimates for one job. Column 1 contains printer 1's cost estimates for each job. Column 2 contains printer 2's cost estimates for each job.

CostFactors is the array that contains the unknown cost factors: x(1) is the cost per page, x(2) is the cost per photograph, and x(3) is the setup charge.

Since you are solving two systems of equations, the result array CostFactors must be a matrix; that is, it should originally be declared with two dimensions. (If CostFactor is not the same size as that of the constant array Estimates, then it is automatically redimensioned to the size of Estimates *before* the MAT...SYS statement is executed.) Each column contains the cost factors for one printer.

A program to solve this manager's problem is listed below:

```
100  OPTION BASE 1
110  DIM Items(3,3),CostFactors(3,2),Estimates(3,2)
120  !
130  ! Items for job 1.
140  DATA 273,35,1
150  ! Items for job 2.
160  DATA 150,8,1
170  ! Items for job 3.
180  DATA 124,19,1
190  MAT READ Items
200  !
210  ! Estimates for job 1.
220  DATA 5835,7362.5
230  ! Estimates for job 2.
240  DATA 3240,4085
250  ! Estimates for job 3.
260  DATA 2775,3517.5
270  MAT READ Estimates
280  !
290  ! Calculate cost factors.
300  MAT CostFactors=SYS(Items,Estimates)
310  !
320  ! Now print results.
330  PRINT "Printer 1   Printer 2"
340  PRINT "---------   ---------"
350  MAT PRINT USING "X,3D.2D,6X,3D.2D" ; CostFactors
360  !
370  END
```

The program displays the cost factors for each printer:

```
Printer 1   Printer 2
---------   ---------
  20.00       25.00
   5.00        7.50
 200.00      275.00
```

The first line of the table displayed above gives the cost per page. The second line gives the cost per photograph. The final line of the table gives the setup charge.

Additional Array Functions

Technical BASIC provides several functions that deal with numeric arrays. For instance, preceding sections gave examples of LBND, UBND, and DET. This section describes the array-related numeric functions that have not yet been described.

In the following descriptions, the variable(s) shown in parentheses (such as `Array` and `Subscript`) signify that the function requires numeric argument(s), which can be *any* numeric expression.

Function	Description
`ABSUM(Array)`	Returns the sum of the absolute values of all the elements in `Array`.
`AMAX(Array)`	Returns the value of the largest element in `Array`.
`AMAXCOL`	Returns the number of the column which contained the largest element in the array specified in the last AMAX function.
`AMAXROW`	Returns the number of the row which contained the largest element in the array specified in the last AMAX function.
`AMIN(Array)`	Returns the value of the smallest element in `Array`.
`AMINCOL`	Returns the number of the column which contained the smallest element in the array specified in the last AMIN function.
`AMINROW`	Returns the number of the row which contained the smallest element in the array specified in the last AMIN function.
`CNORM(Array)`	Returns the column norm of `Array` – the largest sum of absolute values obtained by summing the absolute values of elements in each column of the array.

`CNORMCOL`	Returns the number of the column (of the array specified in the last CNORM function) which contained the largest sum of absolute values.
`DET(Matrix)`	Returns the determinant of `Matrix`, which must be a square matrix.
`DETL`	Returns the determinant of the matrix which was last inverted. (Matrices are inverted by the MAT...INV and MAT...SYS statements).
`DOT(Vector1,Vector2)`	Returns the inner (dot) product of the two vectors.
`FNORM(Array)`	Returns the square root of the sum of the squares of all elements in `Array`. This value is known as the *Froebenius* or *Euclidian* norm.
`LBND(Array,Subscript)`	Returns the lower bound of `Subscript` in `Array` (i.e., the current OPTION BASE).
`MAXAB(Array)`	Returns the largest absolute value of any element in `Array`.
`MAXABCOL`	Returns the number of the column which contained the largest absolute value of any element in the array specified in the last MAXAB function.
`MAXABROW`	Returns the number of the row which contained the largest absolute value of any element in the array specified in the last MAXAB function.
`RNORM(Array)`	Returns the row norm of `Array` – the largest sum of absolute values obtained by summing the absolute values of elements in each row of the array.
`RNORMROW`	Returns the number of the row which contained the largest sum of absolute values in the array specified in the last RNORM function.
`SUM(Array)`	Returns the sum of all the elements in `Array`.
`UBND(Array,Subscript)`	Returns the upper bound of `Subscript` in `Array`.

5 String Manipulation

It is often desirable to store non-numerical information in the computer. For instance, you will often need to store and manipulate alphanumeric characters (text) with programs. This chapter describes several techniques for working with string data.

Chapter Contents

The sections of this chapter cover the following topics:

- What is a string?
- Evaluating string expressions
- Substrings
- String-related functions
- User-defined string functions
- Number-base conversions
- Additional string functions
- String array operations

What is a String?

A string is defined as any sequence of characters. A word, a name, or a message can be stored in the computer as a string. Each character in a string is stored as an eight-bit quantity; thus, there are 255 different characters available with Technical BASIC.

Assigning Values to String Variables

The following are valid assignments to sting variables. Quotation marks are used to delimit the beginning and ending of the string.

```
LET StringVariable$="computer"
Fail$="The test has failed."
FileName$="INVENTORY"
Test$=Fail$[5,8]
```

The left-hand side of the assignment (the variable name) is assigned the string value on the right-hand side of the assignment (the literal).

5

String Variable Names

String variable names are identical to numeric variable names with the exception of a dollar sign ($) appended to the end of the name. They may contain up to 32 characters, including all letters of the alphabet (both uppercase and lowercase), decimal digits 0 through 9, and the underbar (_) character. Just about the only restriction on string variable names is that the first character must be an alphabetic character.

String Variable Lengths

The length of a string is the number of characters in the string. In the previous example, the length of StringVariable$ is 8, since there are eight characters in the string literal "computer".

BASIC allows the dimensioned length of a string to range from 1 to 65 530 characters; the current length (number of characters in the string) can range from 0 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 and COM statements are used to define string lengths up to the maximum length of 65 530 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. A quote preceded by the tilde (~) character will embed a quote within a string.

```
10 Quote$="The time is ~"NOW~"."
20 PRINT Quote$
30 END
```

Produces: `The time is "NOW".`

Dimensioning String Variables

Strings whose length exceeds the default length of 18 characters must have space reserved before assignment. The following statements may be used.

`DIM Long$[400]`	Reserves memory for a 400-character string.
`COM Line$[80]`	Reserves memory for an 80-character string variable in "common" storage.

The maximum length of any string must not exceed 65 530 characters. A string may also be dimensioned to a length less than the default length of 18 characters.

Simple String Variables The DIM statement reserves storage for simple string variables.

```
DIM Part_number$[10],Decription$[64],Cost$[5]
```

The COM statement defines common variables that can be used by subprograms and chained programs.

```
COM Name$[40],Phone$[14]
```

Strings that have been dimensioned but not assigned values contain 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 each string array element, with the parentheses, which define the number of strings in the array. Each string in the array can be accessed by a subscript. For example:

```
PRINT File$(27)
```

Prints element 27 of the array. Since each character in a string uses one bytes of memory and each string in the array is allocated as many bytes as the maximum length of the string, string arrays can quickly use a lot of memory.

Evaluating String Expressions

This section describes how the Technical BASIC system evaluates string expressions.

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 concatenation operator "&". The following program combines two strings into one.

```
100 One$="WRIST"
110 Two$="WATCH"
120 Concat$=One$&Two$
130 PRINT One$,Two$,Concat$
140 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
`"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 greater in ASCII value than the letter "B".

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$[4]`	Specifies a substring starting with the fourth character of the String$ variable, and continuing through the end of the variable's current contents.

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. The subscript must be in the range: 1 to the string's current length (not dimensioned length). Any subscript value larger than this causes an error.

5

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]`	*(error)*

When a single subscript is used it specifies the starting character position of the substring, within the string. An error results when the subscript evaluates to zero or greater than the current length of the string.

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: `String$[Start,End]`

```
LET String$="JABBERWOCKY"
String$[4,6]
BER
```

In the following examples the variable B$ has been assigned to the literal "ENLIGHTENMENT":

Statement	Output
`PRINT B$`	`ENLIGHTENMENT`
`PRINT B$[1,13]`	`ENLIGHTENMENT`
`PRINT B$[1,9]`	`ENLIGHTEN`
`PRINT B$[3,7]`	`LIGHT`
`PRINT B$[4,4]`	`I`
`PRINT B$[13,26]`	*(error)*

An error results if either the first or the second subscript is greater than the current string length.

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:

```
100 A$="CONCAT"
110 A$[7]="ENATION"
120 PRINT A$
130 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 insertion of blank spaces.

It is a good practice to dimemsion all strings including those shorter than the default length of eighteen characters. This helps to manage the amount of memory space used by a string so that no memory space is wasted.

Some very interesting assignments can be attempted. For example, a 14-character string can be assigned to a 3-character substring.

```
100 Big$="Too big to fit"
110 Small$="Little string"
120 !
130 Small$[1,3]=Big$
140 !
150 PRINT Small$
160 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.

```
100 Big$="A large string"
110 Small$="tiny"
120 !
130 Big$[3,10]=Small$
140 !
150 PRINT Big$
160 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.

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 65 530. For example:

```
PRINT LEN("HELP ME")
```

Prints: 7

The following example program prints the length of a string that is typed on the keyboard.

```
  100 DIM In$[160]
  110 INPUT In$
  120 Length=LEN(In$)
  130 DISP Length;"characters in  ";In$
  140 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.

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    OPTION BASE 1
20    DIM Sentence$[40],Word$(6)[8]
30    DATA CAT,ON,A,HOT,TEN,NATION
40    FOR I=1 TO 6 @ READ Word$(I) @ NEXT I
50    Sentence$="WHERE IS THE CAT IN CONCATENATION"
60    !
70    FOR I=1 TO 6
80      Position=POS(Sentence$,Word$(I)) ! <- POS function
90      IF Position THEN SEG1 ELSE SEG2
100 SEG1:
110         PRINT Sentence$
120         PRINT TAB(Position);Word$(I);TAB(35);"is at ";Position
130         PRINT @ GOTO 170
140 SEG2:
150         PRINT "'";Word$(I);"' was not found"
160         PRINT
170       ! End of multi-line IF...THEN...ELSE construct.
180     NEXT I
190     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.

```
100 DIM A$[80]
110 A$="I know you think you understand what I said, but you don't."
120 INTEGER Scanner,Found
130 Scanner=1 !                         Current substring position
140 PRINT A$
150 REPEAT:
160 Found=POS(A$[Scanner]," ") !        Find the next ASCII space
170   IF Found THEN SEG1 ELSE SEG2
180 SEG1:
190     PRINT A$[Scanner,Scanner+Found-1] ! Print the word
200     Scanner=Scanner+Found @ GOTO 140 !  Adjust "Scan" past last match
210 SEG2:
220     PRINT A$[Scanner] !             Print last word in string
230 IF Found THEN REPEAT !              End of REPEAT construct.
240 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 89 will result.

```
Error 89 INVALID PARAM
```

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.

```
100 PRINT "Enter a fraction (i.e. 3/4)"
110 INPUT Fraction$
120 !
130 ON ERROR GOTO Err
140   Numerator=VAL(Fraction$)
150 !
160   IF POS(Fraction$,"/") THEN SEG1 ELSE SEG2
170 SEG1:
180      Delimiter=POS(Fraction$,"/")
190      Denominator=VAL(Fraction$[Delimiter+1])
200      GOTO 240
210 SEG2:
220      PRINT "Invalid fraction"
230      GOTO Err
240 !    End of multi-line IF...THEN...ELSE construct.
250 !
260      PRINT Fraction$;" = ";Numerator/Denominator
270      GOTO Quit
280 Err: PRINT "ERROR Invalid fraction"
290      OFF ERROR
300 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.

```
100 PRINT "Enter your first name."
110 INPUT Name$
120 PRINT Name$
130 PRINT
140 FOR I=1 TO LEN(Name$)
150   PRINT NUM(Name$[I]); ! Print value of each character
160 NEXT I
170 PRINT
180 END
```

Entering the name: JOHN will produce the following.

```
JOHN

 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 10000000000000000,VAL$(10000000000000000)
```

Prints: `1e+016                  1e+016`

Note that scientific notation does not start until there are seventeen digits to the left of the decimal point.

The next program converts a number into a string so the POS function can be used to separate the mantissa from the exponent. Note that this program only works with large positive exponents of size 16 or greater. For example, enter the following program:

```
100 PRINT
110 PRINT "Enter a number with an exponent"
120 INPUT Number
130 !
140 Number$=VAL$(Number)
150 !
160 PRINT Number$
170 E=POS(UPC$(Number$),"E")
180 IF E THEN SEG1 ELSE SEG2
190 SEG1:
200   PRINT "Mantissa is",Number$[1,E-1]
210   PRINT "Exponent is",Number$[E+1]
220   GOTO Quit
230 SEG2:
240   PRINT "No exponent"
250   GOTO Quit
260 Quit: END
```

The program when executed prompts you with the following:

```
ENTER A NUMBER WITH AN EXPONENT
?
```

Enter the following number with its exponent:

```
3E+16 RETURN
```

This returns:

```
3e+016
MANTISSA IS       3
EXPONENT IS       +016
```

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.

```
100 OPTION BASE 1
110 PRINT
120 CLEAR
130 !
140 DATA 34,89,111,117,32,103,111,116,32,105,116,3
150 INTEGER N(13)
160 MAT READ N
170 FOR I=1 TO 13
180   PRINT CHR$(N(I));
190 NEXT I
200 PRINT CHR$(7)
210 END
```

String Functions

Several additional string functions are available when using HP-UX Technical BASIC. This sections provides examples of these functions and a sample user-defined string function.

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.

```
100 DIM List$[30]
110 List$="3.22 4.33 1.10 8.55 12.20 1.77"
120 Length=LEN(List$)
130 Last_space=POS(REV$(List$)," ")
140 DISP "The last item is:";List$[1+Length-Last_space]
150 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.

```
100 OPTION BASE 1
110 DATA 50,900,2,444,37,2001,32768
120 DIM Array(7)
130 MAT READ Array
140 Maxdigits=0
150 FOR I=1 TO 7
160   Digits=INT(1+LGT(Array(I)))
170   IF Digits>Maxdigits THEN Maxdigits=Digits
180 NEXT I
190 Form$="XX,"&RPT$("D",Maxdigits)&".DD"
200 PRINT "Using the image: ";Form$
210 MAT PRINT USING Form$ ; Array
220 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.

```
100 DISP "Enter your first and last name."
110 INPUT Name$
120 First$=TRIM$(Name$[1,POS(Name$," ")])
130 Last$=TRIM$(Name$[1+LEN(Name$)-POS(REV$(Name$)," ")])
140 PRINT Name$,LEN(Name$)
150 PRINT Last$,LEN(Last$)
160 PRINT First$,LEN(First$)
170 END
```

If you need to enter leading or trailing spaces, use the LINPUT statement.

Lettercase Conversion

The lettercase conversion functions, UPC$ and LWC$, return strings with all characters converted to the same lettercase. UPC$ converts all lowercase characters to their corresponding uppercase characters, and LWC$ converts any uppercase characters to their corresponding lowercase characters.

```
100 DIM Word$[160]
110 LINPUT "Enter a few characters on this line: ",Word$
120 PRINT
130 PRINT "You typed: ";Word$
140 PRINT "Uppercase: ";UPC$(Word$)
150 PRINT "Lowercase: ";LWC$(Word$)
160 END
```

User-Defined String Functions

Many string functions not provided by Technical BASIC can be implemented separately as user-defined functions. The following program contains a string function.

```
100 DEF FNStmt$(X) = "Account #"&VAL$(X)
110 Acctnum=10699
120 DISP FNStmt$(Acctnum)
130 END
```

The results after executing this program are:

```
Account #10699
```

For a detailed discussion on user-defined string functions, read the chapter entitled "User-Defined Functions and Subprograms".

String Arrays

A string array is collection of character strings collected under the same string variable name and having the same maximum length. The computer allows both one- and two-dimensional string arrays.

Dimensioning String Arrays

The DIM statement is used to set the upper bounds of the string array and to specify the maximum number of characters in each element.

```
DIM String1$(25)[20],String2$(15,15)[20]
```

The one-dimensional array String1$ has an upper bound of 25 and a length per element of 20. The two-dimensional array String2$ has an upper bound of 15 for both its rows and columns and a length per element of 20. Note that the upper bound(s) and length per element cannot exceed 65530. The lower bound of a string array is determined by the OPTION BASE of the program. The OPTION BASE has no effect on the maximum string length.

String arrays, numeric arrays, and simple variables can be dimensioned in the same DIM statement. For example:

```
10 OPTION BASE 0
20 REM NAMES$ has 11 elements, each with maximum length of 25 characters.
30 REM GRADES has 66 REAL numeric elements.
40 DIM NAMES$(10)[25], GRADES(10,5)
.
.
.
```

If a string array is not explicitly dimensioned, it is implicitly dimensioned with upper bound(s) equal to 10 and maximum string length equal to 18.

The COM statement is used to dimension string arrays which are to be preserved in common between chained programs.

String Expressions and Operations

All the operations and functions provided for manipulating simple string variables can also be used with elements of string arrays.

Operations	Examples

Assignment	`STRING$(1)="eclipse"`
	`STRING$(2)="lunar"`
	`STRING$(3)="75"`

Concatenation	`EVENT$=STRING$(2) & " " & STRING$(1)`
	`DISP EVENT$`
	`lunar eclipse`

Substring	`MOUTH$=STRING$(1)[3,5]`
	`DISP MOUTH$`
	`lip`

Modification	`STRING$(2)[1,3]="sol"`
	`DISP STRING$(2)`
	`solar`

Comparison	`STRING$(2) < STRING$(1)`
	`0`

Functions	Examples

`LEN`	`LEN(STRING$(1))`
	`7`

`POS`	`PLACE= POS(STRING$(1),"p")`
	`DISP PLACE`
	`5`

`VAL`	`DISP VAL(STRING$(3))`
	`75`

`VAL$`	`STRING$(4)=VAL$(12345)`
	`DISP STRING$(3)&STRING$(4)`
	`7512345`

`CHR$`	`STRING$(5)=CHR$(40)`
	`DISP STRING$(5)`
	`(`

`NUM`	`DECVAL=NUM(STRING$(3))`
	`DISP DECVAL`
	`55`

`UPC$`	`SUN$=UPC$(STRING$(2))`
	`SUN$`
	`SOLAR`

The following program sorts a list of words alphabetically. Since string comparisons are based on the decimal codes assigned to each letter, all lowercase letters are converted to uppercase letters before sorting begins.

```
100 OPTION BASE 1
110 DIM Word$(20)[30] !         Dimensions 20-element string array.
120 FOR I=1 TO 16 !             This loop reads and prints DATA.
130   READ Word$(I)
140   Word$(I)=UPC$(Word$(I)) ! Converts word to all uppercase letters.
150   PRINT Word$(I);" ";
160 NEXT I !                    End loop.
170 PRINT
180 FOR J=2 TO 16 !             Begin sort.
190   Temp$=Word$(J)
200   FOR I=J-1 TO 1 STEP -1
210     IF Temp$>=Word$(I) THEN GOTO Insert
220     Word$(I+1)=Word$(I) !   Move element down one position.
230   NEXT I
240 Insert: Word$(I+1)=Temp$ ! Insert element at position I+1.
250 NEXT J
260 FOR I=1 TO 16 !             Print sorted list.
270   PRINT Word$(I);" ";
280 NEXT I
290 PRINT
300 DATA HOW,CAN,you,BE,IN,TWO,PLACES,AT,once,WHEN,YOU,ARE
310 DATA not,ANYWHERE,AT,ALL
320 END
```

6 User-Defined Functions and Subprograms

It is often handy to write algorithms that can be used in several places in a program or by other programmers. The "Program Structure and Flow" chapter described using subroutines for this purpose. Another handy feature of subroutines is that you can use them to "hide the details" of performing tasks from the "main" algorithm, so as not to obscure the readability of the main algorithm.

User-defined functions and subprograms also accomplish these two tasks, but they provide many additional capabilities. This chapter describes these two powerful features of the Technical BASIC language.

Chapter Contents

This chapter discusses the following topics.

- An introduction to user-defined functions
- Passing parameters
- Multiple-line functions
- Functions and local variables
- Data types and declarations
- An introduction to subprograms
- Benefits of using subprograms
- Creating, storing, and calling subprograms
- Deleting, loading, and editing subprograms
- Program/subprogram communication
- Passing parameters
- Using COM variables
- Using system flags
- Memory management with subprograms
- Context switching

User-Defined Functions

This section reviews some resident functions and then introduces you to user-defined functions. It describes several aspects of creating and using user-defined functions.

Review of Resident Functions

There are several resident functions built into the Technical BASIC language. Here are some examples:

```
Y=SIN(X+Phase)
Root1=(-B+SQR(B*B-4*A*C))/(2*A)
DISP "The value of pi = ",PI
PRINT "ASCII code = ";Number;"  Character = ";CHR$(Number)
```

In the first example, the SIN function calculates the sine of the argument X and returns the value so that it can be added to the value of the variable named Phase, and the sum then assigned to the variable Y.

In the second example, the SQR function calculates the square root of the argument B*B-4*A*C, which is then added to the negative value of the variable B, divided by the product 2*A, and this value is assigned to the variable Root1.

6

In the third example, the constant function PI returns the value 3.141 592 653 589 79, which is then displayed following the text.

In the fourth example, the CHR$ string function takes the numeric argument Number and returns the corresponding ASCII character.

Note that in all examples, the **functions return a single value**.

Introduction to User-Defined Functions

You can also define your own functions, which effectively allows you to extend the language if you need a function not provided in BASIC. Here are two examples:

```
X=1/FNSinh(Y^4)
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 define a function. On the other hand, if you want to actually change the data itself, generate more than one value, or perform any sort of I/O activity, it is better to use a subprogram. (A subsequent section describes subprograms.)

With this system, you can define either single-line or multi-line functions. Let's first look at an example of a single-line function.

Example Constant Function

Here is an example of a user-defined string function that returns a constant.

```
DEF FNName$="John Doe"
```

Since a constant function always returns the same value, there is no "argument" to be sent to it. Here are examples of using the function:

```
DISP "His name is ";FNName$
105  IF LEN(Name$)=0 THEN StudentName$(N)=FNName$
```

Functions can be defined anywhere in a program. They need **not** appear before they are referenced.

Let's look at a more common example – a function with argument(s).

Passing Parameters to Functions

The following line defines a function that computes the area of a circle, when supplied with a radius (the "argument").

```
50  DEF FNArea(Radius)=PI*Radius^2
```

Here are examples of invoking the function:

```
100  DISP "The area of a circle of radius 10 = ";FNArea(10)

250  Total_Area=FNArea(R1)+FNArea(R2)+FNArea(R3)
```

Note that a numeric value was "passed" to the function each time it was called: the function call in line 100 passed a value of 10; the function calls in line 250 passed values of variables R1, R2, R3. These values are known as *pass parameters*.

The variable named Radius in the Area function is known as a *formal parameter*. It specifies the variable in the function that is to receive the value passed to the function.

Parameter Lists From the preceding example, it is clear that there are two types of parameter lists:

- Formal parameter lists
- Pass parameter lists

The formal parameter list shows how many values may be passed to a function and gives the names the function will use to refer to those values. The formal parameter list for this example function:

```
50 DEF FNArea(Radius)=PI*Radius^2
```

is simply `(Radius)` – it is a list with one element.

The pass parameter list specifies the value(s) to be sent to the function. The pass parameter list for the following function call:

```
FNArea(10)
```

is simply `(10)`.

Each parameter in the pass parameter list corresponds to a parameter in the formal parameter list provided by the function. The function has the power to demand that the function call match the types declared in the formal parameter list exactly – otherwise an error results. It is also perfectly legal for both the formal and pass parameter lists to be null (nonexistent), as long as both match.

Single-line functions are not restricted to being passed one parameter; you can pass up to 16 numeric or 7 string parameters. These parameters include both simple numeric and string variables and numeric and string arrays.

An Example Multiple-Line Function

Since it is difficult to implement many significant functions while limited to one line of BASIC code, you can also define multiple-line functions. Here is a simple example.

```
110 PRINT "Decimal","Octal"
120 FOR Decimal_no=1 TO 100 STEP 5
130   PRINT Decimal_no,FNOctal(Decimal_no)
140 NEXT Decimal_no
150 STOP
160 !
170 DEF FNOctal(Decimal_Number)
180   Octal_Equiv=0
190   Remainder=Decimal_Number
200   FOR Octal_Place=10 TO 0 STEP -1
210     Octal_Digit=IP(Remainder/8^Octal_Place)
220     Remainder=Remainder MOD 8^Octal_Place
230     Octal_Equiv=Octal_Equiv+Octal_Digit*10^Octal_Place
240   NEXT Octal_Place
250 FNOctal=Octal_Equiv
260 FN END
```

The function's formal parameters are defined in the DEF FN statement. The value of the function is **not** defined in this declaration statement; instead, it is defined later in the function (line 250 in this example).

Functions and Local Variables

In general, all main program variables are accessible to functions. This is true whether they are declared explicitly (with statements such as DIM) or implicitly (for instance, numeric variables are assumed REAL unless explicitly declared otherwise). Here is an example:

```
10 Scale_factor=2
20 DEF FNXyz(Arg)=Scale_factor*Arg^3
30 DISP "FNXyz(2)=";FNXyz(2)
40 END
```

The results of this function call are 16 (=2*2^3) rather than 8 (=2^3). Thus, the main program's variable named Scale_factor was accessible to the function.

On the other hand, all variables declared in the *formal* parameter list are **not** accessible to the rest of the program; they are "local" to the function. This includes function variables that have the same name as a main program variable. For instance, the function's Radius variable is not available to the main program. Here is an example:

```
10 Radius=123
20 DEF FNArea(Radius)=PI*Radius^2
30 DISP "Result of 'Area(10)' =";FNArea(10)
40 DISP "Main program's variable 'Radius' =";Radiu
50 END
```

Here are the results of running the program.

```
Result of 'Area(10)' = 314.159265358979
Main Radius = 123
```

In line 10, the main program assigned a value of 123 to its variable named Radius. The call to Area (line 30) specified that the function's variable named Radius is to be assigned a value of 10. The results of line 30 verify that the function's variable named Radius was assigned a value of 10, while line 40 verifies that the main program's variable named Radius was not changed when the function's Radius was assigned the value of 10.

Formal Parameter Data-Type Declarations

Variables can be declared either implicitly or explicitly. Here is how variables are implicitly declared:

- Numeric variables are assumed to be of type REAL, unless explicitly declared INTEGER or SHORT.
- String variables are dimensioned to have a maximum length of 18 characters.

Explicit type declarations are made with DIM, REAL, SHORT, and INTEGER statements.

Since a function's formal parameters are local to the function, the type of each variable is implicitly declared in its DEF FN statement. Suppose, however, that you want to be able to pass a string longer than 18 characters to a function. In order to do that, you will need to declare a greater string length in the function heading. Here is an example:

```
DEF FNDo_something$(Arg$[100])= ...
```

Limitations

Functions **cannot be used recursively**. For instance, a function cannot call itself nor can it call another function that is used in its definition.

Functions are not restricted to being passed one parameter; you can pass up to 30 parameters to a user-defined function. These parameters include both simple numeric and string variables and numeric and string arrays. However, **user-defined string functions are restricted to returning a maximum of 18 characters**.

Introduction to Subprograms

This section shows you what subprograms are and gives you a glimpse of their capabilities. You will see how to enter and call two simple subroutines from a program.

Simple Examples

As described in the Program Structure and Flow chapter, subroutines are common routines that can be executed by several parts of the program. Subprograms are like subroutines in this respect, but they are much more powerful. But before discussing the additional capabilities that they provide, let's take a look at some simple examples.

Here is a "main" program that calls two subprograms. (A subsequent section shows how to enter, store, and load them.)

```
100  DISP "This is displayed by MAIN program."
110  DISP
120  CALL "FirstSub"
130  !
140  CALL "SecondSub" ("String pass parameter")
150  !
160  DISP "This is the MAIN program again."
170  !
180  END
```

Here are the subprograms.

```
                          100  SUB "FirstSub"
110  DISP "This is displayed by 'FirstSub'."
120  DISP
130  SUBEND

100  SUB "SecondSub" (Formal_param$)
110  DISP "This is displayed by 'SecondSub'."
120  DISP "The value sent to me is '";Formal_param$;"'."
130  DISP
140  SUBEND
```

Here are the results of running the program.

```
This is displayed by the MAIN program.

This is displayed by 'FirstSub'.

This is displayed by 'SecondSub'.
The value sent to me is 'String pass parameter'.

This is the MAIN program again.
```

Here is how the program flows. Executing RUN transfers control to the main program, beginning with line 100. This program line displays the first line of the results shown above. Control then moves to line 110 (DISP) and to line 120, which calls the subprogram named FirstSub. This CALL transfers control to the subprogram.

Lines 110 and 120 of FirstSub then display the third and fourth lines of the above results. Line 130 transfers control back to the calling context (here, the main program).

Line 140 of the main program calls the subprogram named SecondSub. The value `String pass parameter` is passed to the subprogram; it is known as a "pass parameter." Control is then given to SecondSub.

Lines 110 through 130 of SecondSub display the fifth through seventh lines of the above results; in particular, line 120 displays the value `String pass parameter`, which was passed to it by the main program. (You will see more about how parameters get passed in the section called "Program/Subprogram Communication.") Control is then returned to the calling program.

Finally, the main program (line 160) displays the final line of the above results, and program execution is finished when the END statement is reached (line 180).

These simple examples show that subprograms have several things in common with subroutines. Then why use subprograms? The next section provides the answer.

Benefits of Using Subprograms

Like subroutines, subprograms provide the main program with the ability to execute a common algorithm. A subprogram also depends on a main program and cannot be executed alone. It can execute internal subroutines, and can call other subprograms. However, subprograms also provide many *additional* capabilities.

The main power of subprograms is provided by these two characteristics:

- Subprograms can be **handled independently** – that is, they are not part of any main program, so they can be **created, stored, and retrieved separately**.
- You can **give (or deny) a subprogram access to any or all of the variables and values in the main program (or subprogram) that calls it**. You can "pass" specific parameters to them, or allow them to access specific common (COM) variables.

In short, a subprogram provides an easy way to isolate a useful programming routine, store it, and call it back into main memory for execution whenever needed.

There are several benefits to be realized by using subprograms:

- The subprogram allows the you to take advantage of the **"top-down" method of designing programs**. In this technique, the problem to be solved is broken up into a set of smaller and more easily solvable problems (known as "stepwise refinement.") 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.
- By separating all the details of performing the subtasks from the overall logic flow of the main program, the **program is much easier to read** (assuming you name the subprograms judiciously). 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 sub-task.
- **Subprograms can do everything a main program can do**. A subprogram has its own "context," or state, which is distinct from a main program and all other subprograms. This means that every subprogram has its own independent set of variables, DATA blocks, line labels, and so forth. Thus, you don't have think about *not* re-using such things as variable names and line labels used in the main program, because there will never be a conflict.
- 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**, it is easier to locate and correct problems. (This is also known as a "bottom-up" method of testing.)
- 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.

Difference Between Functions and Subprograms

A subprogram is invoked *explicitly* using the CALL statement. A user-defined function is called *implicitly* by using the function name in an expression. The function name can be used in a numeric or string expression the same way a resident system function or constant is used. A function's purpose is to return a single value (either a real number or a string). A subprogram's purpose is generally to calculate more than one value.

Creating and Calling Subprograms

Here are the general steps that you will need to take to enter a subprogram into memory, make a copy of it in mass storage, and call it from a program (or subprogram):

1. Determine what is currently in memory. (This step is optional.)
   a. Use the DIRECTORY statement to get a listing of the program and subprogram(s) currently in memory, if any.
2. If an unwanted program and subprogram(s) are currently in memory, then use SCRATCH to erase them.
3. Enter and store a main program (that calls a subprogram).
   a. Execute FINDPROG with no file name to "point" the editor at the main program's memory area.
   b. Enter the BASIC program lines, which include a CALL to the subprogram (to be written subsequently).
   c. Use the STORE command or SAVE statement to record a copy of the program in a mass storage file.

4. Enter a new subprogram.
   a. Use FINDPROG followed by a subprogram name to point the editor at a memory area to be used for the new subprogram.
   b. Enter the heading by typing SUB followed by the subprogram name (and formal parameters, if any).
   c. Enter the rest of the BASIC code for the subprogram.
   d. End the subprogram with a SUBEND or SUBEXIT statement.
   e. Use the STORE command to record a copy of the subprogram in file.
5. Run the main program.

Checking Memory Contents

Before entering any new program or subprograms, use the DIRECTORY statement to check what is currently in your BASIC memory area.

```
DIRECTORY
```

If no main program or subprogram is currently in memory, then you should see the following display.

```
    BASIC program            bytes   lines   allocated

  > MAIN                         0       0       no
```

If the main program and two subprograms shown in the preceding example were in your BASIC memory area, then you would see something like this:

```
    BASIC program            bytes   lines   allocated

  > MAIN                       196       9       yes
    FirstSub                    76       4       yes
    SecondSub                  156       5       yes
```

Here is a brief description of the columns of the DIRECTORY statement's results.

`BASIC program`	lists the name(s) of the subprogram(s) currently in <memory>. Also shows the size of the "MAIN" program. (There is no main program in memory of both the `bytes` and `lines` columns show 0).
`bytes`	shows the amount of memory required by the program or subprogram.
`lines`	shows the number of lines contained in the program or subprogram.
`allocated`	effectively indicates whether or not the program or subprogram has been "initialized" (by INIT) or run (by RUN).
`>`	indicates which program/subprogram can currently be edited, listed, etc.

You can simultaneously have one program and several subprograms within the memory allocated for your use by Technical BASIC. However, you can only "look at" **one of them at a time**. For instance, executing a LIST command would only show one of them – the one to which the `>` is pointing. More about this feature momentarily.

Entering a Main Program

Now point the system editor at the main program by executing a FINDPROG statement without specifying a file name:

```
FINDPROG
```

(This is a redundant step if you just executed a SCRATCH command as shown in the preceding discussion.)

Now execute this command to verify that you can enter the main program.

```
DIRECTORY
```

You should get this result:

```
    BASIC program          bytes   lines   allocated

  > MAIN                     0       0        no
```

The > pointing toward MAIN indicates that the main program is the one which you can now edit (or, in this case, the one you can enter). This is also the condition of memory at power-up.

Now enter the lines of the example program:

```
100  DISP "This is displayed by MAIN program."
110  !
120  CALL "FirstSub"
130  !
140  CALL "SecondSub" ("String pass parameter")
150  !
160  DISP "This is the MAIN program again."
170  !
180  END
```

Now store the main program using either the STORE command:

```
STORE "MainProg"
```

6

or the SAVE statement:

```
SAVE "MainProg"
```

The differences between these methods are as follows:

1. STORE creates a BASIC/PROG (''object'') file, while SAVE creates a BASIC/DATA (ASCII source) file.
2. LOAD retrieves files stored with STORE, while GET retrieves files stored with SAVE.

Entering a New Subprogram

Use the FINDPROG statement again, this time to point the editor at the beginning of the memory area which will be used for the subprogram that you will be entering. For simplicity, let's call the new subprogram FirstSub.

```
FINDPROG "FirstSub"
```

The system should respond with this message:

```
New program
```

If a subprogram file (of type BASIC/SUBP) with this name **already exists**, then this message is **not** shown, because the system automatically loads the subprogram from mass storage. However, if a program file (BASIC/PROG) with this name already exists, an `ERROR 68: FILE TYPE` error will be reported.

The FINDPROG statement also directs all subsequent program-editing operations (like DELETE, LIST, etc.) to be made on this subprogram.

You should now see something like this:

```
  BASIC program          bytes   lines   allocated

  MAIN                     196       9      no
> FirstSub                   0       0      no
```

The > points to the subprogram (or program) that will be the target of subsequent editing operations. After verifying that the editor is now pointed at subprogram "FirstSub", you can begin typing it in.

A Note about Naming Subprograms A subprogram has a name which may be up to 14 characters long, just as with line labels and variable names. Here are some legal subprogram names:

```
PlotDATA
InitializeDisc
Read_DVM
Sort_2D_array
```

Because up to 14 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.

The name of the subprogram specified in FINDPROG statement is also the name that you must specify in the STORE statement that stores the subprogram in a file. Although you can use any name in the SUB heading, it is probably best to use the same name there, also.

Entering a New Subprogram

Now that you have reserved a location in memory for the subprogram, you can begin typing it in. Enter this example subroutine.

```
100  SUB "FirstSub"
110  DISP "This is displayed by 'FirstSub'."
120  DISP
130  SUBEND
```

Note that the first line must contain the heading declaration SUB followed by the subprogram name. Note also that line numbers in the subprogram are completely independent of line numbers in the main program, so the subprogram can start with any (valid) line number.

6

Storing the Subprogram

Use the STORE command to store a copy of the subprogram in a file. You will need to specify the same name that you specified in the FINDPROG statement that pointed the editor at this subprogram. In this example, you would type:

```
STORE "FirstSub"
```

Only the first 14 characters of the file name are used if you specify one longer than 14 characters.

Now that the subprogram is stored, you can easily call it from any program (or from any other subprogram, for that matter).

Do not use the SAVE command to record a subprogram, because SAVE does not put the subprogram in the proper file type (BASIC/SUBP) for subsequent FINDPROG and CALL statements.

Entering and Storing the Second Subprogram

Now you are ready to perform similar steps to enter and store the second example subprogram. Here is another listing for your convenience.

```
100  SUB "SecondSub" (Formal_param$)
110  DISP "This is displayed by 'SecondSub'."
120  DISP "The value sent to me is '";Formal_param$;"'."
130  DISP
140  SUBEND
```

Repeat the procedure you used to enter and store the first subprogram.

Running the Program

Now that you have entered the program and both subprograms, you are ready to run the program. Execute:

```
RUN
```

You should get results like these:

```
This is displayed by the MAIN program.

This is displayed by 'FirstSub'.

This is displayed by 'SecondSub'.
The value sent to me is 'String pass parameter'.

This is the MAIN program again.
```

Subprograms Are Automatically Loaded

Since the subprogram in the preceding example was already in memory when the main program was executed, the system did not need to load it. However, if a subprogram is not in memory when called, then it will be *automatically* loaded. In order to verify that this is the case, let's first delete one of the subprograms.

Deleting a Subprogram

Use the SCRATCHSUB statement to delete a subprogram currently in your BASIC memory area. This example deletes subprogram FirstSub:

```
SCRATCHSUB "FirstSub"
```

Executing a DIRECTORY statement will now show that the subprogram is not in memory.

```
DIRECTORY

  BASIC program          bytes   lines   allocated

> MAIN                    196       9       yes
  SecondSub               156       5       yes
```

If you run the program at this point, the system will automatically load the subprogram when it is called.

Note that you can also execute SCRATCHSUB from a running program; see the subsequent "Memory Management with Subprograms" section for more complete details.

Explicitly Loading Subprograms (For Editing)

There are two general instances when subprograms will be loaded:

- When a program calls it.
- When you want to edit it.

As mentioned previously, when a subprogram that is not currently in memory is called, the system automatically loads it. Thus, about the only time that you will need to explicitly load a subprogram is for editing purposes.

To load a subprogram for editing purposes, merely execute a FINDPROG statement, specifying the subprogram name – which is also the file name. (It is possible to bring in as many subprograms into the computer as you like, limited only by available memory.) Let's load the example subprogram created in an earlier section:

```
FINDPROG "FirstSub"
```

Executing a DIRECTORY statement will verify that the subprogram has been loaded:

```
DIRECTORY

  BASIC program            bytes    lines    allocated

  MAIN                       196        9       yes
  SecondSub                  156        5       yes
> FirstSub                    76        4        no
```

Any subsequent LIST statements or editing operations (such as entering a line or using SCAN, etc.) will be performed on this subprogram.

For further information about loading and deleting subprograms, see the subsequent section called "Memory Management with Subprograms."

Now that you have the basic mechanics of entering, storing, and calling simple subprograms, let's look closer at some of their more powerful usages.

Program/ Subprogram Communication

As mentioned earlier, the two main features of subprograms that make them so powerful are as follows:

- You can handle each subprogram like a separate program.
- You can allow (or deny) a subprogram access to certain variables and values in the main program (or subprogram) that calls it.

This section disusses the second feature.

Here are the methods that a subprogram can communicate with the main program or with other subprograms:

- By passing parameters (through parameter lists)
- By sharing common variables (declared in COM statements)
- By using system flags.

System flags are accessible to every subprogram (and user-defined function). However, all variables and values in the calling program that are **not** explicitly passed to the subprogram or in COM are **not** accessible to the subprogram.

Passing Parameters

The second subprogram presented earlier in this chapter showed one means of communicating with a subprogram. Here is the relevant statement in the main program.

```
130  CALL "SecondSub" ("String pass parameter")
```

The characters `String pass parameter` are passed to SecondSub by specifying it as a pass parameter (in parentheses).

The SUB declaration in the subprogram (line 100) has a corresponding parameter – the string variable named Formal_param$.

```
100  SUB "SecondSub" (Formal_param$)
110  DISP "This is displayed by 'SecondSub'."
120  DISP "The value sent to me is "；Formal_param$；"'."
130  SUBEND
```

The subprogram has been defined to receive a string parameter from the context that calls it. Within the subprogram, this "local" variable will initially be assigned the value passed to it.

Here are the lines that the subprogram displays.

```
This is displayed by 'SecondSub'.
The value sent to me is 'String pass parameter'.
```

These results verify that the value specified in the CALL was the value that the subprogram received. Let's take a closer look at parameter lists.

Parameter Lists There are two kinds of parameter lists:

- Pass parameter lists
- Formal parameter lists

The calling context provides a pass parameter list. It contains values that are sent to the subprogram. Here is the pass parameter list used in a call to the preceding example subprogram:

```
130  CALL "SecondSub" ("String pass parameter")
```

Each item in this list corresponds to an item in the subprogram's formal parameter list.

The formal parameter list is part of the subprogram's definition. It immediately follows the subprogram's name. Here is the formal parameter list of the preceding example subprogram:

```
100  SUB "SecondSub" (Formal_param$)
```

The **formal parameter list serves three main purposes**:

- It tells *how many values may be passed* to a subprogram: the calling context can pass one value for every formal parameter[1].
- It *names the variables* that the subprogram will use to store and access those values.
- It *shows the general type*[2] *of each pass parameter* – numeric or string.

The *general type* of each pass parameter – numeric or string – must match the *general type* of the corresponding formal parameter; otherwise `Error 32 : PARAM MISMATCH` is reported. The next section provides more details on how the *specific type* of each parameter is declared.

1 Note that the calling context may also pass fewer parameters than are declared in the formal parameter list. See the subsequent section called "Optional Pass Parameters."

2 Note, however, that the formal parameter list does not *declare* the parameter's *specific* type – such as INTEGER for numeric parameters, and string length for string parameters. That subject is discussed in the subsequent section called "When Are Pass Parameter Types Declared?"

Methods of Passing Parameters There are two ways for the calling context to pass parameters to a subprogram:

- By reference (or "address").
- By value.

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

- To pass a parameter **by reference**, the pass parameter list (in the calling context) must use a **variable name** for that parameter.
- To pass a parameter **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.)

The **main difference** between the two methods is that **a subprogram can alter the value of a variable passed by reference** from the calling context. The calling context actually gives the subprogram access to its value area (for that variable). A parameter passed by value provides no such access.

Example of Passing by Reference This program passes a string variable and an INTEGER variable by reference:

```
100  ! Pass two parameters BY REFERENCE.
110  !
120  DIM String$[30]
130  String$="A string of thirty characters."
140  !
150  INTEGER Intgr
160  Intgr=32
170  !
180  DISP "Before pass by reference:"
190  DISP String$,Intgr
200  DISP
210  CALL "ChangeParams" (String$,Intgr)
220  !
230  DISP "After pass by reference:"
240  DISP String$,Intgr
250  DISP
260  !
270  END
```

Here is the subprogram:

```
100  SUB "ChangeParams" (Formal$,FormalN)
110  !
120  DISP "At subprogram entry:"
130  DISP Formal$,FormalN
140  DISP
150  !
160  Formal$="Short string."
170  FormalN=FormalN*2
180  !
190  DISP "At subprogram exit:"
200  DISP Formal$,FormalN
210  DISP
220  !
230  SUBEND
```

Here are the results of running the program.

```
Before pass by reference:
A string of thirty characters.                 32

At subprogram entry:
A string of thirty characters.                 32

At subprogram exit:
Short string.        64

After pass by reference:
Short string.        64
```

The program passes the variables String$ and Intgr to the subprogram by reference. The subprogram accesses them as Formal$ and FormalN, but it is actually accessing the main program variables String$ and Intgr. When the subprogram changes the value of these variables, the change is made directly to the main program's String$ and Intgr variables.

Example of Passing by Value This program passes a string value and an INTEGER variable by value:

```
100  ! Pass two parameters BY VALUE.
110  !
120  DIM String$[30]
130  String$="A string of thirty characters."
140  !
150  INTEGER Intgr
160  Intgr=32
170  !
180  DISP "Before pass by value:"
190  DISP String$,Intgr
200  DISP
210  CALL "ChangeParams" ("String value.",(Intgr))
220  !
230  DISP "After pass by value:"
240  DISP String$,Intgr
250  DISP
260  !
270  END
```

Note that the program calls the same subprogram as in the last example. Here are the results of running the program.

```
Before pass by value:
A string of thirty characters.              32

At subprogram entry:
String value.        32

At subprogram exit:
Short string.        64

After pass by value:
A string of thirty characters.              32
```

These parameters were passed by value, which means that the values were assigned to the formal parameters Formal$ and FormalN but *no* "addresses" of any main program variables were passed with the values. Thus the value of Intgr was not changed, because the subprogram did not have access to the variable.

When Are Pass Parameter Types Declared? As you studied the preceding examples, you may have wondered just how and when the data type of a subprogram's formal parameters are declared. The answer depends on how the parameter was passed to the subprogram:

- If the parameter is passed *by reference*, then the corresponding formal parameter *inherits* information about the variable from the calling context (data type, simple or array variable, etc.).
- If the parameter is passed *by value*, then the formal parameter has the *default* attributes for that general data type: 18 characters for strings, and REAL for numerics.

Thus it is possible, for example, to pass an INTEGER, SHORT, or REAL variable to a subprogram without causing `ERROR 113 : PARAM MISMATCH`. (Of course, the corresponding formal parameter must be a numeric variable.) Here is a simple example:

```
100  ! Explicitly declare an INTEGER.
110  INTEGER Intgr ! Explicitly declared an INTEGER.
120  Intgr=32
130  CALL "ShowParam" (Intgr)
140  !
150  ! Implicitly declare a REAL.
160  Number=12.34
170  CALL "ShowParam" (Number)
180  !
190  END
```

Here is the subprogram that it calls.

```
100  SUB "ShowParam" (AnyNumeric)
110  !
120  DISP "Value of numeric parameter =";AnyNumeric
130  DISP
140  !
150  SUBEND
```

Here are the program's results.

```
Value of numeric parameter = 32

Value of numeric parameter = 12.34
```

The program first *explicitly*[1] declares the simple numeric variable named Intgr to be of type INTEGER, assigns it a value, and then passes the value by reference to the subprogram. The subprogram then displays the value, and returns control to the calling program.

The program then *implicitly* declares the simple numeric variable named Number to be of type REAL, assigns it a value, and then passes the value to the subprogram by reference. The subprogram displays this value, and then returns control to the calling program.

In summary, the declaration of a variable's type, whether explicit or implicit, is made in the *defining context* – the program or subprogram in which it was declared. When a variable is passed to a subprogram *by reference*, information about the variable (type; simple or array, and array size; etc.) is inherited by the subprogram.

Optional Pass Parameters Another important feature of passing parameters is that **all pass parameters are optional**. However, the rules requiring matching of parameter types still apply. For instance, you may legally pass just three parameters to a subprogram that lists five formal parameters. However, these three pass parameters must match (in order, by type) the *first* three formal parameters. You cannot pass, for example, only the last three parameters.

There is a standard function called NPAR which can be used inside the subprogram to find out how many parameters the calling context actually did pass. If no parameters are passed to the subprogram, NPAR will return 0. (If used inside the main program, it will also return 0.)

The optional parameter feature 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.

1 Further details of explicit and implicit type declarations are given in the "Numeric Computation" and "String Manipulation" chapters.

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 numbers 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 parameters, the range is most likely to be altered.

A reasonable setup routine for the voltmeter might look like this:

```
2000 SUB "DVM_Setup" (Dvm,Readings,Delay,PRange,PTrigr,PFormat)
2010 ! Assume that AT LEAST 3 parameters will always be passed.
2020 !
2030 ! (Re)set defaults.
2040 Range=2 ! 1-Volt range.
2050 Trigr=1 ! Internal trigger.
2060 Format=1 ! ASCII format.
2070 !
2080 IF NPAR<4 THEN GOTO Build_Strings
2090    Range=PRange
2100    !
2110 IF NPAR<5 THEN GOTO Build_Strings
2120    Trigr=PTrigr
2130    !
2140 IF NPAR<6 THEN GOTO Build_Strings
2150    Format=PFormat
2160    !
2170 Build_Strings: Rdngs$="N"&VAL$(Readings)&"S"
2180                Delay$="D"&VAL$(Delay)&"S"
2190                Range$="R"&VAL$(Range)
2200                Trigr$="T"&VAL$(Trigr)
2210                Format$="F"&VAL$(Format)
2220                !
2230 OUTPUT Dvm ; Rdngs$&Delay$&Range$&Trigr$&Format$
2240 !
2250 SUBEND
```

The subprogram defines defaults for voltmeter range, trigger, and format modes (lines 2040 through 2060) for the instances when these parameters are not passed to it. If, for instance, a value for range is passed (through PRange), then the subprogram assigns this value to the *local* variable named Range.

Legal invocations of the Setup_dvm subprogram are as follows:

```
570 CALL "DVM_Setup" (Dvm,100,0.001) !       Default Range,Trigr,Format
630 CALL "DVM_Setup" (Dvm,500,0.05,3) !      Default Trigr,Format.
850 CALL "DVM_Setup" (Dvm,50,0.005,1,2) !    Default Format.
950 CALL "DVM_Setup" (Dvm,50,0.005,1,2,2) ! Explicitly define all params.
```

Using COM Variables

Since we've discussed parameter lists in detail, let's turn now to the second method a subprogram has of communicating with the main program or with other subprograms – using COM variables[1].

Here is an example of a valid COM declaration:

```
10   OPTION BASE 1
20   COM Array(15),INTEGER,CMin,CMax,Pile_status$[20],Tolerance
```

The following COM declaration would be legal in a subprogram (or in a chained program that is to keep the same COM structure):

```
100   OPTION BASE 1
110   COM Z(15),INTEGER,MinC,MaxC,St$[20],ErrorMax
```

As in parameter lists, COM variables are matched by *position* and *type*, not by variable names. Note that the OPTION BASE must match, and that COM statements must be placed following the OPTION BASE statement and before any other reference to the variable.

1 Note that COM variables can also be used for program-to-program communications when chaining programs; however, COM cannot be used within a user-defined function. See the chapter called "Program Structure and Flow" for a description of chaining. The subsequent section of this chapter called "Passing Flags to Chained Programs" describes using COM for program-to-program communications during chaining.

Note also that, from left to right in a given COM list, all variables following a numeric data-type declaration keyword have that numeric type until another numeric declaration keyword appears in the list. In the above examples, both CMin and CMax (MinC and MaxC) are INTEGERs, but Tolerance (ErrorMax) is a REAL variable; this effect is due to the fact that Pile_status$ (St$) is a string, which causes the following numeric variable to be of the default numeric type REAL.

Consider the following COM declaration:

```
10  COM INTEGER Range,Format,N,REAL Delay,Lastdata(40),Status$[20]
```

The following COM block matches the preceding COM block explicitly and is legal:

```
110 COM INTEGER Range,Format,N,REAL Delay,Lastdata(40),Status$[20]
```

The following COM declaration within a different subprogram matches the preceding COM statement and is also legal. (Even though some variables' names have been changed, the order and number of variables and their types are the same).

```
110  COM INTEGER R,F,N,REAL D,L(40),S$[20]
```

The following declaration is **illegal**, since it uses explicit size specifications on the array and string which do **not** match the original definition (line 10).

```
120  COM INTEGER Range,Format,N,REAL Delay,Lastdata(30),Status$[15]
```

The following declaration is also **illegal**, since it violates the types set forth by the defining block (here Range, Format, and N are implicitly declared to be of type REAL).

```
120  COM Range,Format,N,REAL Delay,Lastdata(40),Status$[20]
```

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

COM survives pre-run[1]. In general, all numeric variables are assigned values of 0 and strings are assigned the null string by executing RUN or INIT, or upon entering a subprogram; this is also true of COM the *first* time RUN or INIT is executed. However, after COM variables are defined, they retain their values until one of the following conditions occurs:

- SCRATCH is executed.
- A COM statement is modified.
- LOAD or CHAIN loads a new program which has a COM structure that **doesn't** match the existing COM structure (which includes programs that don't declare any COM 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 such things as the line number, possibly a line label, and the subprogram header. 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. All COM statements within a context are part of the definition of that context's COM structure. COM statements can be interwoven with other statements, though this is considered sloppy practice.

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[1]. For instance, one routine might be responsible for setting the voltage range on a voltmeter, while another routine which may need to know what the current voltage range is in order to set up the scale on a graph properly. (Technical BASIC also has system flags which you can use for this purpose. See the subsequent section of this chapter called "Using System Flags" for details.)

1 Pre-run is described in the section called "A Closer Look at Program Execution" in the "Program Structure and Flow" chapter.

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 CALL and SCRATCHSUB 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 COM variable), 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.

An example of this situation might include data base management libraries: hashing tables may need to be maintained for accessing data quickly. Three dimensional graphics libraries are another example: window, viewport, and clip information need to be kept, as well as object definitions and related transformations.

Using System Flags

System flags are the third method for communications between a main program and its subprograms (and also subsequently chained programs). In programming, the term "flag" denotes an indicator or reminder. Flags are used flags for various functions, such as in determining when to branch or in calculating the value to be assigned to a variable. For instance, you can use a flag to keep track of which mode a routine is operating in and thus whether to call a subroutine:

```
200  InsertMode=1 ! Set the flag.
 .
 .
 .
300  IF InsertMode THEN GOSUB Insert !  Branch if flag set.
```

With Technical BASIC, you can also resident flags (numbered 1 through 64). Here is an example analogous to the preceding one:

```
100  InsertFlag=10 !  Specify system flag used for Insert Mode.
 .
 .
 .
200  SFLAG(InsertFlag) !  Set the flag.
 .
 .
 .
300  IF FLAG(InsertFlag) THEN GOSUB Insert !  Branch if flag set.
```

If the FLAG function returns a 1, then the program branches to the subroutine called Insert.

General System Flag Features Each of 64 flags which can be individually set and cleared. When set, a flag contains a value of 1. When cleared, its value returns to 0. They are initially cleared upon entering the BASIC system.

The normal scope of system flags is a program and its subprogram(s), since executing a CHAIN statement clears all flags. However, you can store the flags in COM, as discussed later in this section, to pass them to chained programs.

While flags are usually used within running programs, they can also be set, tested, and cleared from the keyboard.

Setting Flags Individual flags are set by using the SFLAG statement. For instance, this statement sets system flag 32 (to 1).

```
SFLAG 32
```

Note that this statement may be used within a program:

```
100  LET MinFlag=MIN(N1,N2)
110  SFLAG MinFlag
```

Reading Flags The following function call determines the current setting of system flag 32:

```
FLAG(32)
```

If the flag is set, then this function call returns the *numeric* value 1; if currently clear, then 0 is returned.

Clearing Flags The following statement clears system flag 32:

```
CFLAG 32
```

Executing this function call now returns a numeric value of 0:

```
FLAG(32)
 0
```

The CFLAG statement clears one flag at a time, whereas executing INIT, RUN, or CHAIN clears all 64 flags. Note also that a parameter less than 1 or greater than 64 will generate an error report. Also, both CFLAG and SFLAG rounds flag numbers containing fractional parts.

Accessing System Flags as a String The concise way to set or clear each of the 64 flags in a single statement is to use this syntax of the SFLAG command:

```
SFLAG FlagString8$
```

This FlagString8$ expression may contain up to 8 characters (64 bits) of information. The value of the characters in the string determine whether flags are set or cleared: flags that correspond to 1 bits (in the binary representation of the character) are set, and flags that correspond to 0 bits are cleared.

For example, if you were to set the flags using the character string "ABCD0123", you could determine the resultant bit patterns (and corresponding flag settings) using the following method:

Character	Decimal Code	Binary Code	String Position
A	65	01000001	1
B	66	01000010	2
C	67	01000011	3
D	68	01000100	4
0	48	00110000	5
1	49	00110001	6
2	50	00110010	7
3	51	00110011	8

Using 1's and 0's, the following diagram specifies the settings of flags 1 through 64 from left to right, respectively; the 64 bits (flags) have been grouped into eight characters (or bytes).

```
(01000001)   (01000010)   (01000011)   (01000100)
    1            2            3            4

(00110000)   (00110001)   (00110010)   (00110011)
    5            6            7            8
```

You can use either of the following programs to set the flags shown in the above diagram:

```
10  SFLAG "ABCD0123"
20  END
```

or

```
10  FlagString8$=CHR$(65)&CHR$(66)&CHR$(67)&CHR$(68)
20  FlagString8$=FlagString8$&CHR$(48)&CHR$(49)&CHR$(50)&CHR$(51)
30  SFLAG FlagString8$
40  END
```

SFLAG truncates strings longer than eight characters at the eighth character. Strings shorter than eight characters are filled with "null" control characters, CHR$(0); consequently, all flags after the last one are set to 0 (cleared).

Passing Flags to Chained Programs If you desire to pass flags from program to program as you chain them, then you will need to use the COM statement, because the CHAIN statement clears **all** flags. The following segment of a program is an example of how flags can be passed to the next program when chaining.

```
100  COM SysFlags$[8]
.
.
.
790  SysFlags$=FLAG$
800  CHAIN "NextProg" ! Must have identical COM structure.
```

This passes all 64 flags to the chained program through the "common" variable storage area. Note that the chained program **must** have a matching COM structure, or the existing COM will be destroyed by the new COM, and the new COM variables will be initialized implicitly: numeric variables will be set to 0, and string variables will be set to the null string.

Memory Management with Subprograms

The CALL and SCRATCHSUB statements allow for subprogram overlays to re-use the same memory space. This is useful for programs which are large enough that they won't all fit into memory at once, whether the programs themselves are too numerous and/or too large, or whether variables and arrays use enormous amounts of space.

The SCRATCHSUB statement allows you two options:

- You can specify the name of a single subprogram to be deleted from memory.

```
SCRATCHSUB "Sort"
```

- You can specify a subprogram and delete that subprogram and all subprograms in memory *from that point on.*

```
SCRATCHSUB "Control_valves" TO END
```

If the system tries to delete a subprogram which is not currently in memory, no error will be reported.

A subprogram can only be deleted if it is not currently "active.". This means that:

- A subprogram can not delete itself.
- A subprogram cannot delete the subprogram that called it. (Otherwise it wouldn't have any place to go when the SUBEND or SUBEXIT statement was encountered!)

Between the time that a subprogram is entered and the time it is exited, the Technical BASIC system keeps track of an "activation record" for that subprogram. Thus if the subprogram calls a subprogram which calls a subprogram, and so forth, then 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 eventually the program will need to return to the calling context.

Context Switching

6

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. Consequently there are many things, such as line numbers, line labels, and variables, which are "local" to programs and subprograms. On the other hand, there are several modes, flags, and so forth, that are "global" to programs and subprograms. This section shows what is local and what is global.

Global Declarations

Default lower bound of array dimensions	`OPTION BASE`[1]
Trigonometric modes	`DEG`, `RAD`, and `GRAD`
All working directory changes	`MASS STORAGE IS`
All file-create operations	`CREATE`

1 Since OPTION BASE is global, attempting to use different OPTION BASE statements in program and subprograms will produce errors.

System screen operations	`CRT IS, ON CURSOR, OFF CURSOR, FLIP`
System printer operations	`PRINTER IS, PRINT ALL, NORMAL`
All graphics operations	`PLOT, DRAW, PEN, PLOTTER IS` etc.
Error reporting	`DEFAULT ON, DEFAULT OFF`
System flags	`FLAG, SFLAG, CFLAG`
File buffers	`ASSIGN#`

Local Declarations

Error trapping	`ON ERROR GOTO/GOSUB, OFF ERROR`
Interface interrupts	`ON INTR...GOTO/GOSUB, OFF INTR`
Softkey interrupts	`ON KEY# .. GOTO/GOSUB, OFF KEY#`
Keystroke trapping	`ON KYBD GOTO/GOSUB, OFF KYBD`
Timeout interrupts	`ON TIMEOUT...GOTO/GOSUB, OFF TIMEOUT`
Timer interrupts	`ON TIMER#...GOTO/GOSUB, OFF TIMER#`

For further details on each statement, see the *Technical BASIC Language Reference*.

7 Error Handling

Most programs are initially subject to errors at run time, even if all the typographical/syntactical errors have been shaken out while entering the program into the computer. There are three general courses of action to take with respect to run-time errors:

1. Try to prevent the error from happening in the first place.
2. Once an error occurs, try to recover from it and continue execution.
3. Do nothing – let the program "roll over and die" if an error occurs.

The last alternative, which may seem frivolous at first glance, is certainly the easiest to implement. Furthermore, the friendly nature of the HP-UX Technical BASIC system makes this a feasible choice – if the person running the program is a programmer, or better yet is the person who wrote the program. Upon encountering a run-time error, the BASIC system pauses program execution and displays a message giving the error number[1] and the line in which the error happened. The operator/programmer can then examine the program in light of this information and fix things up.

On the other hand, if the person running the program did not write it, then the first two approaches above should be used. The program should attempt to prevent errors from happening in the first place, and when they do occur to recover gracefully and continue running.

1 A complete list of error numbers and definitions is provided in the back of the *HP-UX Technical BASIC Language Reference.*

Chapter Contents

This chapter discusses the following topics:

- How the BASIC system handles errors
- Anticipating operator errors
- Checking for boundary conditions
- Comparison errors
- Trapping errors
- Determining error numbers and location
- Displaying the normal system error message

How the System Handles Errors

The Technical BASIC system is designed to recognize a broad range of errors. For instance, if a program attempts to evaluate a mathematical operation whose results are not defined (such as division by zero), then the system will report an error condition. This section briefly describes how the system normally reports these errors. The subsequent sections describe how you can anticipate these errors and handle them from a program.

Errors in Keyboard Calculations

Errors encountered while you are trying to execute a command or evaluate an expression from the keyboard are trapped by the system. The default response of the system is to give you a warning. Here is an erroneous math operation:

```
1/0
```

Here is the system's **default response** to attempting the operation:

```
warning 2 OVERFLOW
 1.79769313486231e+308
```

The system attempts to tell you more about the error by showing that the result is larger than it can represent; the value shown is the largest number that the system can handle. This is known as the "default value" for the out-of-range result. (There are similar errors in the range of error number 1 through 8.)

If you now execute:

```
DEFAULT OFF
```

The system will not display the "default value" for the out-of-range math error. Here are the results of performing the preceding 1/0 calculation with DEFAULT OFF:

```
Error 2  OVERFLOW
```

The differences between DEFAULT ON and DEFAULT OFF while a program is running are much more significant. They are the topic of the subsequent paragraphs.

Run-Time Errors

Run-time errors occur while a program is being executed. There are three ways that the BASIC *system* can handle "out-of-range" math errors:

- Display a `warning`, give the default value, and continue the calculation with the default value. (This is the system's response with `DEFAULT ON`.)
- Halt execution and display an error report. (This is the system's response with `DEFAULT OFF`.)
- Not display the report, but pass it on to an "error handler" routine in the BASIC program.

The next section shows how to anticipate (and thereby avoid) most errors. The section after that shows how to trap errors (and optionally correct them) from a running program.

Anticipating Operator Errors

The programmer that writes a program (hopefully) knows exactly what the program is expected to do and what kinds of inputs make sense for each task. Given this viewpoint, there is a strong tendency not to take into account the possibility that other people using the program might **not** understand the range of valid inputs.

As a programmer who wants your programs to be reasonably reliable, you really have no choice but to assume that users can make mistakes that cause errors *every* time they have the opportunity to enter information. Thus, the goal is to make the program *reasonably* foolproof.

Boundary Conditions

A classic example of anticipating an operator error is the "division by zero" situation. For instance, suppose that 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 crashes 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  DISP "Miles traveled and total hours" @ INPUT Miles,Hours
110  IF Hours<=0 THEN 120 ELSE 160
120    BEEP
130    PRINT "Improper value entered for hours."
140    PRINT "Try again!"
150    GOTO 100
160  ! Input OK, so continue normally.
```

Consider another simple example of giving a user the choice of six colors for a 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:

```
100  CLEAR
110  DATA GREEN,BLUE,RED,YELLOW,PURPLE,PINK
120  DIM Colors$(6)[6]
130  FOR Indx=1 TO 6
140    READ Colors$(Indx)
150  NEXT Indx
160  FOR I=1 TO 6
170     PRINT USING "DD,X,K";I,Colors$(I)
180  NEXT I
190 Ask: DISP "Pick the number of a color" @ INPUT I
200  IF I>=1 AND I<=6 THEN Valid_Color
210     BEEP
220     DISP "Invalid answer -- ";
230     WAIT 1
240  GOTO Ask
250 Valid_color: ! Program continues here when input is OK.
```

The above example needs a little extra safeguarding. The input variable I should be declared to be an INTEGER, since the only valid inputs are 1, 2, 3, 4, 5, and 6. An answer like "You have picked the 3.14th color listed" does not make sense.

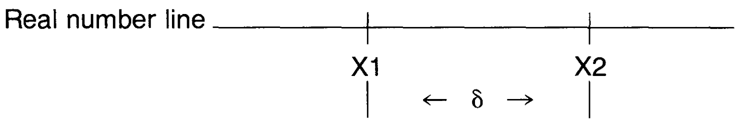
Here is an example that tests real number boundaries:

```
7000 AskFreq:  DISP "Enter the waveform's frequency (in KHz)"
7010           INPUT Frequency
7020           IF Frequency<=0 THEN AskFreq
7025           !
7030 AskAmpl:  DISP "Enter the amplitude (0-10 volts)"
7040           INPUT Amplitude
7050           IF Amplitude<0 OR Amplitude>10 THEN AskAmpl
7055           !
7060 AskDeg:   DISP "Enter the phase angle (in degrees)"
7070           INPUT Angle
7080           IF Angle<0 OR Angle>180 THEN AskDeg
```

REAL Numbers and Comparisons

A word of caution is in order about the use of the = comparison operator in conjunction with real numbers – numbers of type REAL and SHORT. 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 – even though it will be really close! What this means is that a program should not use the = operator for comparing real 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. However, if it is absolutely necessary to do an equality comparison with a pair of real numbers, then a second method must be used. This method 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 δ, we'll say that X1 and X2 are "equal" to each other for all practical purposes. The value of δ will depend upon the application, and must be chosen with care.

For an example, assume that we've picked a tolerance of 1E – 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
```

Trapping Errors

Despite your best efforts at screening the operator's inputs to avoid errors, errors will still occasionally happen. It is still possible to recover from run-time errors, provided you predict the places where errors are most likely to happen.

Setting Up the Error Branch

The ON ERROR command sets up a branch which will be initiated any time a recoverable error is encountered at run time. The branching action taken may be either GOTO or GOSUB. GOTO and GOSUB are purely local in scope – that is, they are active only within the context in which the ON ERROR is declared, not in any subprogram or user-defined function.

Here is a simplistic example of using the ON ERROR statement:

```
10  ON ERROR GOTO Trap
20  A=1/0
30  DISP "DONE" !  This and next line not executed.
40  STOP
50  !
60 Trap: DISP "ERROR: Division by 0"
70       END
```

7

Executing an ON ERROR statement directs the BASIC system **not** to report subsequent errors; instead, the system is to initiate a branch, GOTO or GOSUB, to the specified location in the program. The BASIC statements at that location can then handle the error.

When line 20 is executed, an error is detected and the system executes the `GOTO Trap` specified in the ON ERROR statement. For simplicity, the Trap routine merely prints the corresponding message and ends the program. (Note that lines 30 and 40 are not executed.)

Determining Error Number and Location

In the preceding example, it was assumed that the only error that could be produced was error 2 – the division by 0 in line 20. However, it is rarely the case that you know *which* error has happened or *where* it has happened. A more general error-trapping routine would determine which error happened and where it happened.

ERRN is a function which returns the error number which caused the branch to be taken. ERRN is a global function; it can be used in the main program or in any subprogram to determine the number of the most recent error. Here are a couple of simple examples:

```
100  DISP "Error number ";ERRN;" has occurred."
```

```
740  IF ERRN=18 THEN GOTO String_Error
```

ERRL is a function which is used to find the line in which the error was encountered. ERRL is a boolean function. The program passes it a line identifier (either line number or label), and the function returns either a 1 or a 0 – depending upon whether or not the specified identifier indicates the line which caused the error, respectively. ERRL is also a global function.

```
1140  IF ERRL(710) THEN DISP "The error occurred in line 710."
```

```
910   IF ERRL(Compute) THEN Fix_compute
```

Error Subroutines

The ON ERROR GOSUB statement sets up and enables a branch to the error service routine which will RETURN execution to the line *following* the one that caused the error.

```
100   Radical=B*B-4*A*C
110   Imaginary=0
120   ON ERROR GOSUB Esr
130   Partial=SQR(Radical)
140   IF Imaginary THEN Partial=SQR(Radical) ! Re-cal
150   OFF ERROR
        .
        .
        .
350 Esr: IF ERRN=10 THEN Imagin ELSE OtherError
360 Imagin: Imaginary=1             Set flag.
370         Radical=ABS(Radical) ! Make arg. positive
380       GOTO EndIf
390 OtherError: BEEP
400         BEEP
410         DISP "Unexpected Error (";ERRN;")"
420         PAUSE
430 EndIf: RETURN
```

Displaying the System Error Message

When you trap an error programmatically, you disable the system's normal error reporting mechanism. The system assumes that you want to handle errors yourself, which may include correcting the problem and then re-trying the operation. However, there are certain times when you do not (or cannot) fix the error. In some of these cases, you may want to report the error to the computer operator, who may just note the error or try to correct it.

The ERRM statement displays the "error message" that would have been reported by the system when the last error occurred. Here is an example of using this feature.

```
100  IF StillNotFixed THEN ERRM
110  RETURN
```

8 Debugging Programs

Naturally, the ideal way to develop a program is to design and implement it correctly the first time and not have to debug it at all. This is a worthwhile goal, and most programmers strive constantly to achieve it. Hopefully, the techniques discussed in preceding chapters will help you get a little closer to this goal.

However, no matter how good a programmer you are or how much time you have spent designing your programs, most programs will at one time or another be plagued with a "bug" – a bug is present whenever the program does not do what the user expects it to do.

You may usually think of a bug as something that generates an error condition, such as `ERROR 68 FILE TYPE`. However, a bug doesn't always inform you of its existence. In fact, the most insidious bugs cause your program to give a wrong answer *without* any indication that a bug even exists. This chapter deals with the methods available with Technical BASIC to diagnose problems in both logic and semantics.

The problem of debugging a program is distinct from the issues raised in the "Error Handling" chapter. That chapter was based on the premise that the programmer is *already* satisfied that the program works as it should, and that the *next* step is to make it as foolproof as possible. That assumption 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.

Chapter Contents

This chapter discusses the following topics:

- Whence cometh bugs?
- Methods of debugging programs
- Code walk-throughs
- Printing all program results
- Cross references
- Tracing program flow
- Setting breakpoints
- Checking variables' contents from the keyboard
- Continuing program execution.
- Single-stepping a program
- Software testing.

Whence Cometh Bugs?

As mentioned in the introduction of this chapter, a bug is present when the program does not do what the user reasonably expects it to do. Generally, this definition involves two main steps:

- Determining (or setting) the user's expectations.
- Making sure that the program actually does what is expected.

This chapter discusses getting your program to do what is expected. Determining and setting users' expectations are discussed in the chapter called "Communicating with the Operator."

The following two topics are discussed, with the primary focus on the second one:

- Methods of *designing* programs so that they do what you want them to do.
- Methods of *checking* to see that part(s) of a program are doing what you want.

A Model of the Software Development Process

In order to find places where bugs originate, let's take a brief look back at the steps of the software design process shown in the "Program Development" chapter.

1. Understand and describe the problem.
2. Outline a solution.
3. Design algorithms and data structures, and then refine.
4. Translate the data structure and algorithms into BASIC code.
5. Debug and test the program.
6. Document and support the program.

Note that most of these steps somehow involve either *the communication or translation of information.* For instance, step 1 involves translating the user's needs into a set of "requirements", while step 4 involves translating the algorithms data structures into a set of programming language statements.

This translation process is one of the largest sources of bugs. It is here that you should begin debugging programs, because many errors in the program are only manifestations of these problems.

1 For an excellent treatise on the origin and extermination of bugs, see *Software Reliability* by Glenford J. Meyers, John Wiley and Sons, New York, 1976.

Methods of Debugging Programs

Now that you have at least an inkling of where bugs originate, you are better prepared to find them in your programs. This section describes several methods of ridding your programs of these annoying little creatures.

Here are the general methods discussed in the remainder of this chapter:

- Algorithm and code walk-throughs
- Cross references
- Tracing program flow
- Setting breakpoints
- Examining variables' contents from the keyboard
- Single-stepping the program

Walk-Throughs

There are generally two times when you can walk through a program:

- Before it is coded.
- After it is coded.

In general, the *sooner* you find a bug, the *less* it costs to fix it.

Algorithm Walk-Throughs After developing an algorithm (and before coding it), you should walk through it. This walk-through is especially useful in checking whether you have properly translated the problem description into the outline and then into the algorithms and data structures.

You will perform the walk-through by *acting as if you were the computer* executing the algorithm on some actual data. At this point, you should walk through the algorithms with those programmers whose algorithms will be interacting with yours. It is also a good idea to include at least one programmer who is *not* involved with the project in this exercise.

You may also want to use specific test data (with known results) in this phase.

BASIC Code Walk-Throughs Once you have coded your program, you should perform the exercise of walking through it to verify again that it is going to do what you want it to do. This walk through checks to see whether you have correctly translated the algorithms and data structures into program code.

Printed Records of Debugging

When using the techniques presented in the remainder of this chapter, you will often find that you want to get printed records of what has happened. Normally, the cross-reference and tracing statements direct information to the current CRT IS device. However, you can use the PRINT ALL statement to direct the system to duplicate these messages on a printer, or you can specify another CRT IS device. For details of using printers and displays with your particular system, see the *Getting Started* manual for your HP-UX Technical BASIC system.

To return to sending the information only to the display, use the NORMAL statement.

Cross References

A cross reference is a list of this information:

- Where variables are used in the program.
- Where line numbers (and labels) are referenced by GOTO and GOSUB statements.

This section explains how to obtain and interpret cross references.

The XREF statement is programmable as well as executable from the keyboard. It provides a cross-reference table of program line numbers, line labels, and user-defined functions in the program (or subprogram) currently in memory.

Where Are Variables Used? XREF V displays a cross-reference table of all the variable and user-defined functions in the current program (or subprogram). It is very handy in finding such subtle errors as misspelled variable names.

Test the XREF V command out by entering the following program:

```
10  OPTION BASE 1
20  DIM SArray$(1)[5]
30  SArray$(1) = "Codes"
40  DISP "SArray$(1) = ";SAray$(1)
50  END
```

Next, execute this statement:

```
XREF V
```

The resulting display looks like this:

```
Variable     Dim1    Dim2    Max1    Type      References

SAray$         10              18    string       40
SArray$         1               5    string       20   30

...end of xrefv
```

The listing makes it easier to see that there are two variables, one of which is merely a misspelled version of the other.

Here is what information each column contains:

`Variable`	the name of the variable or user-defined function.
`Dim1`	the upper bound of the first subscript in an array variable (left blank if the variable is not an array).
`Dim2`	the upper bound of the second subscript in an array variable (also left blank if the variable is not an array, or is an array with only one subscript).
`Max1`	the maximum length of a string variable (left blank if the variable is not a string).
`Type`	INTEGER, REAL, SHORT, array, or string.
`References`	lines referencing the variable or user-defined function, including function definitions (DEF FN...), function value assignments (FN... = ...), and function calls (FN...).

Where Are Program Lines Referenced? XREF L generates an entry in the line cross-reference table whenever a line number or line label is referenced. To test the XREF L command, enter the following program:

```
10  X=10
20  Y=20
30  IF X=10 THEN GOTO 50
40  Total=X+Y
50  IF Total > 300 THEN  Finish
60  DISP "Total = ";Total
70  Finish: END
```

Next, execute this command:

```
XREF L
```

The resulting table looks like this on your display:

```
Line Cross Reference Table

  50 _____________ occurs on   30
  70 Finish:______ occurs on   50

... end of xrefl
```

Line numbers on the left of the display show a line that is referenced (such as with a GOTO or GOSUB). Line numbers on the right side of the display show where the reference occurred. In this example, a reference to line 50 occurs on line 30 (in the GOTO statement). A reference to line label Finish (line number 70) occurs on line 50 (also in a GOTO).

Program Traces

The Technical BASIC system provides means of tracing the following events:

- A branch in the linear flow, such as when a GOTO or GOSUB is executed.
- An assignment to a variable.
- All program flow (including flow of control from one line to the next) and all variable assignments.

Tracing Branches For this section, you will be tracing bugs in the following segment of code:

```
100  DIM Arg$[100],Result$[100]
110  INTEGER BeginPos,EndPos
120  !
130  Arg$="  Text  "
140  DISP "Arg$=(";Arg$;")   ","LEN=";LEN(Arg$)
150  GOSUB Trim
160  DISP "Result$=(";Result$;")   ","LEN=";LEN(Result$)
170  !
180  STOP
190  ! *******************************************
200  ! Given string in Arg$, this subroutine
210  ! trims leading and trailing blanks.
220  ! Trimmed string is returned in Result$.
230  ! *******************************************
240 Trim:
250    BeginPos=0
260    TrimFront:  BeginPos=BeginPos+1
270                IF BeginPos>LEN(Arg$) THEN Result$="" @ RETURN
280                IF Arg$[BeginPos,BeginPos]=" " THEN TrimFront
290                !
300    EndPos=LEN(Arg$)+1
310    TrimEnd:    EndPos=EndPos-1
320                IF Arg$[EndPos,EndPos]=" " THEN TrimEnd
330                !
340    Result$=Arg$[BeginPos,EndPos]
350    !
360 RETURN
```

Here are the results of running the program without tracing.

```
Arg$=(  Text  )      LEN= 8
Result$=(Text)       LEN= 4
```

Here are the results of executing a TRACE statement and then running the program.

```
Arg$=(  Text  )      LEN= 8
Trace line 150 to 240
Trace line 280 to 260
Trace line 280 to 260
Trace line 320 to 310
Trace line 320 to 310
Trace line 360 to 160
Result$=(Text)       LEN= 4
```

As you can see, **only** the branches (from otherwise linear program flow) are shown on the TRACE listing. Note also that the program's output also appears on the screen.

You can also use TRACE statements in a program to enable tracing for **only selected portions** of the program. For instance, insert these lines into the preceding program:

```
255  TRACE

285  NORMAL
```

Now execute:

```
NORMAL
```

to disable the TRACE enabled earlier.

Here are the corresponding results of running the program.

```
Arg$=(  Text  )      LEN= 8
Trace line 280 to 260
Trace line 280 to 260
Result$=(Text)       LEN= 4
```

Note that TRACE also shows when a subprogram is called. Here is a typical display:

```
Entering subprogram SUB_1a
```

The trace also shows when the subprogram is exited:

```
Leaving subprogram SUB_1a
```

However, note that TRACE in this case is not enabled while in the subprogram. To do that, you will need to store a TRACE statement in a line that will be executed when the subprogram is called.

When tracing user-defined functions, only the line number is shown, which is the same as with normal branches.

Tracing Variable Assignments You can use the TRACE VAR statement to display a message when a variable is assigned a value.

Using the preceding example program, trace the variable named BeginPos. (If you inserted the TRACE and NORMAL statements on lines 255 and 285, respectively, you may want to delete them now; if so, then you will also have to execute INIT before executing the TRACE VAR statement.)

```
TRACE VAR BeginPos
RUN
```

Here are the results:

```
Arg$=(  Text  )      LEN= 8
Trace line 250 BeginPos=0
Trace line 260 BeginPos=1
Trace line 260 BeginPos=2
Trace line 260 BeginPos=3
Result$=(Text)       LEN= 4
```

As the variable being traced is assigned values, the trace shows the line number where the assignment is made. For numeric variables, the trace also shows the value assigned to the variable. For string variables, the value assigned to the variable is **not** shown. For instance, here is a trace of a string variable. (Note that the tracing of the variable named BeginPos is disabled with the NORMAL statement.)

```
NORMAL
TRACE VAR Arg$
RUN

Tracing line 130 Arg$
Arg$=(  Text  )      LEN= 8
Result$=(Text)       LEN= 4
```

As with other TRACE statements, TRACE VAR can be executed from a program or from the keyboard.

If you need to trace variables in subprograms, you will need to put a TRACE VAR statement on a line of the subprogram that will be executed when the subprogram is called.

8

The TRACE VAR statement does **not** trace variables in user-defined functions.

Tracing All Flow and Variables When the TRACE ALL statement is executed, it causes the system to issue a message prior to executing *every* line, not just those where branches occur. This shows the order in which **all** statements were executed.

Here is a long, boring TRACE ALL of the example program shown in the preceding sections:

```
Trace line 100 to 110
Trace line 110 to 120
Trace line 120 to 130
Tracing line 130 Arg$
Trace line 130 to 140
Arg$=(  Text  )      LEN= 8
Trace line 140 to 150
Trace line 150 to 240
Trace line 240 to 250
Trace line 250 BeginPos=0
Trace line 250 to 260
Trace line 260 BeginPos=1
Trace line 260 to 270
Trace line 270 to 280
Trace line 280 to 260
Trace line 260 BeginPos=2
Trace line 260 to 270
Trace line 270 to 280
Trace line 280 to 260
Trace line 260 BeginPos=3
Trace line 260 to 270
Trace line 270 to 280
Trace line 280 to 290
Trace line 290 to 300
Trace line 300 BeginPos=9
Trace line 300 to 310
Trace line 310 BeginPos=8
Trace line 310 to 320
Trace line 320 to 310
Trace line 310 BeginPos=7
Trace line 310 to 320
Trace line 320 to 310
Trace line 310 BeginPos=6
Trace line 310 to 320
Trace line 320 to 330
Trace line 330 to 340
```

8

If you have a large program, you will probably not want to perform a TRACE ALL of the whole thing. You can insert a program line containing TRACE ALL at the beginning of where you want to enable tracing, and insert a line containing NORMAL where you want to disable it.

If you need to enable TRACE ALL in subprograms, you will need to put a TRACE ALL statement on a line of the subprogram that will be executed when the subprogram is called.

The TRACE ALL statement does **not** trace either line numbers or variables in user-defined functions.

Returning to Normal Execution NORMAL cancels the effects of any active TRACE, TRACE VAR, or TRACE ALL statements. It also disables PRINT ALL mode. The NORMAL statement may be executed either from the program or from the keyboard.

Pausing Program Execution

On most consoles and terminals, you can pause (temporarily halt) program execution. You can use either the **Break** key, or **CTRL-C**. This is a rather crude way of debugging, since it does not allow you to determine which program line will be executed next. The next section describes a better way to debug using breakpoints.

Setting Breakpoints with PAUSE

A breakpoint is a point in the program where execution is halted. With Technical BASIC, you can use the PAUSE statement to set a breakpoint. Let's use the example program from the last section. Here is another listing of the program for convenience:

```
100  DIM Arg$[100],Result$[100]
110  INTEGER BeginPos,EndPos
120  !
130  Arg$="  Text  "
140  DISP "Arg$=(";Arg$;")    ","LEN=";LEN(Arg$)
150  GOSUB Trim
160  DISP "Result$=(";Result$;")    ","LEN=";LEN(Result$)
170  !
180  STOP
```

```
190  ! ********************************************
200  ! Given string in Arg$, this subroutine
210  ! trims leading and trailing blanks.
220  ! Trimmed string is returned in Result$.
230  ! ********************************************
240 Trim:
250    BeginPos=0
260    TrimFront:  BeginPos=BeginPos+1
270                IF BeginPos>LEN(Arg$) THEN Result$="" @ RETURN
280                IF Arg$[BeginPos,BeginPos]=" " THEN TrimFront
290                !
300    EndPos=LEN(Arg$)+1
310    TrimEnd:    EndPos=EndPos-1
320                IF Arg$[EndPos,EndPos]=" " THEN TrimEnd
330                !
340    Result$=Arg$[BeginPos,EndPos]
350    !
360 RETURN
```

(If you still have a TRACE on line 255 and NORMAL on line 285 from a previous example, then you may want to delete these lines now.)

Insert this statement into the program:

```
245  PAUSE @ DISP "Breakpoint 1"
```

Now run the program. It will display the message `Breakpoint 1` and then pause.

You can do any of several things at this point:

- Start tracing variables or program flow.
- Examine or change the value of variables.
- Execute statements or commands.
- Resume execution by executing the CONT (continue) command.
- Single-step the program.

Tracing operations were explained in the preceding section. Subsequent sections explain the latter four topics.

Accessing Variables from the Keyboard

One of the pleasing characteristics of Technical BASIC system is that you can access variables from the keyboard any time that it is in the "paused" state. You can also change variable's values from the keyboard. Note, however, that you cannot access another variables in another context; for instance, you cannot access the main program's variables while in a sub-program.

You can **determine the current value of a variable** (in the current context) by typing its name on a blank line and then pressing the carriage return key:

```
BeginPos
```

The system responds with:

```
0
```

You can also **perform calculations**, such as:

```
LEN(Arg$)
```

The system responds:

```
8
```

You can also **assign a new value to a variable**. For example, to assign a value of 5 to the variable named BeginPos, execute:

```
LET BeginPos=5
```

Now examine the variable by typing:

```
BeginPos
```

It will return:

```
5
```

8

If you had typed this instead:

```
BeginPos=5
```

You would have gotten this response:

```
0
```

because BeginPos = 5 is a boolean expression whose value is "false" (since BeginPos is NOT equal to 5); the system represents a false condition with a 0, while a boolean true is represented by a 1.

Note also that you can create a variable from the keyboard by assigning it a value:

```
LET NewVariable=6.023
```

You can use this variable in subsequent keyboard operations.

```
NewVariable-1
 5.023
NewVariable>0
 1
```

Executing Commands and Statements

You can also issue commands while a program is paused. For instance, you can examine the catalog of a directory, list the program to a printer, and turn the graphics or alpha displays on and off.

Continuing Program Execution

When the program is in a paused state, you can continue program execution with the CONT command. Program execution then resumes normally, or in the trace mode that was in effect at the time the program was paused.

Preceding paragraphs declared that you can execute nearly all commands from the keyboard while a program is paused. You can also add, modify, or delete program lines, or attempt to alter the control structures of the program; however, the program **cannot be continued** after such modifications. You will have to pre-run[1] the program (using INIT) or execute RUN.

Single-Stepping a Program

One of the most powerful debugging tools available is the capability of single-stepping a program – executing it one line at a time. This process allows you to access variables before or after each line of a program is executed.

1 Pre-run is described in the "Program Structure and Flow" chapter.

Single-stepping is performed with the SINGLESTEP command. There are two prerequisites to using this command:

- The program must have been "pre-run"[1] by executing INIT or RUN.
- The program must be in the paused state.

Type in the following example, execute an INIT command, and then begin single-stepping by executing the SINGLESTEP command three or four times. (If you typed in the preceding example program, then you will probably want to store it now, because you will be using it again in the next section.)

```
100  OPTION BASE 1
110  REAL Array(5),ArraySum
120  INTEGER Indx
130  !
140  ! Enter five numbers, and calculate their sum
150  ArraySum=0
160  FOR Indx=1 TO 5
170     DISP "Enter numeric value #";Indx
180     INPUT Array(Indx)
190     ArraySum=ArraySum+Array(Indx)
200  NEXT Indx
210  !
220  ! Display input data and sum.
230  DISP "Array:"
240  MAT PRINT Array;
250  PRINT "Sum of array elements:",ArraySum
260  !
270  END
```

Notice that it is difficult to tell which program line is being executed without using the TRACE ALL command.

As you can see from the TRACE ALL results, the SINGLESTEP command executes a program line and then increments the program counter to the next program line. Thus, SINGLESTEP steps through *every* program line, including those containing non-executed statements like OPTION BASE, REAL, INTEGER, and ! comments (which are handled during pre-run and cause no action during program execution).

As you single-step through the program, you can check variables' contents to see how they change. You can also change them as desired to create and test special conditions.

If the program is in an INPUT or LINPUT statement, then SINGLESTEP is sufficient to terminate the operation. After executing SINGLESTEP on one of these input statements, you must first enter the expected data from the keyboard and then terminate it with a carriage return. Executing a subsequent SINGLESTEP then will execute the following line.

Software Testing

In general, testing a program involves verifying that it does work without errors. So in order to test a program, you will ordinarily use it across its normal range of conditions. In addition, you will often want to ensure that it will not crash when asked by a user to operate outside this range.

There are many methods of testing; they range from testing segments individually to testing the entire program as a whole. Although the subject of software testing is extensive, it is mentioned here to make sure that you are aware of the need for testing and to help you realize that there are many texts available that describe methodical approaches to testing.

Despite the gamut of available testing methods, here are some approaches that are common to most methods:

- It is difficult to thoroughly test your own programs. *It is best to have someone else test code that you have written.*
- Question assumptions. For instance, you may assume falsely that the user will not input string data when numeric data is expected.
- Determine boundary conditions for valid inputs, and test each one. For instance, if you are expecting a string of up to 20 characters, test your software for strings with lengths 0 and 20 (maybe even 21).
- Check every local branch in the code to make sure that each will be executed properly in all directions. Then globally make a test case for each unique path through the program.

- Check to see if there are any sensitivities to any particular data patterns.

The "Error Handling" and "Communicating with the Operator" chapters also discuss anticipating and handling erroneous inputs.

Testing the Example Program

Using the program presented earlier in this chapter, you can get a feel for implementing some of the suggestions shown above. Here is the program again.

```
100  DIM Arg$[100],Result$[100]
110  INTEGER BeginPos,EndPos
 .
 .
 .
180  STOP
190  ! ********************************************
200  ! Given string in Arg$, this subroutine
210  ! trims leading and trailing blanks.
220  ! Trimmed string is returned in Result$.
230  ! ********************************************
240 Trim:
250    BeginPos=0
260    TrimFront:  BeginPos=BeginPos+1
270                IF BeginPos>LEN(Arg$) THEN Result$="" @ RETURN
280                IF Arg$[BeginPos,BeginPos]=" " THEN TrimFront
290                !
300    EndPos=LEN(Arg$)+1
310    TrimEnd:    EndPos=EndPos-1
320                IF Arg$[EndPos,EndPos]=" " THEN TrimEnd
330                !
340    Result$=Arg$[BeginPos,EndPos]
350    !
360 RETURN
```

Since you didn't write this program, you qualify as a candidate for testing it.

Some **assumptions** that you may question are as follows:

- Will the input always be less than 100 characters?
- Is it acceptable to leave leading or trailing, non-printing control characters, such as CHR$(0), in the string? or is the program to remove them before removing the spaces?

- Is a string of length 0 acceptable? or should it generate an error message?

Boundary conditions for the routine are strings of length 0, 100, and 101.

There are three, two-way branches in the program. From studying the **permutations of possible branch combinations**, there are six unique, valid paths through the code (some of the possible paths are identical[1]). Here are the cases that test these paths:

1. Null Arg$ (LEN = 0) with no leading or trailing spaces.
2. Null Arg$ with leading and trailing spaces.
3. Non-null Arg$ with leading spaces.
4. Non-null Arg$ with no leading or trailing spaces.
5. Non-null Arg$ with trailing spaces.
6. Non-null Arg$ with both leading and trailing spaces.

There seem to be no sensitivities to particular data patterns. However, note that the case of the non-printing control character embedded in leading or trailing spaces may fit into this category.

This listing shows testing the routine with the five cases shown above.

1 Some identical combinations are: 1. Only leading spaces and null Arg$ (LEN = 0) or Arg$ is all spaces; 2. Only trailing spaces and null Arg$; 3. Both leading and trailing spaces and null Arg$.

```
100  DIM Arg$[100],Result$[100]
110  INTEGER BeginPos,EndPos
120  !
121  Arg$=""
122  DISP "Arg$=(";Arg$;")   ","LEN=";LEN(Arg$)
123  GOSUB Trim
124  DISP "Result$=(";Result$;")   ","LEN=";LEN(Result$)
125  !
126  Arg$="         "
127  DISP "Arg$=(";Arg$;")   ","LEN=";LEN(Arg$)
128  GOSUB Trim
129  DISP "Result$=(";Result$;")   ","LEN=";LEN(Result$)
130  !
131  Arg$="   !#*1?"
132  DISP "Arg$=(";Arg$;")   ","LEN=";LEN(Arg$)
133  GOSUB Trim
134  DISP "Result$=(";Result$;")   ","LEN=";LEN(Result$)
135  !
136  Arg$="::;;<<==>>??"
137  DISP "Arg$=(";Arg$;")   ","LEN=";LEN(Arg$)
138  GOSUB Trim
139  DISP "Result$=(";Result$;")   ","LEN=";LEN(Result$)
140  !
141  Arg$="@A^_`a{}      "
142  DISP "Arg$=(";Arg$;")   ","LEN=";LEN(Arg$)
143  GOSUB Trim
144  DISP "Result$=(";Result$;")   ","LEN=";LEN(Result$)
145  !
146  Arg$="  Two Words "
147  DISP "Arg$=(";Arg$;")   ","LEN=";LEN(Arg$)
148  GOSUB Trim
149  DISP "Result$=(";Result$;")   ","LEN=";LEN(Result$)
150  !
180  STOP
190  ! ********************************************
200  ! Given string in Arg$, this subroutine
210  ! trims leading and trailing blanks.
220  ! Trimmed string is returned in Result$.
230  ! ********************************************
240 Trim:
250    BeginPos=0
260    TrimFront:  BeginPos=BeginPos+1
270                IF BeginPos>LEN(Arg$) THEN Result$="" @ RETURN
280                IF Arg$[BeginPos,BeginPos]=" " THEN TrimFront
290                !
```

8

```
300    EndPos=LEN(Arg$)+1
310    TrimEnd:    EndPos=EndPos-1
320                IF Arg$[EndPos,EndPos]=" " THEN TrimEnd
330                !
340    Result$=Arg$[BeginPos,EndPos]
350    !
360 RETURN
```

Here are the results of running this test program (without tracing).

```
Arg$=()              LEN= 0
Result$=()           LEN= 0
Arg$=(        )      LEN= 8
Result$=()           LEN= 0
Arg$=(   !#*1?)      LEN= 8
Result$=(!#*1?)      LEN= 5
Arg$=(::;;<<==>>??)                 LEN= 12
Result$=(::;;<<==>>??)              LEN= 12
Arg$=(@A^_`a{}    )                 LEN= 12
Result$=(@A^_`a{})                  LEN= 8
Arg$=(  Two Words )                 LEN= 12
Result$=(Two Words)                 LEN= 9
```

Note that several different characters were used in several of the tests. This may help show that the display device has no data sensitivities. Also note that only the last test tried using spaces embedded in the argument. This illustrates that *testing almost never stops.*

9 Communicating with the Operator

Have you ever been confused by the question posed by a program? Have you ever wondered which button to press next? Have you ever gotten a cryptic error message, or lost some important or irreproducible data? If you answered "yes" to any of these questions, then you know some of the frustrations of using a poorly designed computer/human interface[1].

As a programmer, you are on the other side of this interface. You have the responsibility of designing a program that others can use and, more importantly, will want to use. How will you ask questions? What assumptions are you going to make? How much time will you spend making your program easy to use? The time and effort you invest could mean the difference between a popular piece of software and one that everyone avoids like the plague.

Chapter Contents

This chapter describes the system features available for communicating with the computer operator. It contains these topics:

- An overview of the elements in a human interface
- General suggestions for improving computer/human communications
- Sending audio messages
- Sending alphanumeric messages
- Accepting input from the softkeys
- Accepting information from the keyboard

1 A computer/human interface, or simply human interface, is informally defined to be the means by which the computer operator interacts with the computer. This interface includes hardware, software, and information.

Overview

In order to design an effective human interface for a program, you need to take a closer look at the operator/program communication process.

A Simplified Model

Although the human interface involves many aspects of the flow of information between computer to operator, here is a simplistic model of the communication process at this interface:

1. The **computer prompts** the operator for information.
2. The **operator receives** the message.
3. The **operator thinks** about the question, and then **formulates a response**.
4. The **operator makes the response**.
5. The **computer accepts the information** entered by the operator.

Communications Devices These steps in the communication process are generally carried out by the following physical devices:

- **Computer output devices:** alphanumeric and graphics displays, beepers and voice-synthesis devices
- **Human input devices:** eyes, ears, sense of touch
- **Human information processor:** brain
- **Human output devices:** fingers and hands, voice
- **Computer input devices:** keyboards, graphics input devices (such as a knob, mouse, touch screen, and digitizer stylus), and voice-recognition devices

Other Factors Along with these output and input devices, there are some other factors that affect the communication of information.

- Method of presenting the information (terminology, page layout, etc.)
- Placement of the preceding output and input devices

- Operator's past experience and present mental state
- Various other human factors

Importance of the Human Interface

In general, the most important function of a computer is to manipulate data. Although the computer can receive data from other computers and devices, it is probably more common that it gets data from a computer operator.

If you are the only person that uses a program you've written, then that program may not need a quality human interface. This normal requirement is eliminated because you know exactly what data the program needs, when it is needed, and how to enter it into the computer. However, if a program is used by other people, then the demands for a good human interface rise greatly – especially if they have different backgrounds. When the intended users do not understand computers, your program must be very skillfully written so that it does not confuse or intimidate the operator or make great demands on their computer expertise.

This part of the process of using software is one of the most error-prone, because it involves the subtly complex process of human communications. And the problem is further compounded because the humans are separated by space and time, as well as restricted to communicating with limited means – usually only visual computer prompts and manual human input.

General Design Suggestions

Good human interfaces don't just happen; they require effort, logical thinking, and thorough testing. In many programs, at least 25% of the code is dedicated to the human interface. And it is not unusual to use 60% of a good program for explanatory messages, operator interaction, error trapping, and so forth. Obviously, these estimates depend upon many factors, such as the task being performed and the intended operators; however, they do show that a significant portion of the program design effort should be devoted to the human interface.

Here is a brief list of general suggestions for developing an effective human interface.

- Ask simple, definite questions or prompts
- Present the questions and prompts in a natural, logical order
- Limit the set of alternative answers, if possible
- Supply a default answer, if possible
- Provide a chance for the operator to verify choice(s) picked or information entered
- Trap invalid operator responses, and give them another chance

Sending Messages to the Operator

The information you can send to the operator generally fits into these two categories:

- Descriptions of what the program is currently doing (or what mode it is currently operating in)
- Descriptions of what the user is expected to do

These "status reports" and "prompts" for information, respectively, may be made in one of these ways:

- With words (text)
- With pictures (graphics)
- With sound (auditory messages)

Let's deal with sound first, because of the simplicity of the related Technical BASIC features, and then with text. Graphics are covered in the "Graphics" chapter.

Sending Audio Messages

It would be a real attention getter to have the computer synthesize a pleasant voice that says: "Please don't touch the keyboard right now." However, sometimes a simple warning "beep" is enough to give the same message.

Generic Beeps With some terminals and consoles[1], the only audio message available is the "bell" sound. This BASIC statement directs the terminal to make the bell sound.

```
BEEP
```

You can also get the same response by "displaying" the `bell` control character:

```
DISP CHR$(7)
```

This method works well when the operator probably knows what he is doing is wrong, but just needs a gentle reminder.

Varying Tones On other terminals and consoles there is a tone generator, which you can use to produce sounds of varying frequency and duration. Execute this statement to see if your machine has these capabilities.

```
BEEP 10,10 @ BEEP 20,20
```

If you get two different pitches, then your hardware has these capabilities.

The first parameter in the BEEP statement controls the *frequency*, while the second controls the *duration*.

```
BEEP Frequency,Duration
```

The range of the frequency and duration parameters are given in the *Specifics* appendix for your system.

Displaying Messages on the Alpha Screen

The mechanics of using alpha displays with HP-UX Technical BASIC systems vary from system to system. This section contains some general information about using displays with all systems. Consequently, you may want to refer to the *Getting Started* manual for your Technical BASIC system as you read this section.

Using printers to display information is described in a subsequent section.

1 The tones that your terminal or console can generate are listed in the *Implementation Specifics* appendix to the *HP-UX Technical BASIC Language Reference.*

The Essence of Displaying Messages Giving instructions to the operator can be condensed into these basic steps:

1. Clear the display of any irrelevant information.
2. Make sure that the display device is operating in the proper mode (for instance, not in insert mode)
3. Use as much of the display as necessary to give unambiguous, understandable instructions.

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 short message.

Please realize that those days are over. Take a typical HP-UX system as an example. The standard machine is shipped with over a half-million characters of memory, and there is no significant restriction on program size. Neither is there any real restriction due to the display size, since most HP displays supported on Technical BASIC usually have at least 20 lines of 80 characters each visible at all times. It is generally false economy to display tiny, cryptic messages.

Which Device Is the Display Screen? Statements that display text (like DISP and CAT) send the characters to the current CRT IS device. Normally this device will be the screen on which characters appear as you type[1].

You can see which screen is the CRT IS device by executing the following statement:

```
DISP "This is the current CRT IS device."
```

1 If you see no characters on your screen as you type at the keyboard, press the carriage-return key, then type in an ALPHA command, and execute the command by pressing the carriage-return key again. Executing this statement turns on the alpha display.

The display (or printer) on which the message appears is currently the CRT IS device. Normally it will be your terminal or console screen. However, if these characters are currently being sent to a printer (or file), then you can specify that your console is now to be the display device by executing this statement:

```
CRT IS 1
```

The numeric parameter 1 specifies the screen's *device selector.*

You can also specify that a file is to be the CRT IS device. If a file does not exist, then you can create one for this purpose. Then assign a *file selector* to the file, and specify this file number as the CRT IS device.

```
CREATE "CRTISFile",1
ASSIGN 11 TO "CRTISFile"
CRT IS 11
```

The CREATE statement creates a `text/data` file[1] in the current working directory. The ASSIGN statement assigns a file selector of 11 to the file named CRTISFile. Since no directory path was specified, the file was assumed to be in the current working directory. If there is no file named CRTISFile in that directory, then the system automatically creates it.

Subsequent information that would normally be sent to the screen (such as output of CAT, LIST) will be sent to this file. If the file already exists and has information in it, then the subsequent information is appended to the file.

You can also specify a **screenwidth** in the CRT IS statement. For instance:

```
CRT IS 1,65
```

BASIC will subsequently allow the DISP statement to display only 65 columns of text on the screen. (Other methods of writing to the screen are not affected, however.)

1 You can read files of this type from BASIC by using ASSIGN to open the file and the reading lines of text with ENTER statement. For example, see "Using text/data Files" in the "Data Storage and Retrieval" chapter.

Now that you have seen how to determine which display you will be using, and how to specify another, the next step is to find out what you can do with it.

Determining Display Capabilities An inherent requirement of using the steps above is knowing (or determining) the display device's capabilities. If you don't know, for instance, the width of an alphanumeric display screen, then you might try to put more than one line of text on a display line.

Here are some relevant questions you might ask about a display device's attributes and capabilities:

- What is the screen's width (number of columns) and height (number of rows)?
- What characters can it display, including enhancements (such as half-bright and underlining)?
- Can you position its cursor? (The cursor is a pointer that indicates the location at which the next character will be displayed.)
- Does it have special insert or delete modes or operations?
- Can you scroll the text on the display?

There are several **approaches** that you can take to determine a display device's capabilities:

- Read the display device's documentation.
- Observe its operation.
- Have the program determine them.

Using the **first approach**, you can read a display device's documentation, which is usually shipped with the device. For instance, if you are using an Integral Personal Computer, then you can read its installation and operating manuals. A list of the characters it can display, along with operating modes and escape code sequences it implements, is provided in the *Implementation Specifics* appendix shipped with the Integral HP-UX Technical BASIC system.

Multi-user systems, such as Series 500 HP-UX, are capable of supporting several different terminals at one time. In such case, you will either need to read the documentation for each terminal to determine its capabilities, or read the `/etc/termcap` ("terminal capabilities") file and decipher the codes. The *Getting Started* manual for your particular HP-UX Technical BASIC system describes the terminals that are supported on your system.

Using the **second approach**, you could begin displaying some character codes on the screen and observe the results. You should eventually do this anyway to get a feel for what is pleasing to the eye and effectively conveys the desired information.

The following example shows an application of the **third approach**: determining display width with a BASIC program[1].

```
100  DIM Lines$[170]
110  Lines$=RPT$(" ",40)&"40"&RPT$(" ",8)&"50" ! String pos 41,51.
120  Lines$=Lines$&RPT$(" ",28)&"80" !          String pos 81.
130  Lines$=Lines$&RPT$(" ",78)&"160" !         String pos 161.
140  !
150  CLEAR !        Clear the screen.
160  ALPHA 1,1 !    Home the cursor (Row 1, Column 1).
170  AWRIT Lines$ ! Write line (excess will "wrap" to next line).
180  !
190  ALPHA 2,1 !    Move cursor to start of second line.
200  AREAD Lines$ ! Read characters (which will show width).
210  ScreenWidth=VAL(Lines$) !  Convert string to number.
220  DISP "Width of screen=";ScreenWidth;"characters."
230  !
240  END
```

1 The ALPHA and AREAD statements are not implemented on some terminals. Refer to the *Implementation Specifics* for your particular HP-UX Technical BASIC system to see whether they are implemented on the console or terminal you are using.

Here are the results of running the program on an 80-column screen.

```
                                          40        50
80
160
Width of screen= 80 characters.
```

The program creates a string (Lines$) that is longer than any line that any screen can display. It places characters such as "40", "50", and so forth at string locations 41, 51, and so forth, respectively. The ALPHA statement positions the cursor at column 1 of row 1. The AWRIT statement writes this string into screen memory beginning at the current cursor location. Since the length of the string is greater than a screen width, some of the characters will be placed ("wrapped") onto subsequent row(s) of the screen. The AREAD statement then reads the number on the second row, which represents the width of the screen. The VAL function converts the string read by AREAD into a numeric value, which is assigned to the numeric variable named ScreenWidth. You can use an analogous technique to determine the number of lines (rows) on the screen.

Although this is a way for the program to determine the screen's width, it may not be the most reassuring thing for a program's user to see as he begins using the program.

An **alternate method** of programmatically determining screen width might be as follows: set up a table that lists each type of terminal's capabilities; have the program ask the user to identify the product number of the screen device; access the entry in the table that describes that device's capabilities; set up the communication model for that terminal based on the device's capabilities.

Clearing the Screen It is confusing to the operator (and embarassing to the programmer) when two or more displays combine in an unplanned manner. The culprits are often "left-over" alpha and graphics.

To completely erase the alpha display, use this statement:

```
CLEAR
```

It moves the cursor to its "home" position (upper, left corner), scrolling the text if necessary, and clears all characters from the display.

To completely clear left-over graphics, execute GCLEAR. Note that alpha and graphics are displayed separately on some consoles and terminals, but are displayed simultaneously on others.

Turning Off Unwanted Modes As another example, suppose that the previous user left the cursor in the middle of a screen of text with "insert mode" left on. If a subsequent program attempts to display new text without turning off insert mode and clearing the screen, then the result may be a chaotic screen.

The DISP statement does not provide a high-level method for getting the display out of modes like "insert character." Those modes are controlled by sending an escape sequence to the display. In this case, you will need to cancel the insert mode (return to not inserting characters before current cursor position). Here is a simple example:

```
440  CancelInsert$=CHR$(27)&"R"
450  DISP CancelInsert$;
```

The PRINTALL statement directs the system to print all information that is sent to the display screen; the information is printed on the current PRINTER IS device. You can cancel this mode by executing a NORMAL statement.

Positioning the Cursor Whenever you execute a statement that displays characters on the screen, these characters are displayed beginning at the current *cursor location*. For instance, one of the preceding examples showed a method of moving the cursor. Here is a similar example:

```
100  DIM Chars$[170]
110  Chars$="Cursor location."
120  !
130  CLEAR !        Clear screen, and "home" cursor (row 1, column 1).
140  !
140  DISP Chars$ !  Display beginning at cursor location.
150  DISP Chars$
170  !
180  ALPHA 3,20 !   Move cursor to line 3, column 20.
190  AWRIT "Cursor location doesn't change."
200  AWRIT "With AWRIT, loc"
210  !
220  END
```

Here are the program's results:

```
Cursor location.
Cursor location.
                   With AWRIT, loc doesn't change.
```

The CLEAR statement clears the display and sets the cursor location to row 1 and column 1. The subsequent DISP statement displays characters beginning at this location. As the DISP statement finishes, it automatically moves the cursor to the next line by sending an "end-of-line" (EOL) sequence: a carriage-return control character followed by a line-feed control character.

The cursor is then moved to column 20 of row 3 with the ALPHA statement. The AWRIT statement then writes the specified characters on the display. AWRIT is different from DISP in that it does **not** update the cursor location, as shown by the second AWRIT statement beginning at the same location (3,20) and overwriting characters written by the first one.

Determining the Cursor's Location If you are not sure where the cursor is, you can determine its location by using the CURSROW and CURSCOL functions.

- CURSROW returns the row.
- CURSCOL returns the column.

You can use these functions just as you would other numeric system functions. Here is an example of using them in a program.

```
100  Star$="*"
110  !
120  CLEAR
130  FOR RowNumber=1 TO 16 STEP 3
140    Col_=RND *60 !               Random column.
150    ALPHA RowNumber,Col_ !       Move cursor.
160    AWRIT Star$ !                Display the "*".
170    Row_=CURSROW !               Determine row.
180    Col_=CURSCOL !               Determine column.
190    ALPHA ,CURSCOL +3 !          Move cursor (relative).
200    DISP "(";Row_;",";Col_;")" ! Show row and column.
210  NEXT RowNumber
220    !
230  END
```

Here are typical results of running the program:

```
                                  * ( 1 , 43 )


                        * ( 4 , 33 )


                            * ( 7 , 37 )


                           * ( 10 , 36 )


     * ( 13 , 14 )


* ( 16 , 7 )
```

Turning the Cursor On and Off The cursor is the screen location at which subsequently typed or displayed characters will begin appearing. Normally the cursor's location is indicated by an inverse-video block or a blinking underline character.

To disable the visual cursor, execute this statement[1]:

```
OFF CURSOR
```

To re-enable the visual cursor, execute:

```
ON CURSOR
```

Displaying Blank Lines If the cursor position is at the start of a blank line when DISP is executed, that line remains blank. However, if there is text on that line, the text remains. This behavior is due to the fact that a DISP statement with no parameters simply sends an end-of-line sequence, which is a different operation than writing a line of blank characters – ASCII spaces, or CHR$(32). This is not to say that it is "wrong" to use DISP with no parameters. It just means that you cannot guarantee the output of a blank line by using DISP 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 to use the TAB function. Here is a sequence that prints three blank lines:

```
100  ScreenWidth=80 !  This may vary for your display device.
110  DISP TAB(ScreenWidth)
120  DISP TAB(ScreenWidth)
130  DISP TAB(ScreenWidth)
```

Before getting into greater detail about formatting information that you sent to the display, let's take a look at what capabilities you have for sending information to printers.

1 This feature is not implemented on some consoles and terminals.

Printers

The mechanics of using printers with HP-UX Technical BASIC systems vary from system to system. This section contains some general information about using printers with all systems. Consequently, you may want to refer to the *Getting Started* manual for your Technical BASIC system as you read this section.

The PRINT statement sends information to a printer in the same fashion as the DISP statement sends information to a screen display. The device specified in the last PRINTER IS statement, or the default system printer[1], receives PRINT statements' output[2].

To see which printer is the current PRINTER IS device, execute this statement:

```
PRINT "This is sent to the PRINTER IS device."
```

You can also specify that another device is to be the system printer. Here is an example of creating a file and then specifying that the file is to be the system printer.

```
CREATE "PRTISFile",1
ASSIGN 11 TO "PRTISFile"
PRINTER IS 11
```

The CREATE statement creates a text/data file in the current working directory. The ASSIGN statement assigns a *file selector* of 11 to the file named `PRTISFile`. Since no directory path was specified, the file was assumed to be in the current working directory. If there is no file named `PRTISFile` in that directory, then the system automatically creates it.

1 With single-user Integral HP-UX systems, the default system printer is the built-in printer. With other single-user and multi-user HP-UX systems, the default system printer is the display screen.

2 Multi-user HP-UX systems use intermediate files to receive the output of PRINT statements, which are then "spooled" to the printer.

There are times when you want to have printed records of what has been displayed on the screen. The PRINT ALL statement directs the system to print a copy of whatever information is sent to the display on the current PRINTER IS device.

If you have not been operating in PRINT ALL mode, but you find that you need to get a printed version of what is currently on the screen, you can use the DUMP ALPHA[1] statement to send a copy to the current PRINTER IS device.

A Typical Printer's Character Set Most ASCII characters are printed on an external printer much like they appear on the display screen[2]. However, depending on your printer, there will be exceptions. Several printers will also support an alternate character set; this alternate set is often a foriegn 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. This section describes typical characters that printers can print and use as control information.

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.

Printer's Response	Control Character's ASCII Value	Keyboard Character
ring printer's bell	7	CTRL-G
backspace one character	8	CTRL-H
horizontal tab	9	CTRL-I
line-feed	10	CTRL-J
form-feed	12	CTRL-L
carriage-return	13	CTRL-M

1 DUMP ALPHA requires that the printer is capable of displaying graphics. Note also that it is not implemented on some terminals. For further details, see the *Implementation Specifics* appendix to the *HP-UX Technical BASIC Language Reference* for your particular HP-UX Technical BASIC system.

2 A list of the characters available on a particular printer is given in the documentation sent with that printer.

One way to send control characters to the printer is with the CHR$ function. Execute the following.

```
PRINT CHR$(12)
```

The printer usually responds by executing a form-feed – it moves the paper to the beginning of the next blank sheet, and re-positions the print head to the beginning of the first line.

Other control characters may be valid for your printer. For example, sending a control-N to the 82905B printer changes the character size of subsequent text.

```
30  Big$=CHR$(14)
40  PRINT Big$;"Double-Width Text"
50  END
```

Refer to the appropriate printer manual for a complete listing of control characters and their effect on your printer. Note that some printers allow control characters to affect only the line of text on which they were used.

Escape-Code Sequences Similar in use to control characters, escape-code sequences allow additional control over most printers. These sequences consist of the escape control character, CHR$(27), followed by one or more characters.

For example, the 2631A printer is capable of printing characters in several different fonts. To print extended characters on this printer, an escape code sequence is sent to the printer before the actual text to be printed is sent.

```
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  END
```

Since each printer may respond differently to control characters and escape code sequences, check the manual that came with your printer.

Formatting Information

This section describes how to use the DISP and PRINT statements to "format" the information you print.

For many applications the PRINT or DISP statement provides adequate formatting. The simplest method of formatting is by specifying a comma or semicolon between items.

When the comma is used to separate items, the items are aligned on field boundries. Fields start in column one and occur every 21 columns (columns 1,22,43,64,...). Here is an example of this type of formatting with PRINT statements:

```
PRINT "123456789012345678901234567890123456789";
PRINT "012345678901234567890123456789"
DATA 1.1,-22.2,300000,5.1E+8
READ A,B,C,D
PRINT A,B,C,D
```

Here are the results:

```
123456789012345678901234567890123456789012345678901234567890123456789
 1.1                 -22.2                 300000               510000000
```

Using the semicolon as the separator causes the numbers to be put as closely together as the compact form allows. The compact form always uses one leading space (or − when the number is negative) and one trailing space. That is why the positive numbers in the previous example appear to print one column to the right of the field boundries. The next example shows how the compact form prevents numeric values from running together.

```
PRINT "123456789012345678901234567890123456789";
PRINT "012345678901234567890123456789"
DATA 1.1,-22.2,300000,5.1E+8
READ A,B,C,D
PRINT A;B;C;D
```

Here are the results:

```
123456789012345678901234567890123456789012345678901234567890123456789
 1.1 -22.2  300000  510000000
```

The comma and semicolon are often all that is needed to format a simple table.

You can also format the entire contents of an array, using the comma or semicolon to control the format of the output. Here is an example of printing an array in which each array element contains the value of its subscript:

```
10 OPTION BASE 1
20 DIM A(5)
30 DATA 1,2,3,4,5
40 READ A(1),A(2),A(3),A(4),A(5)
50 PRINT A(1);A(2);A(3);A(4);A(5)
60 END
```

Here are the results:

```
 1  2  3  4  5
```

Another method of aligning items is to use the TAB function.

```
10  PRINT "1234567890123456789012345678901234567890123456789"
20  PRINT TAB(16);"*"
30  END
```

Here are the results:

```
1234567890123456789012345678901234567890123456789
               *
```

A more powerful formatting technique employs the ability of the PRINT and DISP statements to use an image to specify the format.

Using Images Just as a mold is used for a casting, an image can be used to format data. An image specifies how each item should appear. The computer then attempts to format the items 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 list of items to be printed.

This statement prints the value of π (3.141592654...) rounded to three digits to the right of the decimal point.

```
PRINT USING "D.DDD";PI
```

Here is its result:

```
3.142
```

For each character "D" within the image, one digit is 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.14D";PI

3.14159265358979
```

Instead of typing fourteen "D" specifiers, one for each digit, a shorter notation was used to specify a repeat factor before the digit field specifier. The image "DDDDDD" is the same as the image "6D".

The image specifier can be included in the PRINT or DISP statement or on its own line. When the specifier is on a different line the PRINT or DISP statement accesses the image by either its line number or line label.

```
100   Format: IMAGE 6Z.DD,X
110   DATA 1.5,25.57,.056,-.555,-3.4,-88.9
120   READ A,B,C,D,E,F
130   PRINT USING Format;A,B,C
120   PRINT USING 100;D,E,F
150   END
```

Executing this program gives the following results:

```
000001.50 000025.57 000000.06
-00000.56 -00003.40 -00088.90
```

Notice that the image specifier Z filled the field to the left of the radix with zeros.

Numeric Image Specifiers Several characters may be used within an image to specify the appearance of a numeric value.

Image Specifier	Purpose
D	Replace this specifier with one digit of the number to be printed. If the digit is a leading zero, then print a space. If the value is negative, then one leading space may be used by the negative sign.
Z	Same as "D" except that leading zeros are printed.
E	Prints two digit of the exponent after printing the sequence "E+". This specifier is equal to "ESZZ". See the *Technical 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 (radix).
R	Print the comma radix.

To better understand the operation of the image specifiers examine the following examples and results.

```
Statement                                        Output

   PRINT USING "K";33.666                         33.666
   PRINT USING "DD.DDD";33.666                    33.666
   PRINT USING "ZZZ.3D";33.666                    033.666

   PRINT USING "ZZZ";.444                         000
   PRINT USING "ZZZ";5.55                         005

   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
```

To specify multiple fields within the image, the field specifiers are separated by commas.

```
PRINT USING "K,5D,5D";100,200,300

100  200  300

PRINT USING "ZZ,DD,DD";1,2,3

01 2 3
```

If the items to be printed can each use the same image, then the image need be listed only once. The image will then be re-used for the subsequent items.

```
10 PRINT "123456789012345678901234567890123"
20 PRINT USING "5D.DD" ; 3.98,5.95,27.5,129.95
30 END
```

This program produces the following after execution:

```
123456789012345678901234567890123
   3.98    5.95   27.50  129.95
```

The image is re-used for each value. However, an error will result if the number cannot be accurately printed by the image specifier. For instance, the number 20 cannot be accurately printed by the "D" image specifier, since it requires at least two significant digits.

String Image Specifiers Similar to the numeric field image characters, several characters are provided for the formatting of strings.

Image Specifier	Purpose
A	Print one character of the string. If all characters of the string have been printed, then print a trailing blank.
X	Print a space, CHR$(32).
"literal"	Print the characters between the quotes.

Note that the long strings of numbers above the results are used to show column spacing they are not part of the result. The same type of long number strings were used in previous programs for the same purpose but they were part of the program output.

The following examples show various ways to use string specifiers.

Executing these statements:

```
PRINT "12345678901234567890123456 7"
PRINT USING "5X,10A,2X,10A";"Tom","Smith"
```

Produces the following results:

```
1234567890123456789012345 67
     Tom         Smith
```

Executing these statements:

```
10 IMAGE 5X,"John",2X,10A
20 PRINT "1234567890123456789012"
30 PRINT USING 10;"Smith"
40 END
```

Produces the following:

```
1234567890123456789012
     John  Smith
```

Executing these statements:

```
10 IMAGE "PART NUMBER",2X,10D
20 PRINT "12345678901234567890123"
30 PRINT USING 20;90001234
40 END
```

Produces the following:

```
12345678901234567890123 4
PART NUMBER    90001234
```

Additional Image Specifiers The following image specifiers serve a special purpose.

Image Specifier	Purpose
B	Print the ASCII character whose code is given by the "binary" number. (This is similar to the CHR$ function.)
#	Suppress the otherwise automatic end-of-line sequence (carriage-return and line-feed).
/	Send an end-of-line sequence.

Examples

To print a form-feed but suppress the automatic end-of-line sequence, execute the following:

```
PRINT USING "#,B";12
```

To print the ASCII characters that correspond to the codes given by three integers, execute the following statement:

```
PRINT USING "B,B,B";67,97,116
```

The following appears on the display:

```
Cat
```

Accepting Messages from the Operator

There are several ways to get data from the operator:

- From the keyboard
- From a positioning device (such as a mouse or graphics-tablet stylus)
- From an audio input device

The main focus of this section is on inputs from the keyboard. Inputs from positioning devices are described in the "Grapics" chapter. Audio input is beyond the scope of this book.

Types of Keyboard Inputs

There are two general methods of getting operator input through the keyboard:

- With softkeys
- With alphanumeric keys

Softkeys

When possible, using softkeys is a very good choice. It limits the number of alternative inputs, thereby eliminating the need for translating an endless variety of typing mistakes that might be made by the operator. Another benefit is that softkey input is very tightly controlled by the programmer.

ON KEY# statements are used to set up and enable interrupt service routines to be executed when each softkey is pressed[1]. The KEY LABEL statement updates the screen with visual reminders of each softkey's definition.

```
100  ! This program shows simple usage
110  ! of the softkeys and system clock.
120  !
130  CLEAR !     Clear alpha screen.
140  OFF CURSOR ! Disable visual cursor.
150  !
160  ! Set up softkey definitions.
170  ON KEY# 1,"Start" GOSUB Starts
180  ON KEY# 2,"Stop" GOSUB Stops
190  ON KEY# 3,"Reset" GOSUB Resets
200  ON KEY# 4,"Lap" GOSUB LapTime
210  KEY LABEL ! Show softkey labels on screen.
220  !
230  ! Set up initial screen.
240  ALPHA 2,1 @ DISP "Time of day:",TIME$
250  ALPHA 4,1 @ DISP "Start time:",HMS$(0)
260  ALPHA 6,1 @ DISP "Elapsed time:",HMS$(0)
270  ALPHA 8,1 @ DISP "Lap time:",HMS$(0)
280  !
290  Loop: ALPHA 2,22 @ DISP TIME$
300        WAIT 400 !   Dummy delay.
310        IF NOT Timing THEN Loop !  Don't update elapsed.
320        ALPHA 6,22 @ DISP HMS$(TIME-Tstart) ! Elapsed.
330      GOTO Loop
```

1 Service routines are described in the "Program Structure and Flow" chapter.

```
340       !
350 Starts: Tstart=TIME
360         Timing=1 !    Set flag.
370         ALPHA 4,22 @ DISP HMS$(Tstart)
380       RETURN
390       !
400 Stops: Timing=0 !     Clear flag.
410      RETURN
420       !
430 Resets: Timing=0 !    Clear flag.
440         ALPHA 4,22 @ DISP HMS$(0)
450         ALPHA 6,22 @ DISP HMS$(0)
460         ALPHA 8,22 @ DISP HMS$(0)
470        RETURN
480        !
490 LapTime: IF NOT Timing THEN RETURN
500          ALPHA 8,22 @ DISP HMS$(TIME-Tstart)
510        RETURN
520        !
530        !
540  END
```

Here is a typical starting screen produced by the program:

```
Time of day:        14:35:45

Start time:         00:00:00

Elapsed time:       00:00:00

Lap time:           00:00:00
```

When the program begins execution, only the time of day is updated. The other times shown remain 00:00:00.

Pressing the **Start** key initiates a branch to the subroutine named "Starts." This subroutine directs the program to set the "timing flag" by assigning a value of 1 to the variable named Timing. When this flag is set, the elapsed time is shown along with the time of day when the timing was started (in the "Loop" segment).

Pressing the **Lap** key initiates a program branch to the LapTime subroutine, which displays the time elapsed since the **Start** key was pressed.

Pressing the **Stop** key initiates a branch to the Stops subroutine, which halts timing by clearing the timing flag. When the main loop is executed subsequently, the elapsed time is no longer updated.

Pressing the **Reset** key initiates a branch to the subroutine named Resets. This routine displays 00:00:00 for Start, Elapsed, and Lap times.

The program employs several techniques of moving the cursor and displaying data that were shown earlier in this chapter.

Alphanumeric Input Methods

Unfortunately, it is often necessary to leave the comfortable, controlled world of softkeys. For instance, suppose you need to get a number, such as a device selector, from the operator. Valid values of device selectors range from 1 through 1030. You can't very well define a softkey that increments a variable and expect the operator to press it several hundred times! Instead, you will normally ask the operator to use numeric keys to enter the number.

There are two methods that you can use to accept alphanumeric inputs from the keyboard:

- Use INPUT or LINPUT to enter values that can be assigned to string and numeric variables.
- Use ON KYBD to input individual keystrokes.

With the INPUT and LINPUT statements, the operator can type in information, use the cursor control or backspace keys to edit the data if necessary[1], and then press the carriage-return key to send the data to the BASIC system for evaluation and assignment to corresponding BASIC variable(s). This method is the "high-level" approach to accepting keyboard input, since it lets the system handle the often tricky details of moving the cursor, displaying and erasing characters on the screen, and so forth.

1 These keys may be labeled differently on your particular keyboard. See the subsequent section called "Enabling and Disabling Keys" for further details.

With the ON KYBD statement, each key is handled individually by a service routine, and you, the programmer, have to implement any desired editing capabilities. The ON KYBD method is the "low-level" method, since it involves much more detail; however, it gives the program greater control and flexibility.

With both of these methods, you can use the ENABLE KBD statement to enable or disable certain keys (or groups of keys). For instance, you can disable the **Reset** and **Break** keys[1] while still allowing the typing-aid and alphanumeric keys to function normally.

Before getting into the details of using these methods, here are some general suggestions that apply to all methods of accepting keyboard inputs.

Anticipate Common Problems One task that can be performed by the input routine is to anticipate common problems. Many techniques 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 string variables in an INPUT statement.
- You know that CAPS LOCK might be on or off when the operator starts typing, so use the UPC$ string function to convert the inputs to uppercase characters before comparing them to string constants.
- You know that an operator is likely to just execute CONT (continue) if he isn't sure how to respond, so make sure that your input routine can handle a null response and that it assigns a reasonable default answer for such inputs.

1 These keys may be labeled differently on your particular keyboard. See the subsequent section called "Enabling and Disabling Keys" for further details.

Error Trapping Simplifies Input Routines No matter how much time you have spent anticipating possible errors and making an input routine "bomb-proof," you can always find someone who can enter an incorrect response. However, don't feel bad, because the proper handling of keyboard input may be one of the most difficult areas of applications programs. Instead of writing elaborate input routines that can parse broken English with misspelled words, you can use the ON ERROR mechanism to trap errors that have not been (or cannot be) anticipated. The objective in such an approach is two-fold: to keep the program running, and to give the operator a chance to correct the mistake.

Here is a typical example. You ask the operator for a file name. Your program can't tell if the operator entered the name of a file that exists until it accesses the disc. The ON ERROR routine can tell the operator that the file does not exist on the specified (or default) volume and then ask for another file name. See the "Handling Errors" chapter for more information on error-trapping techniques.

The Two High-Level Input Methods As mentioned before, there are two keywords available for accepting alphanumeric keyboard inputs:

- INPUT
- LINPUT

Both statements allow you to enter string values into BASIC variables; however, only INPUT also allows inputs into numeric variables. Here is an examples of using INPUT:

```
100  ON ERROR GOTO AskEmplNum   ! Set up error trap.
110  !
200 AskEmplNum: DISP "Please enter your employee number."
210             INPUT EmplNum
220             DISP "Is this correct ? (Y/N)",EmplNum
230             INPUT Answer$
240  IF UPC$(Answer$)<>"Y" THEN AskEmplNum
250  OFF ERROR !               De-activate error trap.
```

The example first sets up an ON ERROR branch to the beginning of the input routine. Let's look at how the routine would be executed *without* input errors before describing its error-trapping behavior.

The program displays a prompt for information (with DISP), and then directs the system to await numeric input data (with INPUT). The operator is then expected to type in a number and press the carriage-return key (to send the data to the system for evaluation and assignment into the numeric variable named EmplNum). If the operator enters a "valid" number, then program execution continues with the next line (220).

Next the program "echoes" the input data on the screen and asks the operator to verify that it is correct. If it is, then the operator is expected to type a "Y" and press the carriage-return key again. Note that this section anticipates a common problem – lettercase disagreement – by converting that the first character of the answer to an uppercase letter before comparing it with the uppercase "Y" that indicates an affirmative response.

Now back to the error-trapping mechanism. This is probably the simplest form of error trapping during input from the operator; it merely asks for the operator to input data again. The program will continue to do so until there are no run-time errors during the program. A typical error would be the operator entering string data with no numeric characters. In such case, the system would normally report: `Error 43 on line 210 NUMERIC INPUT REQUIRED`. However, since this program branches to AskEmplNum upon detecting an error, the error report is disabled and the operator is asked again to enter the number.

Here is a similar example that uses LINPUT. (LINPUT stands for "Literal INPUT".)

```
100  ON ERROR GOTO AskIncome    ! Set up error trap.
110  !
120 AskIncome: LINPUT "Monthly income?",Income$
130            LINPUT "Is this correct ? (Y/N)  "&Income$, Answer$
140  IF UPC$(Answer$)<>"Y" THEN AskIncome
150  OFF ERROR !                   Disable error trap.
```

This program uses LINPUT for the primary reason that most people (in America, anyway) use commas in numbers between the hundreds and thousands places, and so forth. For example: `1,500.00`. If you tried to use INPUT to enter this number into a numeric *or string* variable, you would get an erroneous value of 1. This is because INPUT interprets the comma as a field separator. Using the LINPUT statement allows you to enter the number, commas and all, into a string variable. The program can then parse the string to remove the commas, if necessary.

The preceding examples show that there are several differences between INPUT and LINPUT.

The **main advantages of INPUT** are as follows:

- Either numeric or string values can be input.
- 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.

Two of the **disadvantages of INPUT** are as follows:

- Improper entries to numeric variables can cause errors, such as `Error 130 NUMERIC VALUE REQUIRED` and `Error 2 OVERFLOW`.
- Certain characters can cause problems. Commas and quote marks have special meanings and are the primary offenders.

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 BASIC system will display error 43 until the operator either gets tired or realizes the mistake. 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 the input, because the erroneous input initiates a branch back to the beginning of the input routine.

The LINPUT statement can help with these potential problems. The result of any LINPUT statement is a single string that contains an *exact image* of what the operator typed. If no data are input, then the variable is given the value of the "null string" (a string of length 0 characters). If you need things like default values, numeric quantities, and multiple values, then 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 hold a line of text (usually 80 characters) or more, then it becomes highly unlikely that the operator will overflow the string from the keyboard. Therefore, LINPUT is a very "safe" way to get data from the keyboard.

To find out further details regarding the use of INPUT and LINPUT, see the *Technical BASIC Language Reference.*

Enabling and Disabling Keys

You can use the ENABLE KBD statement to enable and disable certain keys and groups of keys. For instance, you can disable the special function keys (during program execution) by executing this statement:

```
100  ENABLE KBD 255-2^5 ! All bits 1, except bit
```

The numeric parameter is the *mask* that specifies which keys are to be enabled or disabled; a 1 in a certain bit position enables that key (or group), while a 0 disables the key(s). Bit definitions in the mask are shown in the *HP-UX Technical BASIC Language Reference.*

Here are the two keys and two key groups that can be enabled and disabled with this statement:

- **Reset**
- **PAUSE** (or **Break**)

- Special function keys (or "softkeys")
- All other keys (such as alphanumeric and cursor-control keys)

The corresponding keycap labels are shown in the *Getting Started* manual for your particular Technical BASIC system.

Low-Level Keyboard Input Routines

With Technical BASIC, you have the capability of trapping every keystroke using the ON KYBD statement. You can use this feature to design very effective keyboard interfaces. However, the programming effort for this type of application is often relatively large. In fact, using ON KYBD to accept keyboard input while displaying a cursor and positioning text is essentially writing a text editor. Unfortunately, programs of that magnitude are beyond the scope of this manual.

Here is an example that shows a simple usage of ON KYBD to detect presses of alphanumeric keys.

```
100  INTEGER KeyBuffer ! Key codes will be stored in this variable.
110  Keys$="ABC" !       Define keys that will initiate branches.
120  !
130  ON KYBD KeyBuffer,Keys$ GOSUB KBDService
140  !
150 Spin:  GOTO Spin
160  !
170 KBDService: ! Service routine for ON KYBD
180    IF KeyBuffer=NUM("A") THEN DISP "Alpha"
190    IF KeyBuffer=NUM("B") THEN DISP "Bravo"
200    IF KeyBuffer=67 THEN DISP "Charlie"
210  RETURN
```

The INTEGER statement declares a variable named KeyBuffer, which will be used as a one-keystroke buffer; in other words, when a key is pressed, the numeric code that it generates will be stored in this variable.

The Keys$ string variable is used to define which keys will be enabled to initiate a program branch. As each key is pressed, the string specified by Keys$ is searched for the presence of the corresponding code. If a match is found, then the branch is initiated; if not, the keystroke is ignored.

The ON KYBD statement enables branching to the subroutine called KBDService; the branch will be initiated whenever any of the keys that generate a code specified in the Keys$ variable is pressed. For instance, running the program and pressing an uppercase A will initiate a branch to the service routine.

The service routine can then determine which key was pressed by accessing the integer variable named KeyBuffer. This service routine defines different actions for pressing each key; pressing A results in the program displaying "Alpha"; pressing B results in "Bravo"; pressing C results in "Charlie".

Note that the key buffer contains the **numeric** code for the key, not the alphanumeric character that the key produces on the screen. Note also that pressing a key which generates a lowercase letter does **not** initiate a branch to the service routine.

Since the keyboard buffer is only one character in length, the service routine can miss keystrokes if keys are not processed quickly. This is due to the fact that keycodes are placed in the buffer as each keypress is detected; if a keycode is already there, then it is overwritten.

In order to disable certain key(s) from initiating the branch, you can execute an OFF KYBD statement that specifies the key(s). For example, this statement would disable **only** uppercase C from initiating the previously defined branch.

```
OFF KYBD "C"
```

Here is an example of disabling **all** keyboard branching:

```
OFF KYBD
```

With ON KYBD, you can also trap keystrokes which would otherwise cause immediate action. For example, most keyboards have a **BackSpace** key that you can press to move the cursor left one space and erase the character at that location. Since this type of key produces an "escape sequence" (a sequence of characters beginning with the ASCII control character "escape"), you can include the key's escape sequence in the key string[1]. Here is an example:

1 A list of the keys and the code that each produces is provided in the *Implementation Specifics* appendix to the *HP-UX Technical BASIC Language Reference.*

```
320  BackSpace$=CHR$(27)&"D"
330  ON KYBD KeyBuffer,BackSpace$ GOSUB KBDService
```

The program segment places the escape code that is produced by the **BackSpace** key (on the Integral computer) into the variable named **BackSpace**, and then enables the **BackSpace** key to initiate a branch by executing an ON KYBD statement.

Reading Text from the Screen

Somewhere between the high-level INPUT or LINPUT and the low-level ON KYBD statements lies another method of accepting alphanumeric input. The AREAD statement reads text from the screen into a string variable. See the example of using this statement in the preceding section of this chapter called ''Sending Messages to the Operator''.

10 Using the Clock and Timers

HP-UX systems feature a real-time clock that maintains date and time of day. You can access this clock from the Technical BASIC system[1]. There are also timers that allow you to generate interrupts at specified intervals.

Chapter Contents

This chapter covers using the clock and timers. Here are the topics covered:

- Reading the current date.
- Reading the current time of day.
- Converting between various time and date formats.
- Measuring time intervals.
- Enabling timers to interrupt normal program flow at specified intervals.

1 On multi-user HP-UX systems, *only* the system administrator can set the time and date.

Using the Clock

This section discusses the Technical BASIC features available for reading the date and time of day, and for measuring time between events.

Reading the Date

The DATE$ string function returns the current date in the form: *yy/mm/dd* (year/month/day).

```
DATE$
84/10/17
```

The numeric function for obtaining the date is DATE. Executing this function returns a date in the form: *yyddd* (*yy* indicates the last two digits of the year; *ddd* indicates the day of the year, in the range 1 through 366).

```
DATE
 84291
```

where the date displayed is the 291st day of the year 1984.

Reading the Time of Day

The TIME$ string function returns the system clock reading in this 24-hour notation: *hh:mm:ss* (hours:minutes:seconds). Assuming your system clock has been properly set, the reading returned by the TIME$ function shows the time elapsed since midnight of the current day.

```
TIME$
 8:54:47
```

Another method for determining the time elapsed since midnight is to use the TIME numeric function. This function returns the total number of seconds elapsed since midnight. This example of invoking the TIME function returns the numeric equivalent of a time of day of "8:54:57".

```
TIME
 32087
```

The last value returned in a day's time is 86 399. When the counter reaches this value, it is reset to 0 and the date is incremented by one. Note that all of the functions mentioned in this section are programmable. The following short program is an example:

```
10  DISP "Today's date is: "; DATE$;" or";DATE
20  DISP "This program was run at: ";TIME$
30  DISP "Time of day (in seconds since midnight) is: ";TIME
40  END
```

Time and Date Format Conversions

Technical BASIC has additional time functions that perform the following notational conversions:

- Converting a specified number of seconds (since midnight) to an hours:minutes:seconds (*hh:mm:ss*) string format.
- Converting a string in the form *hh:mm:ss* to the equivalent number of seconds (since midnight).
- Converting a specified Julian day number to a month/day/year (*mm/dd/yyyy*) string format.
- Converting a string in the form month/day/year (*mm/dd/yyyy*) to the equivalent Julian day number.

Time: Numeric to String Conversions HMS$ is a function which converts a specified number of seconds (since midnight) to an equivalent string in the form *hh:mm:ss* (hours:minutes:seconds). An example is as follows:

```
HMS$(TIME)
```

Here is what it might return:

```
09:45:52
```

Here is another example:

```
DISP "Elapsed time = ";HMS$(Time2-Time1)
```

In this case, the time returned would not be in seconds since midnight; however it would be in a more usable form than just seconds:

```
10:10:00
```

Thus, the elapsed time is 10 hours, ten minutes, and no seconds.

Time: String to Numeric Conversions The HMS function does the opposite of HMS$. This function converts a string in the form *hh:mm:ss* (hours:minutes:seconds) to the integer equivalent in seconds. Here is an example:

```
100  ! Calculate time differential
110  ! and display in "hh:mm:ss" format.
120  !
130  DISP "Time between  6:08:29 p.m."
140  DISP "          and 10:14:32 a.m."
150  !
160  ! Now calculate difference (in minutes).
170  Diff=HMS("12:00:00")+HMS("06:08:29")-HMS("10:1
180  ! Then re-format for human consumption.
190  Diff$=HMS$(Diff)
200  !
210  DISP "           is ";Diff$
220  !
230  END
```

Here are the program's results.

```
Time between  6:08:29 p.m.
         and 10:14:32 a.m.
          is  7:53:57
```

The HMS function can be also be executed from the keyboard:

```
HMS("13:30:15")
```

which returns on the display:

```
48615
```

Date: Numeric to String Conversions The MDY$ string function converts a Julian Day number[1] to an equivalent string expression in the form: *mm/dd/yyyy* (month/day/year). The range of Julian Day numbers that you can pass to this function is from 2299161 through 3199160; these limits correspond to October 15, 1582[2] and November 25, 4046, respectively.

1 The Julian Day number is an astronomical convention representing the number of days since January 1, 4713 B.C.

2 The beginning date of the modern Gregorian calendar.

Here is an example of how you can use MDY$:

```
MDY$(2446000)
```

The function returns the Julian Day number in a more understandable format.

```
10/26/1984
```

Date: String to Numeric Conversions The MDY numeric function does the opposite of the MDY$ function. When it is given a string in the form *mm/dd/yyyy,* it returns the equivalent Julian Day number. Note that the string must lie between the dates 10/15/1582 and 11/25/4046, and consist of exactly 10 characters (including the two slashes). Here is an example of using the function from the keyboard:

```
MDY("11/25/4046")
```

returns the following on the display:

```
3199160
```

Here is a more current example:

```
MDY("10/17/1984")
```

returns the following on the display:

```
2445991
```

Timing the Interval Between Events

Measuring the time between two events is quite simple.

```
100  Tinit=TIME !      Initial time.
110  !
120  FOR J=1 TO 5555
130  !
140  NEXT J
150  !
160  Tfinal=TIME !     Final time.
170  !
180  DISP "Elapsed time =";Tfinal-Tinit;"seconds."
190  !
200  END
```

Here are typical results of running the program:

```
Elapsed time = 4 seconds.
```

Note that the program does not keep track of changes in the day. Thus, if you are timing events that will occur near midnight, you may get a negative time interval. You may want to add code that keeps track of days also. For example, you could multiply the difference in days by the number of seconds in a day (86 400), and add this figure to the differential.

Using the Timers

This section covers the following timer operations:

- Timer branches.
- Measuring time elapsed since a timer was set to interrupt.

Timer Interrupts

The subject of event-initiated branching was discussed near the end of the "Program Structure and Flow" chapter. If you are not familiar with the concept, you may want to read that section before reading this section.

Here are the statements that control timer-initiated branches:

- ON TIMER# sets a specified timer to zero and immediately activates it. The end-of-line branch is initiated when the specified time interval has elapsed. Three timers are available for this purpose; they are numbered 1, 2, and 3.
- OFF TIMER# disables branching for the specified timer.

You can use these timers to generate these types of interrupts:

- Cyclic interrupts
- Delay interrupts
- Time-of-day interrupts

Cycle and Delay Interrupts The ON TIMER# statement enables a branch to be taken as soon as the specified number of *milliseconds* have elapsed. For instance, the following statement enables a GOSUB branch to the subroutine called Cycle2 to occur two seconds from the time that the statement is executed:

```
ON TIMER# 1, 2000 GOSUB Cycle2
```

ON TIMER# remains in effect, re-initiating a branch every two seconds until an OFF TIMER# statement is executed (for timer number 1). Thus, the ON TIMER# statement creates a cyclic interrupt.

To produce a one-time timer interrupt (i.e., a delay interrupt), you will need to execute an OFF TIMER# statement in the timer service routine.

This example shows both usages of timers. It displays the time (hours:minutes:seconds) for a period of two seconds and then prints five random numbers. It repeats this process until eight seconds have elapsed, at which time the program is ended. The ON TIMER# 1 is a cyclic interrupt, while the ON TIMER# 2 statement, along with its OFF TIMER# 2 counterpart, act as a one-time delay interrupt.

```
100  ON TIMER# 1,2000 GOSUB TwoSecCycle
110  ON TIMER# 2,8000 GOTO EightSecDelay
120  !
130  CLEAR !  Clear screen.
140  !
150  DummyLoop: ALPHA 5,1 @ DISP TIME$
160             GOTO DummyLoop
170             !
180  TwoSecCycle: ALPHA 8,1
190               FOR Number=1 TO 5
200                 DISP RND ! Random number.
210               NEXT Number
220               !
230  RETURN
240  !
250  EightSecDelay: OFF TIMER# 2
260                 ALPHA 15,1
270                 DISP "Finished"
280                 !
290  END
```

Here is typical output from the program.

```
12:20:25


 0.744804223761712
 0.0289620654927213
 0.559984130375072
 0.311563463240455
 0.114398065498712


Finished
```

Simulated Time-of-Day Interrupts The ON TIMER# statement allows you to define and enable a branch to be taken when the timer reaches a specified count. You can simulate time-of-day interrupts by using this procedure:

1. Determine the current time of day.
2. Determine the desired time of day at which the interrupt will occur.
3. Calculate the number of seconds between the two.
4. Set a timer interrupt for that number of seconds (from the present time).

Typically, the ON TIMER# statement is used to cause a branch at a specified time. This statement can be use as an interval timer in a program, by storing in a program variable the value of the system clock when the program is started (using the function called TIME) and subtracting this value from a specified final time. The following example uses the interval timer as an alarm to remind you to go to lunch.

```
100  DISP "This is the present time of day: ";TIME$
110  DISP
120  DISP "Specify alarm time using this format: 'hh:mm:ss'"
130  LINPUT Tfinal$ @ DISP "Thank you."
140  !
150  ! Determine number of seconds since midnight.
160  Tfinal=HMS(Tfinal$)
170  ! Set timer to interrupt in Tfinal-TIME (seconds).
180  ON TIMER# 1,(Tfinal-TIME)*1000 GOSUB Alarm
```

```
190  !
200  Spin: GOTO Spin !  Twiddle thumbs.
210  !
220  Alarm:
230      BEEP
240      CLEAR
250      DISP @ DISP @ DISP "Time for lunch!" @ DISP
260      OFF TIMER# 1
270      RETURN
280      !
290  END
```

Here are is the screen that the program produces:

```
This is the present time of day:  9:38:54

Specify alarm time using this format:  'hh:mm:ss'
? 11:45:00
Thank you.
```

The first line of output gives the current time of day. The second line asks you to set the alarm for a time of your choosing using the 'hh:mm:ss' format. The string you enter (11:45:00 above) is converted by the numeric function HMS into seconds since midnight and assigned to the variable Tfinal. Next, timer number 1 is set to interrupt; the time interval is calculated as the difference between "Tfinal" and the current TIME (it is multiplied by 1000 to convert the result to milliseconds, which is how the ON TIMER# statement interprets the interval parameter).

When the specified time interval has elapsed, the timer interrupt service routine displays the "lunch alarm" message.

```
Time for lunch!
```

Timer Functions

The READTIM numeric function returns the number of seconds currently registered on the specified system timer.

- For timer numbers 1, 2, or 3, this is the number of seconds (**not** milliseconds) since the timer was set in the program, or since it last initiated a branch.

- For timer 0, it is the number of seconds elapsed since the system clock was last set, either by the system administrator or by power on.
- If the timer is not currently being used, then READTIM returns 0.
- After an OFF TIMER# statement, READTIM returns the reading of the timer at the point it was disabled.

The following program makes use of the READTIM function. It programmatically defines a function key to call a routine which displays the number of seconds elapsed since timer 1 was set. After ten minutes have elapsed, it displays the message:

```
Ten minutes have elapsed.
```

and then issues a beep.

```
100 ON TIMER# 1,10*60*1000 GOSUB TenMin ! Interrupt after 10 min.
110 ON KEY# 1,"Seconds" GOSUB Elapsed !   Show elapsed time.
120 !
130 Spin: GOTO Spin !                      Idle loop.
140 !
150 STOP
160 !
170 TenMin: DISP "Ten minutes have elapsed."
180         BEEP
190   RETURN
200   !
210 Elapsed: DISP READTIM(1);"seconds since timer 1 set."
220   RETURN
```

Pressing **k1**[1] directs the program to display the number of seconds since TIMER# 1 was set. Here are typical results that the Elapsed subroutine displays:

```
3 seconds since timer 1 set.
4 seconds since timer 1 set.
9 seconds since timer 1 set.
```

1 On some consoles, this key is labeled **k0**. Refer to the *Getting Started* manual for your particular Technical BASIC system for a description of ON KEY# parameters and softkey labels.

Timers and Subprograms

It is possible for a context (program or subprogram) to enable a timer interrupt and then call one or more subprograms before the timer interrupt occurs. As long as the context is *not* executing a subprogram *when* the timer is expected to interrupt, the interrupt will initiate its branch at the correct time. However, if the subprogram *is* being executed when the timer would have otherwise initiated its branch, then the branch to the service routine is not executed until *after* control returns to the context that defined the timer interrupt.

Timer Interrupts While Not Executing a Subprogram The following program is an example of the situation in which the subprogram is finished before the timer interrupts. (The situation of the subprogram being executed when the calling context's timer interrupt would have occurred is covered in the next section).

```
100 ON TIMER# 1,10000 GOSUB TenSecs !  10-second cycles.
110 !
120 Tinit=TIME !  Store initial time.
130 !
140 FOR I=1 TO 3 !  Wait 3 seconds.
150   WAIT 1000 @ DISP TIME-Tinit;"seconds."
160 NEXT I
170 !
180 CALL "SUBTimer1" (Tinit)
190 !
200 FOR I=1 TO 3 !  Wait 3 more seconds (to allow interrupt).
210   WAIT 1000 @ DISP TIME-Tinit;"seconds."
220 NEXT I
230 !
240 END
250 !
260 TenSecs: DISP
270          DISP "At branch to 'TenSecs', READTIM(1)=";READTIM(1)
280          DISP
290   RETURN
```

Here is the subprogram.

```
100  SUB "SUBTimer1" (Tinit)
110  DISP
120  DISP "Entering SUBTimer1."
130  DISP
140  FOR I=1 TO 5
150    WAIT 1000
160    DISP TIME-Tinit;"seconds."
170  NEXT I
180  DISP
190  DISP "Exiting SUBTimer1."
200  DISP
210 SUBEND
```

Here are the results of running this program.

```
 1 seconds.
 2 seconds.
 3 seconds.

Entering SUBTimer1.

 4 seconds.
 5 seconds.
 6 seconds.
 7 seconds.
 8 seconds.

Exiting SUBTimer1.

 9 seconds.
 10 seconds.

At branch to 'TenSecs', READTIM(1)= 0

 11 seconds.
```

The main program starts out by setting timer number 1 to interrupt in ten seconds. The elapsed time is then read every second and displayed until branching to the subprogram.

The subprogram displays a message telling you that it has been given control. It also displays elapsed times every second (for 5 seconds). After five seconds have elapsed, control is returned back to the calling program.

When timer 1 has counted to 10 seconds, the branch to TenSecs is initiated.

The **main point** of this example is that the main program's timer interrupt occurs at the expected time, because the subprogram is not being executed **when** the timer interrupts.

The program also shows that the timer is reset to 0 (as determined by the READTIM function); however, it does not show that the timer is cyclic and is automatically re-enabled and begins counting again. In this case, the program ended before a second interrupt occurred.

Timer Interrupts while Executing Subprograms The following program and subprogram show an example in which the subprogram is being executed when the calling context's timer interrupts. Note that they are slightly modified versions of the preceding program and subprogram.

```
100 ON TIMER# 1,10000 GOSUB TenSecs !  10-second cycles.
110 !
120 Tinit=TIME !  Store initial time.
130 !
140 FOR I=1 TO 3 !  Wait 3 seconds.
150   WAIT 1000 @ DISP TIME-Tinit;"seconds."
160 NEXT I
170 !
180 CALL "SUBTimer2" (Tinit)
190 !
200 FOR I=1 TO 11 !  Wait 11 more seconds (to allow interrupt).
210   WAIT 1000 @ DISP TIME-Tinit;"seconds."
220 NEXT I
230 !
240 END
250 !
260 TenSecs: DISP
270          DISP "At branch to 'TenSecs', READTIM(1)=";READTIM(1)
280          DISP
290   RETURN
```

Here is the subprogram.

```
100  SUB "SUBTimer2" (Tinit)
110  DISP
120  DISP "Entering SUBTimer2."
130  DISP
140  FOR I=1 TO 10
150    WAIT 1000
160    DISP TIME-Tinit;"seconds."
170  NEXT I
180  DISP
190  DISP "Exiting SUBTimer2."
200  DISP
210 SUBEND
```

Here are the results of running this program.

```
 1 seconds.
 2 seconds.
 3 seconds.

Entering SUBTimer2.

 4 seconds.
 5 seconds.
 6 seconds.
 7 seconds.
 8 seconds.
 9 seconds.
 10 seconds.
 11 seconds.
 12 seconds.
 13 seconds.

Exiting SUBTimer2.


At branch to 'TenSecs', READTIM(1)= 3

 14 seconds.
 15 seconds.
 16 seconds.
 17 seconds.
 18 seconds.
 19 seconds.
 20 seconds.
```

```
At branch to 'TenSecs', READTIM(1)= 0

  21 seconds.
  22 seconds.
  23 seconds.
  24 seconds.
```

The main program starts out by setting timer number 1 to interrupt in ten seconds. The elapsed time is then read every second and displayed until branching to the subprogram.

The subprogram displays a message telling you that it has been given control. It also displays elapsed times every second (for 10 seconds this time). After ten seconds have elapsed, control is returned back to the calling program.

The timer in the main program would have initiated its branch, but could not because the subprogram was being executed. This result is shown by the value 3 being returned by the READTIM function. In the calling context (here the program), timer 1 did count to 10 seconds, but it could not initiate the branch to TenSecs because it was not in the current context (the subprogram).

The **main point** of this example is that the main program's timer interrupt is **delayed**, because the subprogram does not return control to the calling context (main program) until **after** the timer interrupt should have occurred. However, the branch is initiated as soon as control returns to the context in which it is enabled.

The program executes 10 additional 1-second waits, in order to demonstrate that the timer will indeed initiate subsequent branches as expected.

11 Data Storage and Retrieval

This chapter describes some useful techniques for storing and retrieving data. The methods fall into these categories:

- Storing data with programs (using DATA and READ statements)
- Storing data in BASIC/DATA files (using ASSIGN#, PRINT#, and READ#).
- Storing data in text/data files (using ASSIGN, OUTPUT, and ENTER).

To store and retrieve data that is part of the BASIC program, use DATA statement(s) to specify data that is to be stored in the memory area used by BASIC programs; thus, the data is always kept in the same file as the program. 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, mass storage BASIC/DATA files are more appropriate. These files provide means of storing data on mass storage devices. The BASIC/DATA files available with Technical BASIC are described in this chapter. A number of different techniques for accessing data in these files are described in detail.

Files of type text/data are used as the interchange method for sharing data between Technical BASIC and the HP-UX system.

Chapter Contents

This chapter discusses these topics:

- Storing data in programs
- Using data files
- Brief mass storage tutorial
- Introduction to BASIC/DATA file access techniques
- A closer look at file access
- A closer look at serial access
- Random file access
- Determining data types
- Trapping EOF and EOR conditions
- Using text/data files

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 was used in the first example program in the "Program Development" chapter. It was a convenient way to store data without knowing anything about data files.

```
    .
    .
    .
110  OPTION BASE 1
120  DIM IncomeName$(2)
130  REAL TargetIncome(2)
140  !
150  ! Assign values to variables.
160  LET IncomeName$(1)="Payroll"
170  LET IncomeName$(2)="Investments"
180  LET TargetIncome(1)=1680.00
190  LET TargetIncome(2)=345.67
    .
    .
    .
```

This technique works well when there are only a relatively few "constants" 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 one 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   DISP "Please enter your ID, and press Return."
110   INPUT ID
  .
  .
210   LINPUT "Enter the value of X",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, limited only by computer memory (or disc space when the program is stored in a file).

Storing Data When you RUN a program, the system concatenates all DATA statements in a given 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 Payroll,Investments
         .
         .
200 DATA 1680.56,345.67
         .
         .
300 DATA Mortgage
```

Data Stream:					
	Payroll	Investments	1680.56	345.67	Mortgage

As you can see from the example above, a data stream can contain both numeric and string data items.

Each data item must be separated by a comma; string items can optionally be enclosed in quotes. Strings that contain a comma or exclamation mark must be enclosed in quotes. In addition, you must use the following notation for every quote you want in the string. For example, to enter the strings UNQUOTED, UNQUOTED, and "QUOTED" into a data stream, use this DATA statement:

```
100 DATA UNQUOTED,"UNQUOTED","~"QUOTED~""
```

The tilde characters indicate that the quote mark that follows it is to be part of the data read into a string variable.

Retrieving Data 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. Here is an example:

```
100 DATA Payroll,Investments
        .
        .
200 DATA 1680.56,345.67
        .
        .
100 READ Income1Name$,IncomeName2$,TargetIncome1
```

This READ statement would read three data items from the data stream into the three variables. Note that the first and second variables are string and the third is a numeric. This corresponds to the order and type of data items in the data stream.

Numeric data items can be READ into either numeric or string variables, with the following restrictions:

- If the numeric data item is of a different specific numeric type than the numeric variable, then the item is automatically converted. For instance, REALs are converted to INTEGERs, and INTEGERs to REALs. However, if the value is out of range for that numeric data type, then an error is reported.
- If the string variable has not been dimensioned to a size large enough to hold the entire data item, then error 56 is reported.

The Data Pointer 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 initially points 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. When a subprogram is called by a main program (or another subprogram), 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 assigned a value. 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.

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 1984,"The date is "
40  READ Str$
50  Print Str$;Month$;Day;Year$
60  END
```

```
The date is November 26 1984
```

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 example below, we show how DATA values can be assigned to elements of a 3-by-3 numeric array.

```
10  OPTION BASE 1
20  DIM Example(3,3)
30  DATA 1,2,3,4,5,6,7,8,9,10,11
40  MAT READ Example
50  MAT PRINT USING "3(X,K),/";Example
60  END
RUN

 1 2 3

 4 5 6

 7 8 9
```

The data pointer is left at item 10; thus, items 10 and 11 are saved for the next READ statement.

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 of the first data stream in that context. If you do include a line identifier in the RESTORE statement, the data pointer is moved to the first data item in the DATA statement at the identified line. The example below illustrates how to use the RESTORE statement.

```
100   DIM Array1(2)       ! 3-element array (OPTION BASE 0).
110   DIM Array2(4)       ! 5-element array (OPTION BASE 0).
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   MAT 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 (line 120).
190   MAT READ Array1     ! Reads first 3 items in stream.
200   RESTORE 160         ! Moves data pointer to item "8".
210   READ D              ! Reads "8".
220                       !
230   PRINT "Array1 contains:"
240   MAT PRINT Array1
250   PRINT "Array2 contains:"
260   MAT PRINT Array2
270   PRINT "A,B,C,D equal: ";A;B;C;D
280   END
```

Here are the results of running the program.

```
Array1 contains:
  1
  2
  3
Array2 contains:
  4
  5
  6
  7
  8
A,B,C,D equal:  1  2  3  8
```

Using BASIC/DATA Files

This section of this chapter describes another general class of data storage and retrieval methods – that of using mass storage BASIC/DATA[1] files. This material is broken up into several parts.

- A look at mass storage, directories, and data files
- Introduction to accessing BASIC/DATA files
- A closer look at using files
- Determining data types
- Trapping EOF and EOR conditions

Brief Mass Storage Tutorial

This section briefly discusses these topics:

- Mass storage in general
- Directories
- BASIC/DATA files

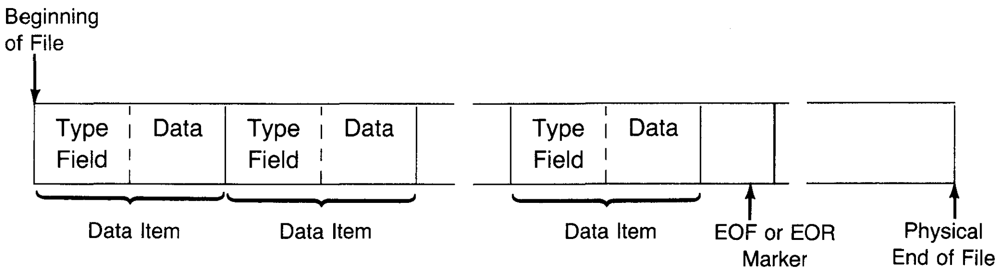
As the adjective "mass" suggests, mass storage devices are data-storage devices which are generally capable of storing "large" amounts of data. Just how much data constitutes a large amount depends on the device itself. However, most mass storage devices are capable of storing on the order of hundreds of thousands to several million data items.

1 The subsequent section called "Using text/data Files" discusses techniques for accessing files of type `text/data`, which is the file type that both Technical BASIC and HP-UX can use (such as for data interchange).

Besides having the ability to store data, mass storage devices are capable of providing means for keeping data **organized** so that logical groups may be accessed systematically and efficiently.

- Data items are organized into logical groups known as *files;* a file is merely a collection of data items which are accessed through one name. Each file may contain one or more *logical records;* each logical record in a file is much like a subset of the file in that it can also contain several data items.
- Files are organized by directories. A directory is an index of files; in any directory, there is an entry for every file within that directory.

When a data file is initially created, it contains nothing. However, you can fill it with any data that you want, which gives the file the general structure shown below.



The data items are stored using either ASCII characters (for string items) or an internal representation (for numeric items). The type fields indicate whether the item is a string or a numeric item. Subsequent sections provide further details of just what the file contains and how to write to and read from them.

The CAT statement shows some of the information that is stored in a directory. Executing CAT with no directory path tells the system to get a catalog of the *current working directory*[1].

```
CAT
```

1 If you don't know the meaning of the term "current working directory," then refer to the discussion of the HP-UX file system in your HP-UX system's documentation.

Specifying a directory path with the file name gives a listing of the files in that directory.

```
CAT "/users/mark/BASICFILE"
```

Introduction to BASIC/DATA File Access Techniques

This section presents BASIC programming techniques useful for accessing BASIC/DATA[1] files. The first section gives a brief introduction to the steps you might take to store data in a file. Subsequent sections describe further details of these steps.

Methods of Accessing Data Files There are two methods of accessing BASIC/DATA files:

- **Serial access:** writing to or reading from the file in sequential order – one item at a time, from the beginning.
- **Random access:** writing to (or reading from) the file, starting at the beginning of any *logical record* within the file. Within any logical record, however, access is strictly serial.

Technical BASIC allows you to use both types of access methods on one file, with only a few restrictions. Each access method has uses in certain applications.

Example of Writing Serially to a File Storing data in files requires a few simple steps. The following program segment shows a simple example of placing several items in a data file.

```
100  ! Allocate memory for variables.
110  OPTION BASE 1
120  DIM IncomeName$(2)
130  REAL TargetIncome(2)
140  !
150  ! Assign variables.
160  IncomeName$(1)="Payroll"
170  IncomeName$(2)="Investments"
180  !
190  TargetIncome(1)=1680.56
200  TargetIncome(2)=345.67
210  !
220  ! Create a data file.
```

1 The subsequent section called "Using text/data Files" discusses techniques for accessing files of type `text/data`, which is the file type that both Technical BASIC and HP-UX can use (such as for data interchange).

```
 230  CREATE "Oct84Income",1 !       Size = 1 logical record.
 240  ASSIGN# 1 TO "Oct84Income" !   Assign a buffer to it.
 250  !
260  PRINT# 1 ; IncomeName$(1),TargetIncome(1)
270  PRINT# 1 ; IncomeName$(2),TargetIncome(2)
280  !
290  ASSIGN# 1 TO "*"  !            Close file (release buffer).
300  !
310  END
```

In order to store data in a file, a data file must be created (or already exist) on the mass storage media to be used. In this case, line 230 creates a BASIC/DATA file for storage. The file is created with 1 *logical* record, which has a default size of 256 bytes. This is a large enough file to store the data in this example. (File size, logical records, and record size are discussed in the subsequent section called "A Closer Look at File Access".)

The file is created in the "current working directory." If the file is to be created in another directory, then the appropriate directory path must be prefixed to the file name. This example creates a file in another directory:

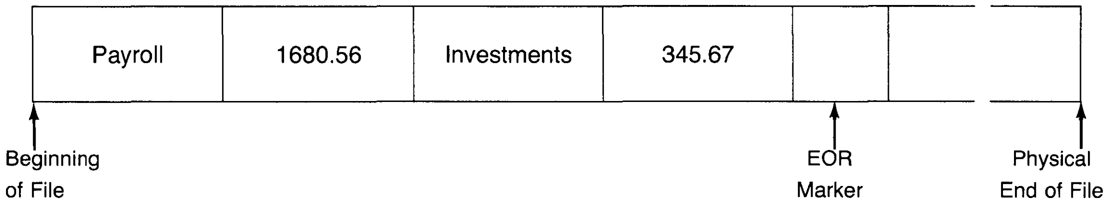
```
CREATE "/users/mark/NovIncomes",4
```

See the *Getting Started* manual for your particular Technical BASIC system for specific information about directories on your system.

Then, in order to store data in (or retrieve data from) the file, you must assign a *buffer number* to the file. Line 240 shows an example of assigning a buffer number to the file (also called "opening the file"). The PRINT# statements on lines 260 and 270 send the previously defined data items being sent to the file.

The file is closed after all data have been sent to the file. (In this case, the close operation is not necessary, because all files are automatically closed by the system by the END statement.)

Here is a conceptual diagram of the file's contents after the program has finished execution.



Although they are not shown in the drawing, the system automatically adds the type fields. You can use the TYP function to read it from BASIC and thus determine the item's type. The subsequent section called "Determining Data Types" gives further details.

The end-of-record (EOR) marker is always placed after the last item written into a file. It is used instead of an end-of-file (EOF) marker, because Technical BASIC allows both random and serial access of the same file. Note, however, that the file is initially filled with EOF markers (when the disc is initialized). Subsequent sections explain EOR and EOF markers in greater depth.

Example of Serially Reading from a File Here is a simple program that reads the data stored in the file created and written in the preceding example.

```
100  ! Allocate memory for variables.
110  OPTION BASE 1  !  Implicit lower subscript bound.
120  DIM IncNam$(2)
130  REAL TstInc(2)
140  !
150  ASSIGN# 1 TO "Oct84Income" !  Assign buffer.
160  !
170  READ# 1;IncNam$(1),TstInc(1) ! Read 2 items from file.
180  READ# 1;IncNam$(2),TstInc(2) ! Read 2 more items.
190  !
```

```
200  DISP
210  DISP "  Category                      Target"
220  DISP " ----------                  --------"
230  DISP IncNam$(1),TgtInc(1)
240  DISP IncNam$(2),TgtInc(2)
250  !
260  ASSIGN# 1 TO "*"
270  !
280  END
```

As in the preceding example, you must assign a file number to the data file before you can access it. Line 150 makes this assignment.

The subsequent READ# statements (lines 170 and 180) read the data into program variables. The general suggestion is to **"read it like you wrote it"**; in other words, match the order and type of each item in the file to the variable into which the item will be read. For instance, if you wrote a SHORT variable, a REAL numeric expression, and a string of length 36 characters into the file, then you should read these items using a SHORT variable, a REAL variable, and a string variable with a length of (at least) 36 characters.

The variable used to read each data item need only be of the same *general* data type as the data item (i.e., numeric or string). It is not strictly required to be of the same *specific* type (i.e., INTEGER, SHORT, or REAL for numeric items; or identical length for string items). However, if the specific types of variables and items are not matched, it becomes possible to generate range errors; for instance, a value of 1E+200 is out of range for INTEGER and SHORT variables, and a string of 200 characters is out of range for a string variable with maximum length of 18 characters.

As you can see, these are very simple examples. However, they show the general steps you must take to serially access files.

A Closer Look at BASIC/DATA File Access

The preceding section showed simple examples of writing data in a file and then reading it back. This section describes what is happening "behind the scenes." It will help you to better understand how to create and use BASIC/DATA files.

File and Record Size Calculations In the preceding example of serially writing a file , the following program line created a file that was used to store the program's data:

```
230 CREATE "Oct84Income",1 !   Size = 1 logical record.
```

The size of the file was specified to be one logical record with (default) record size of 256 bytes. The example stated that this file size was sufficient to store the data. This section will help you verify that statement and show you how to calculate the size of file required to store any data.

The following chart describes the amount of space necessary to store numeric and string data items.

Data Type	Storage Requirements (Per Data Item)
Simple REAL numbers	1 byte for type field, + 8 bytes for number.
Simple SHORT numbers	1 byte for type field, + 8 bytes for number.
Simple INTEGERs	1 byte for type field, + 4 bytes for number.
Simple string	3 bytes for type field, + 1 byte per character, + 3 bytes each time the string crosses a logical record boundary.
REAL array (each element)	1 byte for type field, + 8 bytes for number.
SHORT array (each element[1])	1 byte for type field, + 4 bytes for number.
INTEGER array (each element)	1 byte for type field, + 4 bytes for number.
String array	Each array element requires the same amount of space required for simple string items (shown above)

1 If an entire SHORT array is written into the file as an array variable, then each element requires 5 bytes; however, if SHORT array elements are written individually, then each element requires 9 bytes.

Using the data from the preceding serial access example, here are the calculations for the size of file required to store the data:

Item	Type of data	Bytes required
IncomeName$	"Payroll"	7+3 = 10
	"Investments"	11+3 = 14
TargetIncome	Two REAL numbers	2*(1+8) = 18
		Total = 42

The file size could be 1 logical record (with default size of 256 bytes). You would not need to partition it into smaller logical records, since the data items in the file are only accessed serially.

Note that the size of the file actually created will always be an integral multiple of 1 024 bytes. This effect is due to the fact that the HP-UX file system can only address portions of the disc as small as a 1 024-byte block. Blocks and records are discussed next.

Records and Blocks A *record* or *block* is the smallest unit of mass storage space that is *independently addressable.* There are three types:

- **Logical records** are the smallest unit of mass storage that can be addressed *by a BASIC program.* You can specify the size of logical records in a file when you execute a CREATE statement. If no logical record length is specified, a length of 256 bytes is assumed.
- **Blocks** are the smallest unit of mass storage that can be handled *by the HP-UX file system.* HP-UX file system blocks are always 1 024 bytes in length.
- **Physical records** are the smallest unit of storage that can be addressed *by a mass storage device.* With most HP disc drives, physical records are 256, 512, or 1 024 bytes in length.

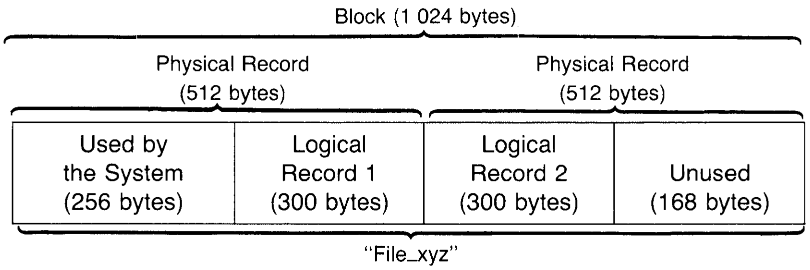
Logical records make it possible to partition a file into several smaller units, each of which the BASIC system can address independently. In fact, each logical record is similar to a file in the respect that it is independently addressable. Within any file, all logical records are the same length; however, each file may have a different logical record length.

Blocks are mentioned only so that you will understand why a file with length of 1 024 bytes will be created if you try to create a file with a length of 256 bytes.

Physical records are only mentioned to avoid confusion with logical records and blocks, should you happen to see that term in your disc manual.

When you create a data file, you specify these parameters: file name, number of logical records, and logical record length (optional). The following drawing shows the file that is created by this statement:

```
CREATE "File_xyz",1,300
```


Block (1 024 bytes)
Physical Record (512 bytes)
Physical Record (512 bytes)
Used by the System (256 bytes)
Logical Record 1 (300 bytes)
Logical Record 2 (300 bytes)
Unused (168 bytes)
"File_xyz"

The example shows several important points about files.

- The file takes up 1 024 bytes of storage, since a file always contains an integral number of blocks. (Similarly, files always begin on a physical record boundary, and thus always contain an integral number of physical records.)
- The Technical BASIC system always uses the first 256 bytes of a BASIC/DATA file for keeping information such as logical record size, number of records, etc.

- After allocating the first 256 bytes for overhead, the system allocates logical records. The first logical record begins at the byte following the last byte of system overhead, and the second record begins on the byte just following the last byte of the first logical record.

  The example also shows that the system will allocate more logical records than specified, if there is room in the file. In this case, there was enough room for one more logical record. As another example, if you create a file with 1 logical record of length 256 bytes, then the file will actually contain 3 records; the system allocates two additional records, rather than leaving the last 512 bytes unusable.
- There are 168 bytes of unusable space at the end of this example file (1 024 − 256 − 2*300), because the next file begins on the next block boundary (which also aligns with the physical record boundary).

The File Pointer The system uses a file pointer to locate and access the data items in a file. The file pointer points to the place where data will be:

- written with the next PRINT# statement, or
- read with the next READ# statement.

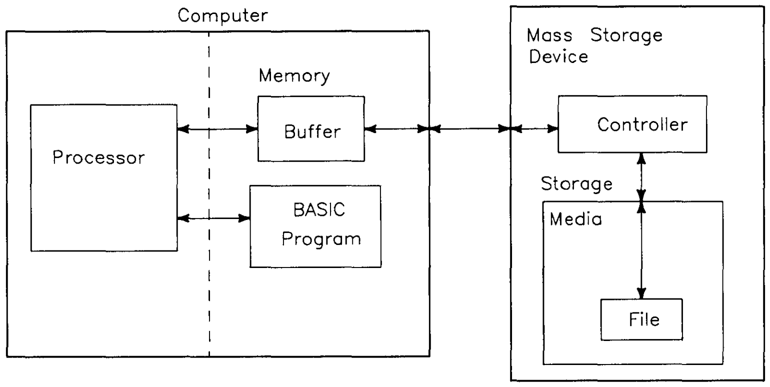
The file pointer is updated automatically by the system whenever the file is accessed. More information about the file pointer will be given in subsequent examples.

File Buffers When a buffer number is assigned to a file, such as in the following statement:

```
ASSIGN# 2 TO "Oct84Income"
```

the BASIC system sets up a file buffer through which it communicates with the mass storage device. This buffer is a small portion of your BASIC memory area, usually a few hundred bytes in length.

Here is a pictorial representation of the communication path.



The purpose of the buffer is to decrease access time for information and reduce the wear on the physical mass storage devices.

Here is an example of how a buffer works. Assume the following conditions: you have created a file with logical records of 9 bytes each, and you want to access 20 of these records in a short program segment.

Without buffering, the BASIC system would have to make 20 different accesses of a mass storage device to obtain the information. And each time an item is requested from the mass storage device, the BASIC system would get a whole block (1 024 bytes) of information, since that is the smallest unit of data that the HP-UX file system can address. Considering the possibility that all of these items might all be located in the same 1 024-byte block, the system would, in this case, be getting about 100 times the information it needs in each of 20 separate mass storage accesses.

With buffering, the BASIC system loads a physical record of information from the mass storage device, and then extracts from that record the information it needs. In our example, if all 20 logical records are in the same mass storage block, the computer only has to make 1 mass storage access; it then can extract each logical record *from the buffer*. Overall, in this particular example, the amount of disc access has been reduced by a factor of 20, and the information flow has been reduced by a factor of about 2000 (=100*20).

This example is *not* necessarily representative of how much mass storage wear and access time can be saved by buffering, but it does make the point that buffering is generally a good technique to use.

File buffers are *automatically* sent to the mass storage device (while writing) at the following times:

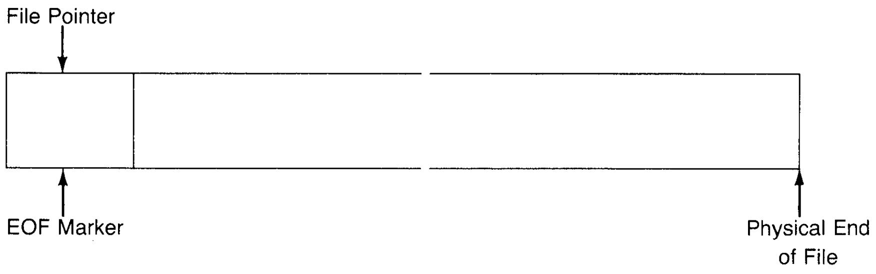
- Whenever the buffer gets full, or when data items in another block are accessed.
- When the file is closed (or when the file number is reassigned).
- When the program is halted (i.e., when PAUSE, STOP, or END is executed).
- When program execution is interrupted (by an event that is set up to cause an "event-initiated branch", as described in the "Program Structure and Flow" chapter).
- When a PRINT# statement is executed *from the keyboard*.

A Closer Look at Serial Access

Serial access is used when a quantity of data is to be stored sequentially in a file and then read back in the same (sequential) order. With this type of access, the file itself is the smallest addressable unit of storage. This is true even if the file being accessed consists of more than one logical record, because the data items are stored and retrieved without regard to logical record divisions (during serial access).

Serial Write Operations When a file is opened, the file pointer is placed at the beginning of the file.

```
ASSIGN# 1 TO "BudgetData"
```

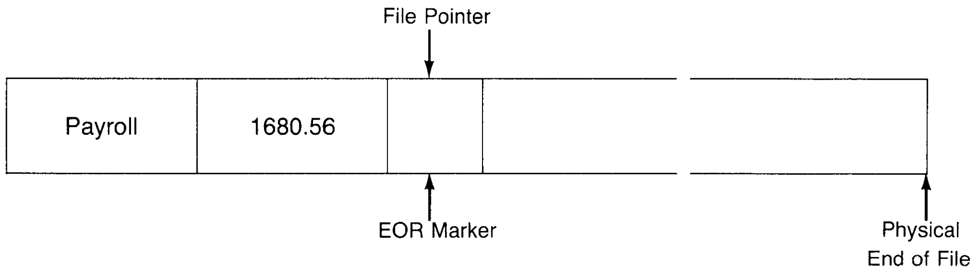


The drawing shows that the file initially contains an end-of-file (EOF) marker at the beginning of the file. Actually, the file is entirely full with EOF markers at the point the file is created.

When a PRINT# statement writes data into the file (through the buffer assigned to the file), the data items are sent one at a time, from left to right in the list, starting at the location indicated by the file pointer. As each item in the data list is stored, the pointer is updated to point to the next available location. When all items in the list have been recorded, the file pointer points at a location just past the end of the recorded data. An end-of-*record*[1] (EOR) marker indicates the position of the last recorded data item.

The location of the file pointer is the point at which a subsequent PRINT# statement will begin.

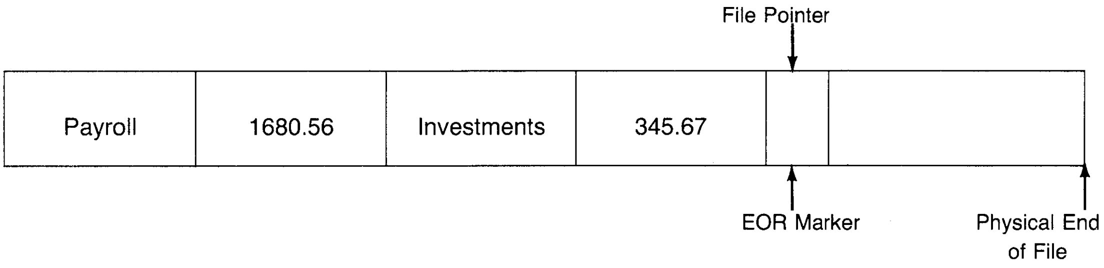
```
PRINT# 1;IncomeName$(1),TargetIncome(1)
```



1 An EOR marker is used instead of an EOF marker, because you can randomly and serially access a file.

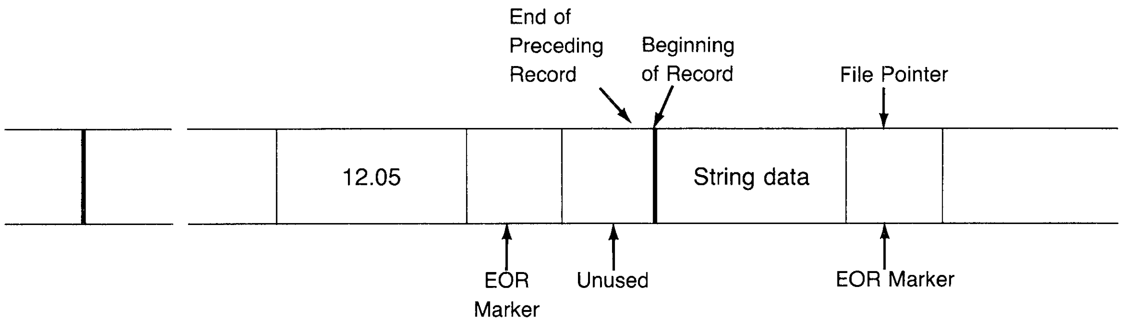
Execution of a subsequent PRINT# statement to the same buffer records the items in the corresponding data list beginning at the current file pointer. The system overwrites the existing EOR marker, writes the items (and corresponding type fields), and then writes another EOR marker at the end of this newly recorded data.

```
PRINT# 1;IncomeName$(2),TargetIncome(2)
```



Earlier in the chapter, it was stated that serial writing essentially ignores logical record boundaries. Here is what actually happens when a serial PRINT# statement crosses a logical record boundary.

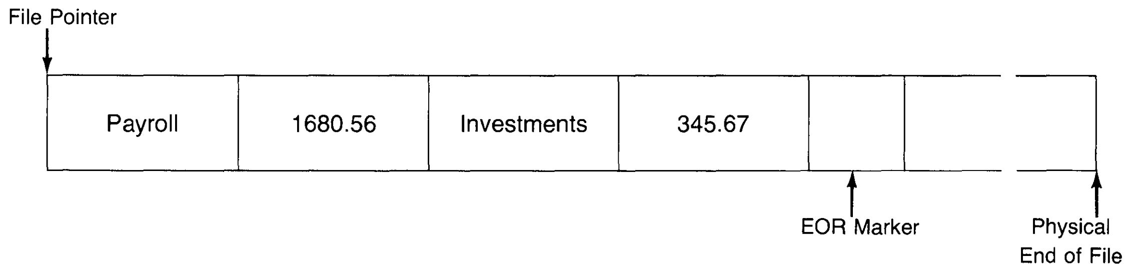
```
PRINT# 1;12.05,"String data"
```



In the above example, there was enough space left in the current logical record to store the numeric item, so it was written. However, there was not enough space to store the string item (at least 4 bytes is required), so an EOR marker was written into the record. The file pointer was then placed at the beginning of the next logical record, and the string item was written. The file pointer is left at the location following the string item. (The *only* situation in which an EOR is not written into the logical record is when there is *exactly* enough room for a numeric item at the end of the record.)

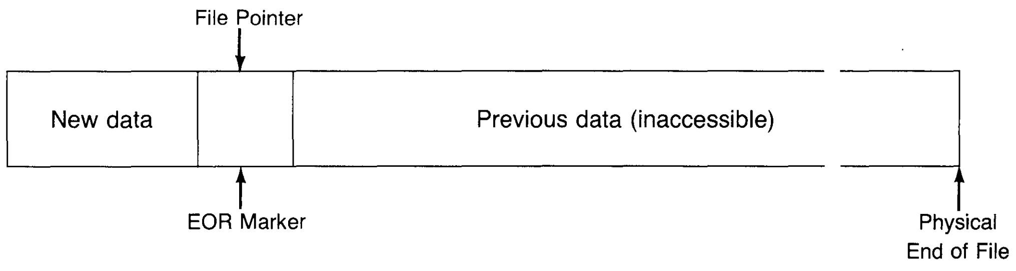
The pointer will continue to move sequentially through the file as shown in the preceding examples, unless moved in another manner. For instance, executing an ASSIGN# statement on the same buffer number moves to the file pointer to the beginning of the file.

```
ASSIGN# 1 TO "Income84"
```



The movement of the file pointer and EOR marker influence the way in which the serial files are updated. For instance, if the pointer is reset to the beginning of the file (as in the preceding ASSIGN# statement) after serially reading a long list of data items, then a subsequent serial PRINT# statement will record new data items over the previous ones. In addition, an EOR marker is placed at the end of the new data items, so the result is that all previous data in the file is inaccessible.

```
PRINT# 1;"New data"
```

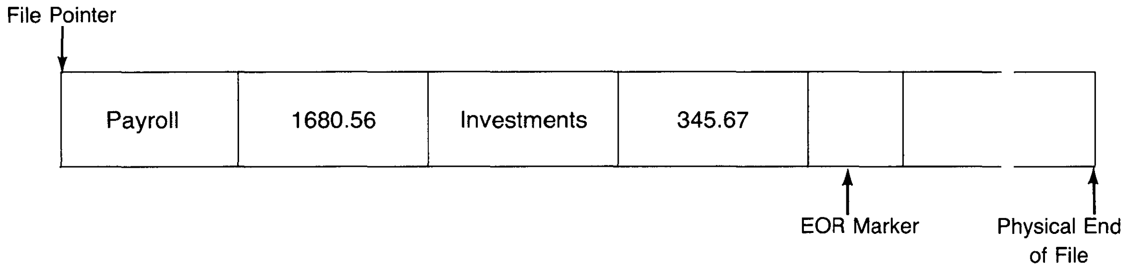


Extending Serial Files These examples do not show that Technical BASIC files are *extensible*. That is, if you create a file and then attempt to serially write past the current *physical* end-of-file (not just past an EOR or EOF marker), then the system will automatically extend the file for you. Each extension is either one block or one logical record in length, whichever is *greater*.

Serial Read Operations Data that has been stored in a data file must be retrieved (i.e., read back into computer memory) before it can be used by the program. Reading data from a file transfers a copy of the data through a buffer in computer memory.

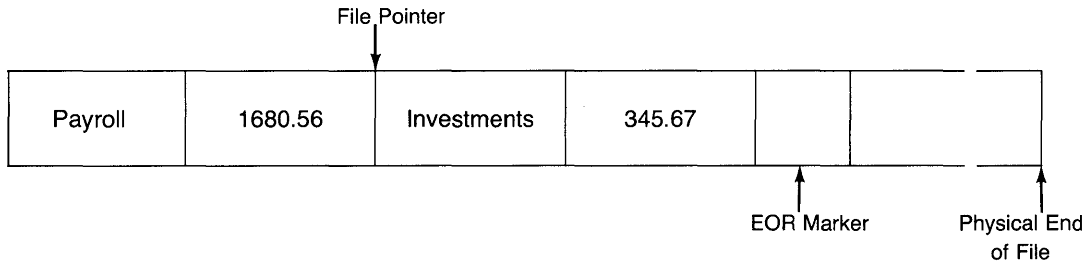
When a file is opened, the file pointer is placed at the beginning of the file.

```
ASSIGN# 1 TO "Oct84Income"
```



Serial reading is accomplished by the READ# statement; items in the data list are filled from left to right. As each data item is retrieved, the file pointer is updated to point to the next data item in the file. Items are accessed sequentially, *ignoring* any logical record boundaries.

```
READ# 1;IncNam$(1),TstInc(1)
```



The variables used to read the data in the file must be of the same *general* data type as the data item (i.e., numeric or string), but they need not be of the same *specific* type (i.e., INTEGER, SHORT, or REAL for numeric items; or identical length for string items). However, matching specific types always works best because it prevents value range errors.

If a READ# statement attempts to read past an EOF marker, an error is reported. You can trap these errors with the ON ERROR statement. See the subsequent section called "Trapping EOF and EOR Conditions" for further details.

Both data stored serially and data stored randomly can be retrieved serially.

Random File Access

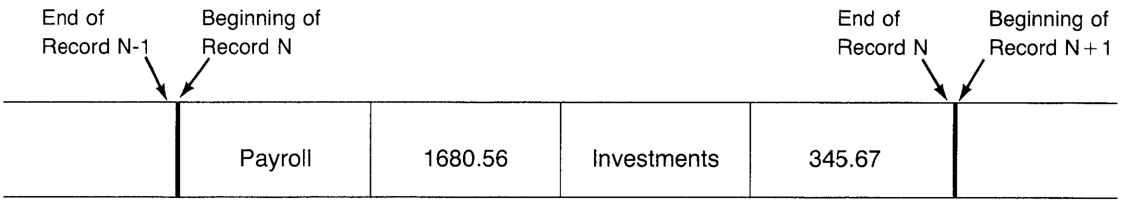
Random access allows you to move the file pointer to the beginning of any logical record within a file. This is in contrast to only setting the pointer to the beginning of a file for serial access, and then sequentially reading data items from the file and having the file pointer be updated automatically by the system. However, random access is like serial access after moving the pointer to the beginning of a logical record, because you will then serially access the data *in that record*.

Random Writing Here is an example of creating a file with 12 logical records: each one contains target incomes (names and values) for a month of the year.

```
100  OPTION BASE 1 !  Lower bound of array subscripts.
110  DIM IncomeName$(2)
120  REAL TargetIncome(2)
130  !
140  IncomeName$(1)="Payroll"
150  IncomeName$(2)="Investments"
160  !
170  TargetIncome(1)=1680.56
180  TargetIncome(2)=345.67
190  !
200  ! Create and open a file.
210  CREATE "TgtInc84",12,42
220  ASSIGN# 1 TO "TgtInc84"
230  !
```

```
240  FOR Month=1 TO 12
250    PRINT# 1,Month !  Move pointer to start of record (random "seek").
260    FOR Category=1 TO 2
270      PRINT# 1;IncomeName$(Category),TargetIncome(Category) ! Serial wrt.
280    NEXT Category
290  NEXT Month
300  !
310  END
```

Here is a conceptual drawing of what is in *each* logical record.



Here are the differences between serially and randomly writing to files.

- In order to randomly write to a data file, you must use a PRINT# statement that specifies a record number.

```
200 PRINT# 1,3;Str$,Intgr  ! Write 2 data items in record 3.
```

- When a random PRINT# statement is executed, the file pointer is moved to the beginning of the specified record. The data items in the PRINT# statement are then recorded in the record, and an end-of-record (EOR) marker is placed after the last item (if there is at least 1 byte left in the record).
- If you want to merely position the file pointer at the beginning of a record, without writing any data, then execute a PRINT# statement specifying only the record number (omitting the data list).

```
PRINT# 1,5
```

- Record divisions are **not** ignored, as they were in serial access. Thus, if you attempt to store more data in one logical record than that record will hold, an EOR error is reported:

  ```
  ERROR 69 : RANDOM OVF
  ```

  or

  ```
  ERROR 72 : RECORD.
  ```

Randomly Reading Here is an example of reading the data that was stored using random access methods. Note that the logical records are accessed in reverse order (12, 11, 10, ..., 1).

```
100  OPTION BASE 1
110  DIM IncomeName$(2)
120  REAL TargetIncome(2)
130  !
140  ! Open the file.
150  ASSIGN# 7 TO "TgtInc84" !  Buffer # 7.
160  !
170  FOR Month=12 TO 1 STEP -1 !  Access records in reverse order.
180    READ# 7,Month !  Move pointer to start of record (random "seek").
190    DISP "Month:";Month
200    DISP "---------"
210    FOR Category=1 TO 2
220      READ# 7;IncomeName$(Category),TargetIncome(Category) ! Serial read.
230      DISP "Income name:",IncomeName$(Category)
240      DISP "Target income:",TargetIncome(Category)
250      DISP
260    NEXT Category
270  NEXT Month
280  !
290  END
```

Here are the results of running the program.

```
Month: 12
---------
Income name:          Payroll
Target income:         1680.56

Income name:          Investments
Target income:         345.67
```

```
Month: 11
---------
Income name:           Payroll
Target income:          1680.56

Income name:           Investments
Target income:          345.67

.
.
.

Month: 1
---------
Income name:           Payroll
Target income:          1680.56

Income name:           Investments
Target income:          345.67
```

Randomly reading files is slightly different from serially reading files.

- In order to read data from a "random" record of a data file, you must use a READ# statement that specifies a record number.

```
200 READ# 1,3;A$,I ! Read 2 data items from record 3.
```

When a record is specified, the file pointer is moved to the beginning of that record. The data item(s) in the READ# statement are then transferred serially (through the buffer) into the specified variable(s).

- Logical record boundaries are **not** ignored. If you attempt to read more data items than are in the record, an EOR error will be reported (`ERROR 72 : RECORD`).
- If you want to merely position the file pointer at the beginning of a record without reading any data, then execute a READ# statement specifying only the record number (omitting the data list).

```
READ# 1,3
```

As with serial reading, the variables used to read the data in the file must be of the same *general* data type as the data item (i.e., numeric or string), but they need not be of the same *specific* type (i.e., INTEGER, SHORT, or REAL for numeric items; or identical length for string items). However, matching specific data types is always best, because it eliminates the potential for value range errors.

Determining Data Types

A preceding section mentioned that each item written in a data file is preceded by a type field. You can use the TYP function to read this field and thereby determine the item's data type.

This function allows you to avoid errors such as attempting to read a string data item into a numeric variable. It also allows you to determine whether the file pointer is pointing at the current end-of-file (EOF) or end-of-record (EOR) marker.

Data-Type Field Values

Here is an example of using the TYP function:

```
ItemType=TYP(1)
```

The function reads the type field of the item at which the file pointer is currently pointing. The parameter passed to the function specifies which buffer is to be read. The preceding statement determines the type of the item at the current location of the file pointer in buffer number 1. An example program is shown below.

Here is the range of integer values that the TYP function can return, and the corresponding data types.

TYP Value	Data Type
1	Numeric
2	Full string
3	End-of-file marker
4	End-of-record marker
8	Start of string
9	Middle of string
10	End of string

Sensing EOF and EOR Conditions

Here is a simple example of using the TYP function to determine whether the file pointer is currently pointing at an EOF marker.

```
100  DEF FNEOF(BuffNo) = TYP(BuffNo)=3
```

Here is an example of using the function:

```
200 WhileNotEOF: IF NOT FNEOF(2) THEN ReadItem ELSE EndOfFile
```

Sensing an EOR marker is almost identical.

```
110  DEF FNEOR(BuffNo) = TYP(BuffNo)=4
```

Here is an example of using the function:

```
200  IF NOT FNEOR(2) THEN ReadItem ELSE NextRecord
```

Trapping EOF and EOR Conditions

There are certain conditions that you can encounter while writing and reading files that will generate an error. This section describes them.

The following operations will generate an end-of-file (EOF) or end-of-record (EOR) error condition:

- Attempting to read past either an EOF marker or the physical end of file (`ERROR 71: EOF`).
- Attempting to read more data items than there are in a logical record during a *random* read operation (`ERROR 72: RECORD`).
- Attempting to write more data than will fit in a logical record during a *random*[1] write operation (`ERROR 69: RANDOM OVF`).

1 This error is only reported during random writes, because attempting to write past the physical end of file during a *serial* write causes the system to *automatically extend* the file.

Here is an example of using the ON ERROR mechanism to trap EOF errors while reading a file. The file is assumed to contain only string data.

```
100  DIM StringData$[65530]
110  !
120  Ask: DISP "Enter file name." @ INPUT File$
130       DISP "Is this correct?  '"&File$&"'  (Y/N)" @ INPUT Ans$
140       IF UPC$(Ans$[1,1])<>"Y" THEN Ask
150       !
160  ASSIGN# 2 TO File$ !  Open specified file.
170  !
180  ON ERROR GOTO ErrorTrap !  Set up branch for errors.
190  !
200  ! Loop until EOF (or other error).
210 NextItem: READ# 2;StringData$ !  Read as string; if error,
220                                !  branch to ErrorTrap.
230             DISP StringData$
240           GOTO NextItem
250           !
260 ErrorTrap: IF ERRN=71 THEN DISP "End of file found." @ GOTO Ask
270              ! ELSE ERRN<>71, so display error message.
280              ERRM
290  END
```

The program runs until either an EOF error (71) or another error is encountered. When an EOF is encountered, the message `End of file found.` is displayed, and the program asks for another file name. When another error is encountered, the system's normal error message is displayed. You can easily expand the ErrorTrap routine to respond to other file-related errors.

Using text/data Files

This section briefly describes how to write and read files of type `text/data`. This type of file provides a method of interchanging data files between Technical BASIC and the HP-UX system (they are HP-UX "ASCII" files).

Accessing this type of file with a C program is described in the "Examples of File I/O" section near the end of the "Binary Programs" chapter.

Writing to a text/data File

This program shows how to open and write data into a file of type `text/data`. If the file does not already exist, the BASIC system will create it for you. Numeric and string data items are then written into the file.

```
100 INTEGER IntVar
110 IntVar=32000
120 !
130 SHORT ShortVar
140 ShortVar=3e+031
150 !
160 REAL RealVar
170 RealVar=1e+308
180 !
190 DIM StringVar$[20]
200 StringVar$="This is a string."
210 !
220 ASSIGN 14 TO "text_file" !       Assign a file selector.
230 !
240 OUTPUT 14 ; IntVar;ShortVar !    Write two values into file.
250 OUTPUT 14 ; RealVar;StringVar$ ! Write two more values into file.
260 !
270 ASSIGN 14 TO "*" !               Close file.
280 !
290 END
```

Note that the items specified in the OUTPUT statement are written according to the rules of the OUTPUT statement; see the *HP-UX Technical BASIC I/O Programming Guide* or the *Technical BASIC Reference Manual* for details.

In this example, the items in the OUTPUT statements are separated by semicolons; therefore, the items will **not** be separated (in the output data stream) by an end-of-line (EOL) sequence, which is normally a carriage-return followed by a line-feed (control characters). However, the EOL sequence is automatically sent after the last item in the OUTPUT statement (unless suppressed with a semicolon or comma).

Note also that the OUTPUT statement does not put end-of-record (EOR) or end-of-file (EOF) markers in the file.

Reading from a text/data File

This example reads the data from the `text/data` file written with the preceding example. It uses this general rule: *read the file in the same way that it was written.*

```
100 INTEGER IntVar
110 IntVar=-1
120 !
130 SHORT ShortVar
140 ShortVar=-1
150 !
160 REAL RealVar
170 RealVar=-1
180 !
190 DIM StringVar$[20]
200 StringVar$="Initial value."
210 !
220 DISP "Value of IntVar     = ";IntVar ! Show the values BEFORE reading file.
230 DISP "Value of ShortVar   = ";ShortVar
240 DISP "Value of RealVar    = ";RealVar
250 DISP "Value of StringVar$ = ";StringVar$
260 DISP
270 !
280 ASSIGN 14 TO "text_file" !      Assign a file selector.
290 !
300 ENTER 14 ; IntVar,ShortVar !    Read two values from file.
310 ENTER 14 ; RealVar,StringVar$ ! Read two more values from file.
320 !
330 DISP "Value of IntVar     = ";IntVar ! Now show values read FROM FILE.
340 DISP "Value of ShortVar   = ";ShortVar
350 DISP "Value of RealVar    = ";RealVar
360 DISP "Value of StringVar$ = ";StringVar$
370 !
```

```
380 ASSIGN 14 TO "*" !                  Close file.
390 !
400 END
```

Here is the output of the program.

```
Value of IntVar     = -1
Value of ShortVar   = -1
Value of RealVar    = -1
Value of StringVar$ = Initial value.

Value of IntVar     =  32000
Value of ShortVar   =  3e+031
Value of RealVar    =  1e+308
Value of StringVar$ =  This is a string.
```

12 Binary Programs

Most of the time, you will be using the Technical BASIC system to execute programs written in the Technical BASIC language. However, you can also write programs in another language available on the HP-UX system, and then call (execute) the program from Technical BASIC. In this manual, such programs are termed binary programs. The term ''binary'' was probably coined because the programs written in another language and compiled into executable object code cannot be easily read by humans – they look like just a bunch of binary patterns.

Binary programs are useful in the following situations:

- An application is already written in another language, and you don't want to translate it into Technical BASIC code.
- Another language supports a feature that is not available in Technical BASIC, or that runs faster in the other language.

Chapter Contents

This chapter describes how to create binary programs in the C programming language and then call them from Technical BASIC. In order to do so, it will provide examples of both BASIC and C programs.

This chapter contains the following major sections:

- An overview, which includes a complete example and general considerations.
- A section describing details of passing parameters to C binaries.

Overview

This section focuses on two fundamental topics:

- An example of writing a C language binary program and then calling it from Technical BASIC.
- A list of general considerations that you must make when calling binary programs.

The section is intended to *quickly* give you a simple example and then move to a global perspective. More specific details of creating and calling C binaries are presented in the subsequent section.

A Simple Example

Here is an example BASIC program that loads and calls a binary. The binary doubles the integer sent to it.

```
100  LOADBIN "bin1"
110  !
120  INTEGER WholeNumber
130  WholeNumber=7
140  !
150  CLEAR
160  DISP "Before CALLBIN:"
170  DISP "WholeNumber =";WholeNumber
180  DISP
190  CALLBIN "entry_pt" (WholeNumber)
200  DISP
210  DISP "After CALLBIN:"
220  DISP "WholeNumber =";WholeNumber
230  END
```

The BASIC program displays the following information:

```
Before CALLBIN:
WholeNumber = 7


After CALLBIN:
WholeNumber = 14
```

The LOADBIN statement (line 100) links the binary to BASIC. This example assumes that the binary program is in a file in the current working directory. If it is not in a file in that directory, then you would need to specify a path name. Here is an example of an absolute path name:

```
LOADBIN "/users/marka/BASIC/CH12/bin1"
```

The BASIC program then assigns a value to an INTEGER variable (line 130) and then displays the value (lines 160 and 170).

The CALLBIN statement (line 190) branches to the specified entry point in the binary; in this case, the entry point is named `entry_pt`.

After the binary has finished execution, it returns control to the BASIC program. In this example, the BASIC program displays the modified value of the variable WholeNumber. Note that the BASIC variable WholeNumber is passed *by reference*, which allows the binary to modify the variable's value. Passing parameters is further described in subsequent sections.

Once you no longer need the binary program, you can unlink it from BASIC with the SCRATCHBIN statement.

```
SCRATCHBIN "bin1"
```

These steps are all you need to do to in order to use a binary program *that has already been written*.

An Example C Binary

Since the HP-UX system was written in the C programming language, it seems appropriate to show an example C binary. Here is a simple C language binary that doubles the value of an integer that is passed to it.

```
entry_pt(int_var_addr)

int  *int_var_addr;

{
   /* Double the value passed to the routine. */
   *int_var_addr = *int_var_addr * 2;
}
```

You can use the following procedure to enter, compile, link, and call the C binary from Technical BASIC:

1. Enter the C source program, and store it in a file. We'll use the HP-UX vi editor for this purpose. Make sure the vi editor is on-line, and execute the following command:

   ```
   vi bin1.c
   ```

   Your screen should fill with tilde characters; the last line should display the specified file name.

   ```
   ~
   ~
   ~
   ~
   ~
   ~
   ~
   ~
   ~
   ~
   ~
   ~
   ~
   bin1.c [New file]
   ```

   Now press **i** (for "insert") and type in the program exactly as shown below. (Refer to the description of the vi editor in your HP-UX documentation if you have problems while typing it in.)

```
entry_pt(int_var_addr)

int  *int_var_addr;

{
   /* Double the value passed to the routine. */
   *int_var_addr = *int_var_addr * 2;
}
```

   When the entire program is in memory, get out of the insert mode by pressing the **Esc** key. Then store the program by typing ZZ. (Make sure that you type *upper-case* ZZ.) You may want to verify that the file exists by getting a listing of the directory in which the file was stored.

2. Compile the C language source code, but don't generate the normal "a.out" (linked, executable) object file. Instead, specify that the C compiler is to generate a ".o" (unlinked, relocatable) object file. The following C compiler *cc* command, with *c* option, accomplishes this task.

   ```
   cc -c bin1.c
   ```

   You may want to verify that the file named `bin1.o` was actually created.

3. The next step is to link the bin1.o object file. Use this HP-UX command:

   ```
   ld  bin1.o  -r  -d  -o bin1  -lc
   ```

   The *ld* command is the "link editor" command. The *-r* option indicates that the specified object file (bin1.o) is to be loaded as relocatable (re-linkable) code. The *-d* option indicates that it is to be loaded into an area of "common" memory that is accessible to the Technical BASIC system. The *o* option specifies that the object file is to be named "bin1", rather than given the default name "a.out". The *lc* option specifies that the C libraries are to be made accessible to the program.

4. Now enter the Technical BASIC system. You are ready to load the binary so that a BASIC program can call it. Use the LOADBIN statement, specifying the name of the *file* loaded with the preceding *ld* command:

   ```
   LOADBIN "bin1"
   ```

5. Now enter and run a BASIC program that calls the binary. Actually, the BASIC program names the *entry point* in the binary to which BASIC will be transferring control; in this case, the entry point is named "entry_pt".

```
100  LOADBIN "bin1"
110  !
120  INTEGER WholeNumber
130  WholeNumber=7
140  !
150  CLEAR
160  DISP "Before CALLBIN:"
170  DISP "WholeNumber =";WholeNumber
180  DISP
190  CALLBIN "entry_pt" (WholeNumber)
200  DISP
210  DISP "After CALLBIN:"
220  DISP "WholeNumber =";WholeNumber
230  END
```

Running the program should produce the following results.

```
Before CALLBIN:
WholeNumber = 7

After CALLBIN:
WholeNumber = 14
```

6. When you are finished using the binary, you can unlink it from the BASIC system by executing this statement:

```
SCRATCHBIN "bin1"
```

Note that the file is still in the HP-UX file system; however, it is not linked (and is therefore inaccessible) to Technical BASIC.

Summary

The preceding example showed how to create a simple C binary. The important points were as follows:

- The binary was loaded as a relocatable program into the common memory area using the *r* and *d* options of the *ld* command.

- The binary was linked to BASIC by executing a LOADBIN statement that specifies the file name of the relocatable program.
- The BASIC program branched to the desired entry point by using CALLBIN. Parameters may be passed to the binary by including them in the CALLBIN statement. The order and type of parameters must match that expected by the binary.
- The SCRATCHBIN statement unlinked the binary from BASIC.

Additional Considerations

When calling binary programs, you will almost always need to provide them with some sort of data. The binary then performs a pre-defined operation on this data, and often returns some resultant data. Thus, when calling binary programs, you must make several considerations:

- What is the name of the routine to be called (i.e., the entry point) ?
- What sort of data, if any, does it require ?
- How are the data items to be passed – by value, or by reference (address) ?
- What does the routine do with the data ?
- What data will be returned ?
- How will the data be returned ?

Rather than proceed with generalizations, the following section gives specific examples of passing parameters to C language binaries.

12

C Binaries

This section discusses details of passing numeric and string parameters from BASIC to C binaries. If you have trouble understanding the mechanisms of "passing by reference" or "passing by value", then you may want to study further examples of passing parameters in the "Subprograms" section of the "User-Defined Functions and Subprograms" chapter.

Passing Simple Numeric Parameters

There are three BASIC data types: INTEGER, SHORT, and REAL. However, you can only pass two of these types to C programs: INTEGER and REAL. Here is the required correspondence between BASIC pass parameters and C formal parameters:

BASIC Pass Parameter	Corresponding C Format Parameter
INTEGER	int
REAL	double

This BASIC program passes three parameters to the subsequent C binary.

```
100 INTEGER RadiusB
110 RadiusB=10
120 REAL AreaB
130 !
140 LOADBIN "area"
150 CALLBIN "area" (PI,(RadiusB),AreaB)
160 DISP "Area of circle with radius";RadiusB;"=";
170 !
180 END
```

Here is the C binary program.

```
area(Pi,RadiusC,AreaC)

  double  Pi;
  int     RadiusC;
  double  *AreaC;

{
  *AreaC = Pi * RadiusC * RadiusC;
}
```

Here are the results of running the program:

```
   Area of circle with radius 10 is 314.159265358979
```

The first BASIC pass parameter, PI, is passed by value, since it is a numeric function and thereby qualifies as a numeric expression. The corresponding C formal parameter is of type double, since PI is a function of BASIC type REAL.

The second BASIC pass parameter, (RadiusB), is also passed by value since it has been enclosed in parentheses (which makes it an expression). The corresponding C formal parameter is of type int.

The third BASIC pass parameter, Area, is passed by reference since it is a variable which is not part of an expression. The corresponding C formal parameter is of type *pointer to* double, as indicated by the leading *. A pointer variable is one that contains the *address* of the variable, rather than its *value*. This is required because the corresponding BASIC pass parameter is passed "by address" (by reference). Passing a variable by reference allows the binary to modify that variable's contents, thereby allowing parameters to be passed *back to BASIC*.

Passing Numeric Array Parameters

Here is a modification to the preceding example that passes 2 arrays to a C binary.

```
100 INTEGER RadiiB(4) ! 5 elements (OPTION BASE 0).
110 FOR I=0 TO 4
120   RadiiB(I)=I
130 NEXT I
140 !
150 REAL AreasB(4)
160 !
170 LOADBIN "arrays"
180 CALLBIN "arrays" (PI,RadiiB(),AreasB())
190 DISP "Radii  Areas"
200 DISP "-----  -----"
210 FOR I=0 TO 4
220   DISP USING "DD.DD,XX,DD.DD" ; RadiiB(I),AreasB(I)
230 NEXT I
240 !
250 END
```

Here is the C program.

```
arrays(Pi,RadiiC,AreasC)

  double  Pi;
  int     *RadiiC;    /* Pointer to array.      */
  double  AreasC[5]; /* Can also be AreasC[]   */

{
  int i;

  for (i=0; i<5; i++) /* Assume 5 elements in each array. */
    AreasC[i] = Pi * *(RadiiC+i) * *(RadiiC+i);
}
```

Here are the results of running the BASIC program:

```
Radii  Areas
-----  -----
 0.00   0.00
 1.00   3.14
 2.00  12.56
 3.00  28.27
 4.00  50.26
```

Note that arrays are *always* passed by reference. However, also note that the C array declarations in the program use different notation:

```
int     *RadiiC;    /* Pointer to array.      */
double  AreasC[5]; /* Can also be AreasC[]   */
```

These two declarations are *equivalent* in purpose because they each declare a pointer to the first element of an array (i.e., the element with subscript 0). The notation you use in the declaration dictates the notation that you will use in accessing array elements. For instance, individual elements of the RadiiC array are accessed by specifying the subscript: `Areas[i]`. The elements of the RadiiC array are accessed by using pointer expressions: `*(RadiiC+i)`.

This example binary assumes that the calling BASIC program will send an array with at least 5 elements. A more general method would be to pass arrays of variable sizes to the binary. In such cases, the calling program can communicate the size of the array using one of two methods:

- By passing parameter(s) that indicate the number of elements (and dimensions).
- By assigning a unique "flag" value to an array element to indicate that it is the last element in the array.

Passing Simple String Parameters

Passing string parameters from BASIC to C binaries is similar to passing numeric parameters. However, passing string values *back* to BASIC may be somewhat trickier. This section describes both operations.

BASIC strings can be passed either by reference or by value. The following BASIC and C programs illustrate passing string parameters.

```
100 DIM ByRef$[10],ByValue$[10]
110 ByRef$="variable"
120 ByValue$="expression"
130 DISP
140 DISP "ByRef$ before call   = '";ByRef$;"'"
150 DISP "ByValue$ before call = '";ByValue$;"'"
160 DISP
170 LOADBIN "strings1"
180 CALLBIN "strings1" (ByRef$,ByValue$&"")
190 !
200 DISP "ByRef$ after call    = '";ByRef$;"'"
210 DISP "ByValue$ after call  = '";ByValue$;"'"
220 !
230 END
```

Here is the corresponding C binary.

```
strings1(ByRef,ByValue)

char  *ByRef,
      ByValue[10];

{
  int   i;

  /* Assign new value to formal parameter 'ByRef',    */
  strcpy(ByRef,"modified");

  /* Assign new value to formal parameter 'ByValue', */
  for (i=0; ByValue[i]!='\0'; i++)
    ByValue[i] = 'x';

}
```

Here are the results of running the program.

```
ByRef$ before call   = 'variable'
ByValue$ before call = 'expression'

ByRef$ after call    = 'modified'
ByValue$ after call  = 'expression'
```

If a string is passed by value, then it must be treated as an array of type char in the C binary. Note that the value is *not* passed back to BASIC as is the value of the string variable that was passed by reference and modified by the binary.

BASIC strings have a length header that indicates how many characters the string *currently* contains. C strings have no such header; they are instead terminated by the null control character: `\0` in C; `CHR$(0)` in BASIC. *Thus, C binaries cannot modify the BASIC string variable's length.*

For instance, suppose that you pass a string variable (by reference) to a C binary. The binary then proceeds to change the length of the string, but it *does not* modify the BASIC string's length header. Thus upon returning to BASIC, there is no indication that the length of the string variable is any different than when it was passed to the binary.

Here is an example that illustrates this behavior:

```
100 DIM ByRef$[10]
110 ByRef$="variable"
120 !
130 DISP "ByRef$ before call = '";ByRef$;"'"
140 DISP "String length = ";LEN(ByRef$)
150 DISP
160 CALLBIN "strings2" (ByRef$)
170 !
180 DISP "ByRef$ after call  = '";ByRef$;"'"
190 DISP "String length = ";LEN(ByRef$)
200 !
210 END
```

Here is the corresponding C binary:

```
strings2(ByRef)

char  *ByRef;

{
  /* Now make string shorter. */
  strcpy(ByRef,"len=5");
}
```

Here is the program's output:

```
ByRef$ before call = 'variable'
String length =  8

ByRef$ after call  = 'len=5le'
String length =  8
```

The BASIC program sets the variable's length in the assign statement (line 110), and then displays its value and length.

The binary then assigns the string a new value. The new length of this string, according to C, is 5 characters. The binary then returns control to BASIC. Since the BASIC string variable was passed by reference (address), its *contents* are affected by the binary; however, the BASIC variable's *length* is *not* changed accordingly.

The BASIC program displays the string's contents and length. This display shows that only the first 6 characters of the variable were changed: the 5 characters `len=5;` and the null character, `\0`, which is not displayed unless the "display functions" mode is in effect. The BASIC variable's length and the remaining 2 characters, `le`, are not changed.

There are two steps in the general work-around for this type of situation:

1. Before passing the variable (by reference), pad the string with blank characters to the *maximum* length of string that the binary can return. For instance, the following statement pads the BASIC string variable with trailing blanks and sets its length to the maximum (dimensioned) length.

   ```
   ByRef$[LEN(ByRef$)+1]=" "
   ```

   Note that this particular statement will cause an error if the string length is already equal to the maximum (dimensioned) length.

2. After returning to the BASIC program, determine the string's new length.

   a. Search the returned string for a null character, CHR$(0), and then set the string length to 1 less than the position of the null.

   ```
   NullPos=POS(ByRef$,CHR$(0))
   ByRef$=ByRef$[1,NullPos-1]
   ```

   b. Pass a string length parameter (by reference) to the binary. After the binary changes the string's length, it can set the length parameter accordingly and then pass it back to BASIC.

Passing String Arrays

As with numeric arrays, string arrays can only be passed by reference. And the restrictions that apply to simple string variables also apply to string arrays. Here are some examples.

BASIC Declaration & Call	C Declaration

`100 OPTION BASE 1`
`110 DIM StrArray$(5)`
`120 CALLBIN "main"(StrArray$())`
	`char *StringArr;`
	or
	`char StringArr[5][18];`
	or
	`char StringArr[][18];`

`100 OPTION BASE 0`
`110 DIM StrArray$(9)[30]`
`120 CALLBIN "main"(StrArray$())`
	`char *StringArr;`
	or
	`char StringArr[10][30];`
	or
	`char StringArr[][30];`

`100 OPTION BASE 1`
`110 DIM StrArray$(5,10)[20]`
`120 CALLBIN "main"(StrArray$(,))`
	`char *StringArr;`
	or
	`char StringArr[5][10][20];`
	or
	`char StringArr[][10][20];`

`100 OPTION BASE 0`
`110 DIM StrArray$(5,10)[50]`
`120 CALLBIN "main"(StrArray$(,))`
	`char *StringArr;`
	or
	`char StringArr[6][11][50];`
	or
	`char StringArr[][11][50];`

The implications of using the declaration `char *StringArr` versus `char StringArr[]` are the same as for numeric array declarations: the notation you use in the declaration dictates the notation that you must use to specify individual array elements.

Here is an example of how to specify array elements when the first declaration method has been used:

```
str_array1(st_arr)

char  *st_arr;

{
  strcpy(st_arr,"Line a");
  strcpy(st_arr+18,"Line b");
  strcpy(st_arr+36,"Line c");
  strcpy(st_arr+54,"Line d");
  strcpy(st_arr+72,"Line e");
}
```

Here is an example of specifying array elements when the second declaration method has been used:

```
str_array2(st_arr)

char  st_arr[5][18];

{
  strcpy(st_arr[0],"Line a");
  strcpy(st_arr[1],"Line b");
  strcpy(st_arr[2],"Line c");
  strcpy(st_arr[3],"Line d");
  strcpy(st_arr[4],"Line e");
}
```

The implications of using `StringArr[5]` versus `StringArr[]` are that the former specifies the (maximum) size of the array, while the latter allows the size of the array to vary.

Restrictions

One of the few restrictions on binaries is that they cannot performing certain I/O operations. Also, HP-UX environment variables such as TERM and PATH are not available to Technical BASIC binaries. This section provides further information about supported and unsupported I/O operations.

Device I/O Is NOT Supported

Binaries should **not** perform operations such as displaying on the screen (**stdout** in C) or getting characters from the keyboard (**stdin** in C). For example, there is no guarantee that the C standard I/O library function `printf` will work in all binaries, although you may get it to work in some instances.

Examples of File I/O

On the other hand, binary programs can perform general file I/O operations, as long as they open files, manipulate file pointers, and close files *independent* of what BASIC is currently doing. For instance, if a BASIC program currently has a particular `text/data` file open (with ASSIGN), then a binary should **not** access that file. However, once BASIC closes the file, the binary may access it. The converse situation has the same restrictions.

Examples of accessing `text/data` files with BASIC programs are provided at the end of the "Data Storage and Retrieval" chapter.

Calling C Binaries that Access Files

Here are examples of BASIC programs that call C binaries which open and write to a file (or read from it).

Sending a BASIC Array to a Binary The first program passes a file name, a string array, and a parameter indicating the number of array elements to a binary; the binary then writes the data into the specified file. (Note that in this case the BASIC program does not create the file.)

```
100 FileName$="sometext"
110 !
120 DIM StringArray$(10)[79]
130 StringArray$(0)="This is the first  line of text."&CHR$(0)
140 StringArray$(1)="This is the second line of text."&CHR$(0)
150 StringArray$(2)="This is the third  line of text."&CHR$(0)
160 StringArray$(3)="This is the fourth line of text."&CHR$(0)
170 StringArray$(4)="This is the fifth  line of text."&CHR$(0)
180 !
190 LOADBIN "text_write"
200 CALLBIN "text_write" (FileName$,StringArray$(),5)
210 !
220 END
```

Storing the Array in a File Here is a listing of a C binary that writes the string array into the specified file.

```
#include <stdio.h>

text_write(file_name,str_array,nlines)

char  *file_name,
       str_array[][79];
int    nlines;

{

  FILE  *file_pointer,
        *fopen(),
        *fclose();
  int   line;

  /* Open the file for writing.        */
  /* (File must NOT be open in BASIC.) */
  file_pointer = fopen(file_name,"w");

  /* Write the string array into the file. */
  for (line = 0; line < nlines; line++)
    fprintf(file_pointer,"%s \n",str_array[line]);

  /* Close the file before returning to BASIC. */
  fclose(file_pointer);

}
```

Reading an Array with a Binary Here is a BASIC program that calls another binary which reads the data written by the preceding BASIC and binary programs.

```
100 FileName$="sometext"
110 !
120 DIM StringArray$(10)[79]
130 FOR Line=0 TO 4 !     Fill strings with spaces
140                 !     (to set string length)
150   StringArray$(Line)[2]=" "
160 NEXT Line
170 !
180 LOADBIN "text_read"
190 CALLBIN "text_read" (FileName$,StringArray$(),5)
200 !
210 FOR Line=0 TO 4
220   DISP StringArray$(Line)
230 NEXT Line
240 !
250 END
```

The Binary Here is the binary program that reads the file.

```
#include <stdio.h>

text_read(file_name,str_array,nlines)

char  *file_name,
      str_array[][79];
int   nlines;

{

  FILE  *file_pointer,
        *fopen(),
        *fclose();
  int   line,
        i;
  char  c;

  /* Open the file for reading.        */
  file_pointer = fopen(file_name,"r");
  if(file_pointer != NULL)
  /* Then file was opened w/o errors. */
  {
    /* Read the data in the file line by line. */
    for (line = 0;  line < nlines; line++)
    {
      i = 0; /* copy line char-by-char */
      while ((c = getc(file_pointer)) != '\n')
        str_array[line][i++] = c;
    } /* end for */

  } /* end if */
  else
  /* File was not opened, or other error occurred. */
    strcpy(str_array[0],"ERROR");

  /* Close the file before returning to BASIC. */
  fclose(file_pointer);

}
```

13 Graphics

Graphic displays are a powerful tool for presenting information. Computer graphics can be equally powerful but an extra step is required between the conception of the idea and the final image. This step is the construction of a mathematical model of the image within the computer.

Since computers only do what they are told, it is essential to have a complete knowledge of the commands that communicate between the real world and the computer's world. This knowledge is needed to create the model within the computer's memory and to understand the resulting image of the model.

Chapter Contents

This chapter contains the following major sections.

- Raster (screen) graphics
- Limits and scaling
- Plotting and reading bytes on the raster
- External output devices
- Interactive graphics

Raster Graphics

A good place to start learning about graphics by using the graphics raster (screen) on your console or terminal. Although many applications require a plotter to produce the final image on paper, it is easier to develop the model on the display. Then, with minor changes, the image can be sent to an external plotter.

Determining Your Screen's Capabilities

Many console and terminal screens are capable of displaying two rasters (images) either separately or simultaneously.

- The alpha raster displays alphanumeric characters.
- The graphics raster displays pixels (picture elements).

The specific graphics capabilities of your screen are listed in the *Implementation Specifics* appendix for your particular system.

Selecting and Initializing the Output Device

One statement selects which device is to receive graphics output and also initializes the graphics system by setting up default conditions.

```
PLOTTER IS 1
```

The graphics system is now ready for use.

With most systems, the alpha raster is initially on and the graphics raster is off. During execution of a graphics program, the alpha raster may be turned off; however, attempting to enter any text turns the alpha raster back on and leaves it on. This statement turns the graphics raster on.

```
GRAPHICS
```

Whenever you wish to clear the graphics raster, execute this statement:

```
GCLEAR
```

If the alpha raster is also on but is not clear, execute this statement.

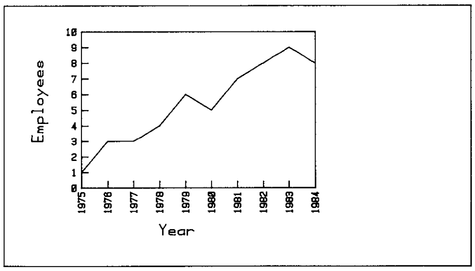
```
CLEAR
```

Executing this statement should draw the current graphics plotting boundaries.

```
FRAME
```

An Example

Here is an example of a graphics image output by a BASIC program.



Here is the program that created the picture.

```
100 PLOTTER IS 1 ! Choose plotter (and initialize it).
110 GCLEAR !       Clear any existing image.
120 FRAME !        Draw line around plotting bounds.
130 !
140 LOCATE 30,100,30,90 !  Define plotting area.
150 FRAME !                Draw new bounds.
160 !
170 SCALE 1975,1984,0,10 !    Scale the plotting area.
180 LAXES 1,1,1975,0,1,1,5 !  Label the axes.
190 !
200 CSIZE 6,0.5 !      Use taller, narrower characters.
210 MOVE 1978,-3 !     Title the horizontal axis.
220 LABEL "Year"
230 !
```

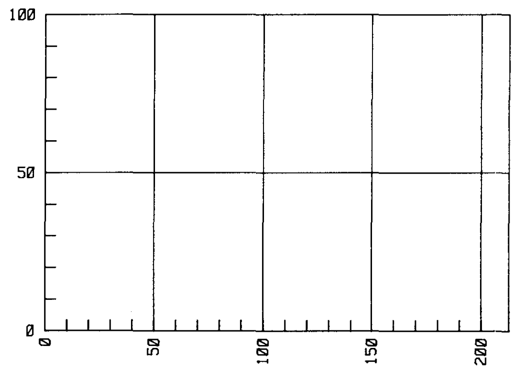
```
240 MOVE 1973,0 !    Title the vertical axis.
250 DEG @ LDIR 90 !  Label direction is up (90 degrees).
260 LABEL "Number of Employees"
270 !
280 DIM Y(10) !    Array with 10 points (OPTION BASE 0).
290 DATA 1,3,3,4,6,6,7,8,9,9,9
300 FOR Point=0 TO 10
310   READ Y(Point)
320   DRAW 1975+Point,Y(Point)
330 NEXT Point
340 !
350 END
```

Coordinate Systems

Since you create a drawing by telling the computer where to draw points and lines, the drawing area must have a coordinate system that allows you to specify the locations of these points and lines. With Technical BASIC, there are several different methods available for setting up a coordinate system for your plotting area.

The Default Coordinates and Scale: Graphics Units (GU's) When the graphics system is initialized (by PLOTTER IS), the default scale is measured in Graphic Units. The origin (location 0,0) is in the lower left corner of the graphics raster. The shorter side of the raster is 100 GU's in length. The number of GU's in the longer side is determined by the aspect ratio: width/height. For instance, if the screen is exactly two times as wide as it is high, then its aspect ratio is 2. Thus the X coordinate of the right bound is 200, while the Y coordinate of the upper bound is 100.

This example shows the default (GU) scaling for the Integral PC's graphics raster, which has an aspect ratio of approximately 2.01.



Aspect Ratio The current plotter's aspect ratio (width/height) is returned by the following BASIC function.

```
RATIO
```

Here is how the RATIO is calculated:

RATIO = width / height = (Xmax − Xmin) / (Ymax − Ymin)

Thus RATIO has no units, since they cancel in the division of width by height.

RATIO can be used to determine the length of the longer side of the plotting area, since GU's are specifically chosen so that the shorter of the plotter's width or height is exactly 100 GU's long. If the height is shorter than the width, then this expression gives the plotting area's width (in GU's).

```
100*RATIO
```

If the width is shorter than the height (indicated by RATIO returning a value less than one), then this expression gives the plotting area's height (in GU's).

```
100/RATIO
```

User Units The first example in this chapter set up a more relevant plotting scale with this statement:

```
170 SCALE 1975,1984,0,10
```

The parameters define the coordinates of left, right, lower, and upper boundaries of the plotting area, respectively. The scaling units set up by this statement are known as User Units, or UU's.

Here is a SCALE statement that uses meaningful variable names to specify the parameters that set up a User Units coordinate system:

```
170 SCALE Left,Right,Bottom,Top
```

The subsequent "Limits and Scaling" section describes both GU's and UU's in greater detail.

Axes and Grids

The AXES statement can be used to draw axes and to put tick marks on the axes. The following example shows the use of all of the parameters available with the AXES statement:

```
AXES XtickSpc,YtickSpc,XLocYAxis,YLocXAxis,Xmajor,Ymajor,Size
```

The XtickSpc and YtickSpc parameters specify the number of units between the tick marks.

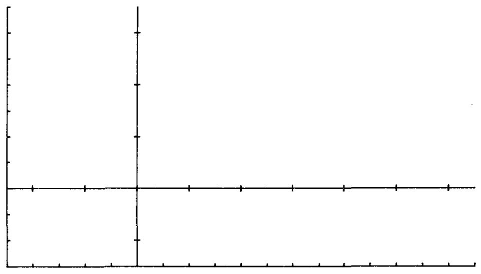
XLocYAxis and YLocXAxis specify the location of the intersection of the axes: XLocYAxis is the X location at which the Y axis crosses the X axis, and YLocXAxis is the Y location at which the X axis crosses the Y axis. If these parameters are not specified, the default cross locations are 0 and 0.

Xmajor and Ymajor specify which ticks are to be "major" (full-size) ticks; all other ticks will be "minor" (half-size) ticks. For example, if Xmajor is set to 4, then every fourth tick on the X axis will be a major tick.

Tick length is determined by the Size parameter. It specifies the size, in GU's, of the major ticks; minor ticks are always half the length of major ticks. If it is not specified, major ticks have a default length of 2 GU's, and minor ticks have a default length of 1 GU.

The following program shows two examples of axes.

```
10 PLOTTER IS 1
20 AXES 10,10 !       10 units between the ticks with origin at (0,0).
30 AXES 20,20,50,30 ! 20 units between the ticks with origin at (50,30).
40 END
```

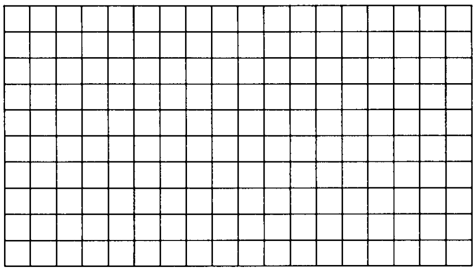


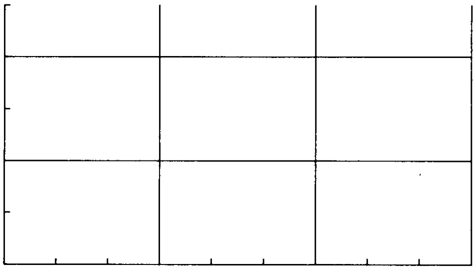
The AXES statement has a related statement: GRID. This statement is best thought of as a pair of axes with very long tick marks. GRID uses the same parameters as AXES, except that the Size parameter specifies the minor tick length (since GRID's "major ticks" span the plotting area).

```
GRID Xtick,Ytick,Xorigin,Yorigin,Xmajor,Ymajor,Size
```

Here are examples of using GRID in a program.

```
100 PLOTTER IS 1
110 GCLEAR
120 !
130 GRID 10,10 !          Grid with X and Y ticks 10 GU's apart.
140 WAIT 5000
150 !
160 GCLEAR
170 GRID 20,20,0,0,3,2,4 ! Ticks 20 GU's apart;
180 !                      origin at 0,0;
190 !                      X major grid every 3rd tick;
200 !                      Y major grid every 2nd tick;
210 !                      Tick length = 4 GU's.
```





Pens and Background

Traditionally, drawing requires pen and paper. With raster graphics, the paper is replaced by the graphics raster, and the pen is replaced by software that turns raster pixels on and off.

Monochromatic Pens On monochromatic graphics rasters, the PEN statement lets you choose between four different pens.

`PEN 1`	white pen (turns pixels on).
`PEN 0`	pen off (does not affect pixels).
`PEN -1`	black pen (turns pixels off).
`PEN -2`	complementing pen (white pixels are changed to black, and black pixels are changed to white).

The preceding examples did not need to specify a pen number, since the default is PEN 1 which draws a white line (on a default background of black).

Clearing to a White Background GCLEAR normally clears a monochromatic raster to a black background (all pixels off). However, you can also use it to clear the raster to a white background (all pixels on) by using this sequence of statements.

```
PEN -2
GCLEAR
```

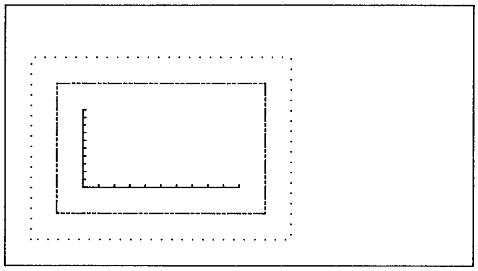
Line Types There are eight different types of lines available with the LINE TYPE statement. Examples are solid, dashed, dotted, and alternating dashes and dots. The *HP-UX Technical BASIC Reference* shows examples of each type.

Two parameters are allowed with the LINE TYPE statement: the line type, and the repeat length. When these parameters are not specified, the line type defaults to 1 (a solid line) and the repeat length defaults to 5 (the pattern repeats every 5 GU's). Thus the default line is LINE TYPE 1,5.

Since FRAME and AXES draw several lines at once, they are useful when exploring the LINE TYPE statement. This program shows a couple of different line types.

13

```
100 PLOTTER IS 1
110 !
120 LINE TYPE 1 !          Defines a solid line.
130 FRAME !                Frame plotting area.
140 !
150 LINE TYPE 3 !          Dotted line.
160 LOCATE 10,110,10,80 !  Smaller plotting area.
170 FRAME !                Frame plotting area.
180 !
190 LINE TYPE 8,10 !       Long dash and two short dashes.
200 LOCATE 20,100,20,70 !  Smaller plotting area.
210 FRAME !                Frame plotting area.
220 !
230 LINE TYPE 1 !          Solid line.
240 LOCATE 30,90,30,60 !   Smaller plotting area.
250 AXES !                 Draw a X and Y axes (in GU's).
260 !
270 END
```



When the graphics raster is used as the plotting device, the repeat factor is system-dependent.

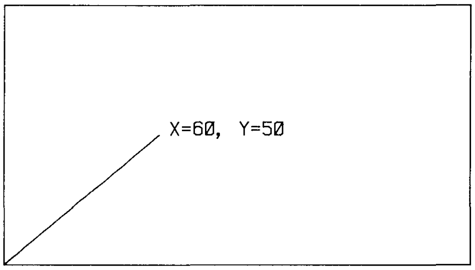
Moving the Pen

Several statements control the movement of the pen on the drawing surface.

`MOVE X,Y`	moves the pen to the coordinate X,Y (without drawing).
`DRAW X,Y`	draws a line from the current pen position to the coordinate X,Y.
`PLOT X,Y,PenCtrl`	moves or draws to point X,Y, as directed by the value of PenCtrl.

DRAW and MOVE do exactly what their names imply. PLOT can do both moves and draws depending on the pen control parameter. Try the following example.

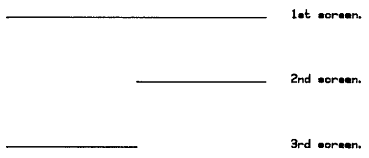
```
100 PLOTTER IS 1
110 GCLEAR
120 FRAME
130 DRAW 60,50
140 LABEL " X=60, Y=50"
```



A white line is drawn from the lower left corner to the middle of the screen. Why from the lower left corner? Because executing the statement PLOTTER IS 1 returns the pen to location 0,0. This is currently the lower left corner of the display.

Execute the following program to see these statements' effects.

```
100 PLOTTER IS 1
110 !
120 PEN 1 !        White pen.
130 GCLEAR !       Clear graphics raster (to black
140 MOVE 0,50 !    Move to left center.
150 DRAW 100,50 !  Draw solid white line.
160 WAIT 3000 !    Wait 3 seconds.
170 !
180 MOVE 50,50 !   Move to center of line.
190 PEN -1 !       Change to black pen.
200 DRAW 0,50 !    Draw over left half of line.
210 WAIT 3000 !    Wait another 3 seconds.
220 !
230 PEN -2 !       Complementing pen.
240 DRAW 100,50 !  Draw over entire line.
250 !
260 END
```



A white line is first drawn across the screen. Then the pen is moved to the midpoint of the line, and a black pen draws over the left half of the line. Finally, the entire line is complemented.

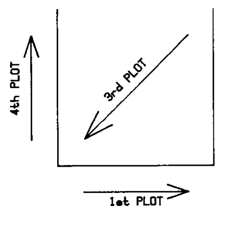
Pen Control with PLOT Pen action is automatically defined for the MOVE and DRAW statements. The pen is always raised *before* a MOVE and lowered *before* a DRAW. The PLOT statement has an optional pen control parameter that determines the pen's action according to the following table.

Control Value	Result
Positive Odd	Move pen, then lower it
Positive Even	Move pen, then raise it
Negative Odd	Lower pen, then move it
Negative Even	Raise pen, then move it

Note that when a positive parameter is used, the pen's up/down status is *not* changed before moving it. For instance, if the pen is currently lowered and a postive pen control parameter is used in a PLOT, then the pen remains down throughout the entire operation; it is not raised before then move and then lowered after the move.

The following example shows controlling the up/down motion of the pen by using the optional pen control parameter.

```
100 PLOTTER IS 1
110 !
120 PEN 1 !          White line.
130 GCLEAR !         Clear (to black).
140 !
150 PENUP !          Make sure pen is rasied.
160 PLOT 20,20,1 !   Move, then lower.
170 PLOT 40,20 !     Draw (since pen lowered).
180 PLOT 40,40,0 !   Draw, then raise.
190 PLOT 20,20,1 !   Move, then lower.
200 PLOT 20,40 !     Draw again.
210 PEN 0 !          Raise pen.
220 !
230 END
```



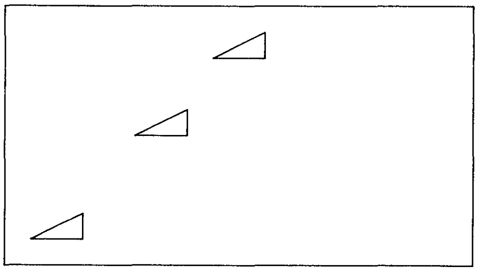
Relative Plotting A second method of moving and drawing involves using a new origin, and specifying pen movements relative to this origin. The relative plot (RPLOT) statement uses the current pen position as a new origin to define a second coordinate system. This new origin is located wherever the last plotting statement (*other than* RPLOT) left the pen. Since RPLOT uses a movable origin, it is useful when drawing a figure that needs to be repeated at different locations on the display.

Here is an example usage of the statement:

`RPLOT X,Y,P` where X and Y are relative displacements.

This program draws a triangle at three different locations.

```
100 PLOTTER IS 1
110 GCLEAR
120 !
130 MOVE 50,50 !   Sets the relative origin.
140 GOSUB Triangle
150 !
160 MOVE 10,10 !   Move the relative origin.
170 GOSUB Triangle
180 !
190 MOVE 80,80 !   Move it again.
200 GOSUB Triangle
210 !
220 STOP
230 !
240 Triangle: ! Draw using relative coordinates.
250   RPLOT 20,10,-1
260   RPLOT 20,0
270   RPLOT 0,0,2 ! Pen up after draw.
280 RETURN
```



Note that the command RPLOT 0,0 returns the pen to the local origin (e.g. 50,50) not the absolute origin (0,0).

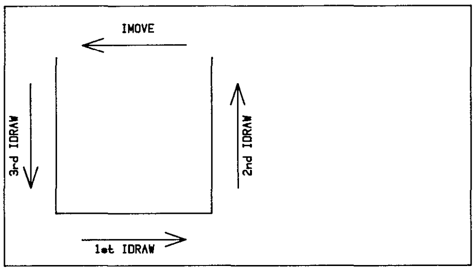
Incremental Pen Positioning It also useful in some situations to have statements that define the pen's *current* location as a new origin. Plotting coordinates are then specified relative to this new origin, which is moved *every* time the pen is moved.

```
IDRAW X,Y
IMOVE X,Y
IPLOT X,Y,P
```

This type of plotting is similar to RPLOT, except that *every* pen movement defines a new origin – including those produced by IDRAW, IMOVE, and IPLOT.

Execute the following program and watch the results.

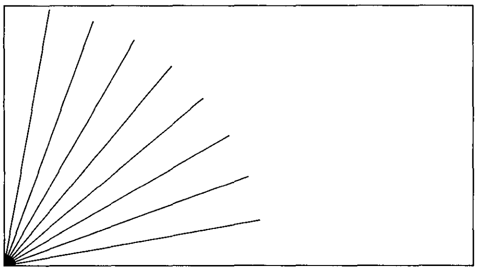
```
10 PLOTTER IS 1
20 !
30 MOVE 40,40
40 IDRAW 30,0 !  Draw right.
50 IDRAW 0,30 !  Draw up.
60 IMOVE -30,0 ! Move left.
70 IDRAW 0,-30 ! Draw down.
```



With each incremental movement of the pen, a new origin is created for the subsequent incremental-plotting statement.

Rotating Incrementally Plotted Lines Lines generated by incremental plotting (IDRAW, IMOVE, and IPLOT) can be rotated by the PDIR (plot direction) statement. The current angle mode is determines how the angle parameter is interpreted; in the following example, the DEG statement specifies that the angle parameter of PDIR is to be interpreted in degrees.

```
100 PLOTTER IS 1
110 GCLEAR
120 !
130 DEG ! Use angular mode of degrees.
140 FOR Angle=0 TO 90 STEP 10
150   PDIR Angle
160   MOVE 0,0
170   IDRAW 100,0 ! Rotated Angle degrees.
180 NEXT Angle
```

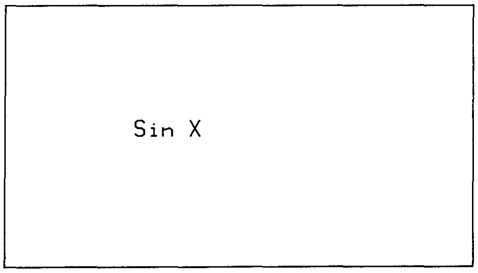


PDIR 0 will return to the system to normal (no rotation).

Labeling the Image

Although images can convey a great deal of information, a few labels help explain what is being presented. The following program places a label on the graphics raster.

```
10 PLOTTER IS 1
20 GCLEAR
30 !
40 MOVE 50,50
50 CSIZE 12 ! Large characters.
60 LABEL "Sin X"
```



LABEL USING allows formatted labels to be plotted just as PRINT USING allows formatted text to be printed.

Several statements affect the printing of labels.

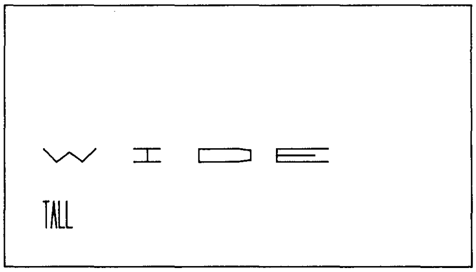
CSIZE controls the character size, aspect ratio, and slant.

LDIR label direction specifies the printing angle of the label.

LORG label origin adjusts the location of the label.

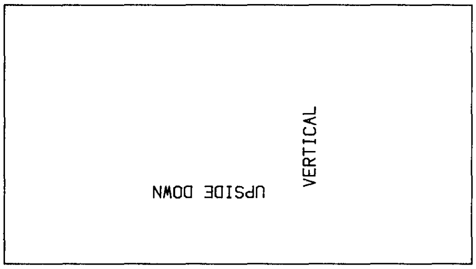
Each of the following examples illustrates one of the above statements.

```
100 PLOTTER IS 1
110 GCLEAR
120 !
130 MOVE 15,40
140 CSIZE 10,4 !  Height 10; aspect ratio 4
150 LABEL "WIDE"
160 !
170 MOVE 15,15
180 CSIZE 20,0.2 ! Height 20; aspect ratio=0.2
190 LABEL "TALL"
```



Label direction is interpreted according to the current angular mode: degrees, radians, or grads.

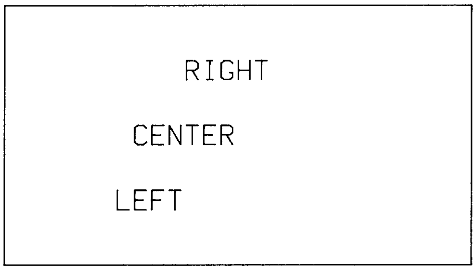
```
100 PLOTTER IS 1
110 !
120 DEG ! Set degrees angular mode.
130 !
140 MOVE 90,10
150 CSIZE 9
160 LDIR 90 ! Label direction is bottom to top.
170 LABEL "VERTICAL"
180 !
190 MOVE 80,10
200 LDIR 180 ! Label direction is right to left.
210 LABEL "UPSIDE DOWN"
220 END
```



There are nine possible label origins used for adjusting the location of the label. This program shows three. (See the *HP-UX Technical BASIC Language Reference* for a description of all nine.)

```
100 PLOTTER IS 1
110 GCLEAR
120 CSIZE 6
130 !
140 MOVE 60,50
150 LORG 1 !     1st "label origin" statement.
160 LABEL USING "K";"LEFT"
170 !
180 MOVE 60,30
190 LORG 9 !     2nd "label origin" statement.
200 LABEL USING "K";"RIGHT"
210 !
```

```
220 MOVE 60,10
230 LORG 5 !     3rd "label origin" statement.
240 LABEL USING "K";"CENTER"
250 END
```



Storing and Retrieving Raster Images

One handy feature of using the graphics raster as a plotting device is that the image can be stored in a file. For instance, this statement stores the current graphics raster in the file named LORG_Raster.

```
GSTORE "LORG_Raster"
```

The statement creates a file of type BASIC/GRAF in the current working directory, and then stores the pixels in the file.

The image can be returned to the graphics raster from a file by this statement:

```
GLOAD "LORG_Raster"
```

The BASIC/GRAF file's contents are loaded back into the raster.

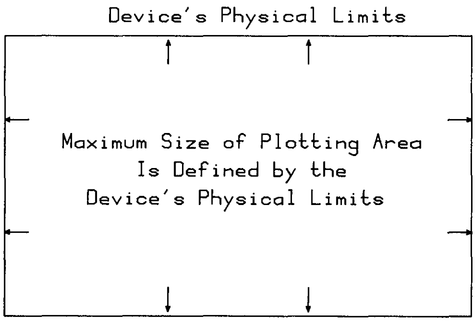
Limits and Scaling

The preceding section gave examples of default Graphics Units (GU) and User Units (UU) scaling. These coordinate systems map into an area known as the plotting area, which can be moved either by software or by a plotter's front-panel controls. This section gives an in-depth treatment of the subject of changing the size of the plotting area and changing coordinate systems.

Physical Limits

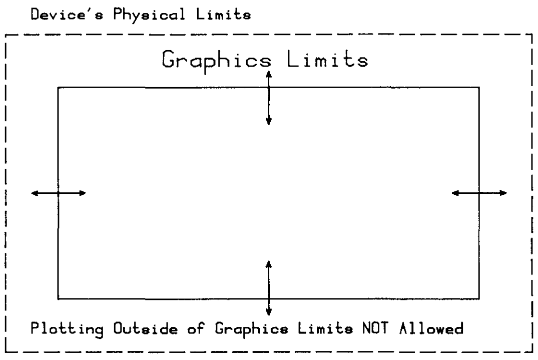
13

The raster display and all other plotting devices have physical limits which **define the maximum size of graphics image** that can be produced on the device. For example, you cannot produce an image on a graphics display screen that is larger than its graphics raster. Similarly, the physical limits of a plotter determine the maximum size of drawing that it can produce.



Graphics Limits

Within the physical limits of a device, you can choose the location of the graphics output by setting the graphics limits. The graphics limits are the boundaries for all[1] graphics output.



Default Graphics Limits The default graphics limits for your console's or terminal's graphics raster can be found in the *Implementation Specifics* appendix for you particular Technical BASIC system.

The default graphics limits vary for different external plotters, but are generally close to the physical limits of the device. Refer to the documentation accompanying the plotter for additional information regarding physical and default graphics limits.

The graphics limits are set to their default locations when the BASIC system is entered.

1 The only exceptions to this statement are the byte-plotting operations performed on a graphics raster, which can performed outside of the current graphics limits (but not outside the physical limits).

Scaling Maps into the Graphics Limits

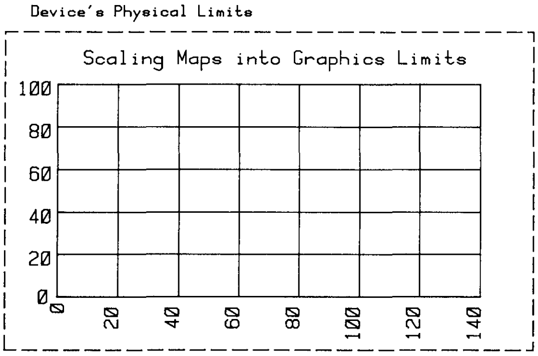
When you want to move the pen, you specify the coordinates to which you want the pen to move. For instance, this statement tells the pen to move to the coordinate 50,50.

```
MOVE 50,50
```

The *physical location* to which the pen moves depends on the coordinate system currently set up for the device. As an example, this sequence of statements moves the graphics limits, sets up a user units (UU's) coordinate system which maps into the graphics limits, and shows the coordinate system.

13

```
LOCATE 20,120,20,80 !      Plotting area bounds (in GU's).
SCALE 0,140,0,100 !        Sets up UU scaling & coordinates.
LAXES 20,20,0,0,1,1,300 ! Draws & labels axes.
```



The X coordinate of the left graphic limit is 0, while the X coordinate of the right limit is 140. The Y coordinates of the lower and upper graphics limits are 0 and 100, respectively.

The length of 1 user unit in the X direction is determined by the difference in X coordinates of the graphics limits (here 140) divided by the physical distance between right and left graphics limits (depends on the plotting device). The length of 1 user unit in the Y direction is determined similarly.

This is only a brief look at scaling. However, it does show how the coordinate system is mapped onto the physical plotting device. More details of scaling methods are shown in subsequent sections.

Moving the Graphics Limits

On most graphics devices, the graphics limits can be moved either manually or with the LIMIT[1] statement. In either case, these limits are read by the BASIC system when it executes a PLOTTER IS statement.

```
LIMIT Xmin,Xmas,Ymin, Ymax
```

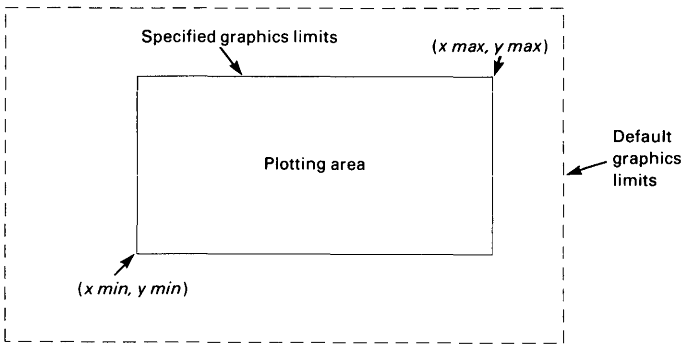
13

Since the LIMIT parameters are **in millimeters**, they specify the **absolute** locations of the graphics limits. The origin (0,0) is normally the lower-left physical limit of the plotting device. X coordinates increase as you move toward the right physical limit of the plotting area, while Y coordinates increase toward the top physical limit. LIMIT enables you to move the graphics limits anywhere within the physical limits of the plotting device.

The following program shows an example of default and user-defined graphics limits:

```
10 ! *** Limit ***
20 PLOTTER IS 1 !             Specifies the CRT as the plotter.
21 !
30 GCLEAR !                   Clears the graphics display.
40 LINE TYPE 5 !              Specifies dashed line type.
50 FRAME !                    Frames the default plotting area for
51 !                          reference.
60 LIMIT 30,30+80,20,20+40 !  Specifies an 80 mm X 40 mm plotting
61 !                          area that is offset from the displays
62 !                          lower-left physical bounds.
70 FRAME !                    Frames the specified plotting area.
80 END
```

1 In addition to specifying the graphics limits, executing the LIMIT statement also activates the graphics default conditions. See the *HP-UX Technical BASIC Reference* for further details about default graphics conditions.



Scope of LIMIT Statements As demonstrated by the program, the LIMIT statement overrides any previously set or default graphics limits. These graphics limits remain in effect until one of the following operations is performed:

- Another LIMIT statement is executed.
- A PLOTTER IS statement is executed.
- The BASIC system is exited and re-entered.

If you do not execute a LIMIT statement in a program and your plot turns out smaller than you expected, then the plotting device is probably using the graphics limits set by a previous LIMIT statement.

Range of Graphics Limit Parameters The ranges of the LIMIT parameters are determined by the current PLOTTER IS device's physical limits. For the graphics raster of your console or terminal, the range of LIMIT parameters are supplied in the *Implementation Specifics* appendix for your particular Technical BASIC system. For external plotters, the range is given in the documentation supplied with the plotter.

If a LIMIT statement parameter is out-of-bounds, the system returns an error message and ignores the statement.

Another Look at the Ratio Function

The RATIO function returns a value equal to the ratio width/height. The value of the RATIO function depends on the current graphics limits, which can be set by default, by LIMIT, or manually on the plotting device.

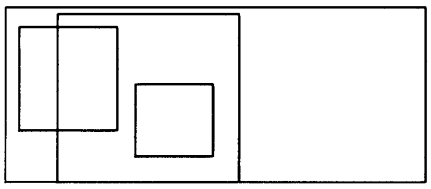
```
10 LIMIT 5,95,10,60 ! New graphics limits (in mm).
20 DISP RATIO !       Width/height=(95-5)/(60-10)=90/50.
30 END
```

The value returned by RATIO is:

```
1.8
```

The RATIO function is useful for changing the size or location of the plotting area, without changing the proportions. A sample program is given below.

```
100 ! *** RATIO ***
120 PLOTTER IS 1 !            Specifies the display as the plotter.
130 GCLEAR !                  Clears the graphics raster.
140 LIMIT 20,90,0,70 !        Specifies the graphics limits.
150 FRAME !                   Frames the plotting area.
160 R=RATIO !                 Assigns RATIO to the variable R.
170 LIMIT 0,R*(60-20),20,60 !Specifies the graphics limits while
180 !                         maintaining the same RATIO as the
190 !                         previous LIMIT.
200 FRAME !                   Frames the plotting area.
210 LIMIT 50,80,10,30/R+10 ! Specifies the graphics limits while
220 !                         maintaining the same RATIO as the
230 !                         previous LIMIT.
240 FRAME !                   Frames the plotting area.
250 END
```



Scaling the Plotting Area

Once the plotting area is defined, either by default or by specifying the graphics limits, the scale can be chosen to suit your particular graphics application. You can use the default scale – graphics units (GU's) – or you can specify your own scale – user units (UU's). If you do not specify your own units, the BASIC system sets UU's equal to GU's.

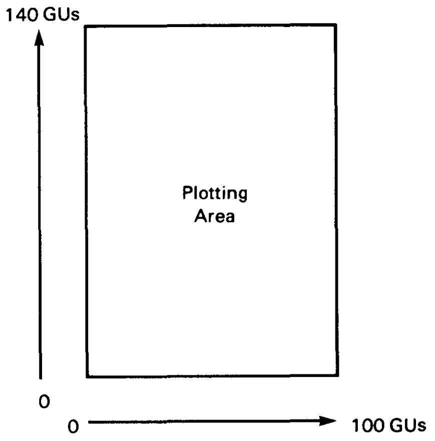
Graphics Units Scale The graphics units scale is the default coordinate system. It remains in effect until a scaling statement is executed.

As mentioned earlier, the BASIC system determines the shortest dimension of the area defined by the graphics limits, and then scales it from 0 to 100 GU's. That is, one GU corresponds to one percent of the shortest side of the rectangle formed by the graphics limits. The other dimension is scaled with the same size units (equal unit, or isotropic, scaling) starting at 0; however, the largest coordinate on the longest side is either RATIO 100 or 100/RATIO, whichever is larger.

The graphics units scale maps onto the area defined by the graphics limits. When the graphics limits change, the size of the graphics units scale also changes. For example, this statement moves the graphics limits to form a 50-by-70 millimetre plotting area:

```
LIMIT 10,60,0,70
```

The graphics unit scale now maps onto this plotting area. The length of one GU is again equal to 1/100 (one percent) of the length of the shortest side of the area bounded by the graphics limits. The length of the longest side of the plotting area is again something greater than 100, depending on the width/height aspect ratio.



The graphics units scale provides a simple method of scaling the plotting area on a percentage basis, regardless of the size of the plotting area.

The following program draws a line from point (0,0), in GUs, to the opposite corner of the plotting area. Enter the graphics limits from the keyboard; the RATIO function is used to compute the length in GUs of the longest side of the plotting area.

```
100 ! *** Graphics Units ***
120 PLOTTER IS 1 !                    The display is the plotter.
125 GCLEAR
130 DISP "Enter LIMIT parameter: xmin,xmax,ymin,ymax"
140 INPUT xmin,xmax,ymin,ymax
150 LIMIT xmin,xmax,ymin,ymax !       Specifies graphics limits.
160 DISP "RATIO = ",RATIO !           Displays current RATIO.
170 WAIT 2000
175 !
180 GCLEAR !                          Clears the graphics area.
190 FRAME !                           Frames plotting area.
200 MOVE 0,0 !                       Moves the pen to lower-left
210 !                                corner.
220 Xmax=100*MAX (1,RATIO) !         Maximum x value in GUs.
230 Ymax=100*MAX (1,1/RATIO) !       Maximum y value in GUs.
240 DRAW Xmax,Ymax !                 Draws a line to upper-right
250 !                                corner.
260 END
```

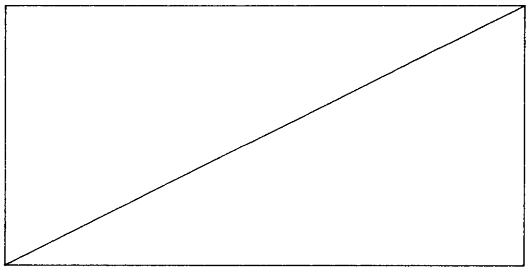
Execute the above program and enter the following data when prompted to do so.

```
10,110,5,55
```

An alpha display of the RATIO is given as the first part of the result.

```
RATIO = 2
```

The final part of the result from executing the program is this graphics display:



User Units Scale There are three scaling statements that allow you to specify the user units scale:

- SCALE – sets up scaling in user units (UU's)
- SHOW – like SCALE, but the units in X and Y directions are equal in length (isotropic scaling)
- MSCALE – sets up scaling in mm units

All three scaling statements specify the scale for the current plotting area (defined by the graphics limits) or by a LOCATE statement (which also specifies plotting boundaries, as described later in this section).

The SCALE statement defines the coordinates of the limits of the current plotting area. The syntax for scale is as follows:

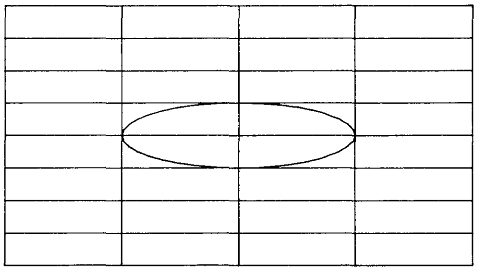
```
SCALE x_min,x_max,y_min,y_max
```

The parameters can be numeric constants, variables, or expressions.

This program shows an example of setting up a user-units coordinate system:

```
100 ! *** Scale ***
110 PLOTTER IS 1 !                The display is the plotter.
120 GCLEAR !                      Clears the graphics display.
130 !
140 SCALE -2,2,-4,4 !             Specifies UU scale.
150 GRID 1,1,0,0 !                Draws a grid with  1 UU spacing.
160 !
170 DEG !                         Sets degrees mode.
180 MOVE 1,0 !                    Moves to the start of the ellipse.
190 FOR Degrees=0 TO 360 STEP 10 !  10 degree increments.
200   DRAW COS(Degrees), SIN(Degrees) ! Draws in UU's.
210 NEXT Degree
220 END
```

The following graphics display is the result of executing the above program.



The SCALE statement specifies user units independently in the X and Y directions.

The SHOW statement specifies user units for a plotting device such that one unit on the X axis is the same length as one unit on the Y axis (isotropic scaling). Thus, the plotting area is scaled with unit squares. The SHOW statement parameters specify the *minimum* number of units to be mapped onto the current plotting area. If necessary, units are added to a dimension to scale the plotting area isotropically (an example is provided subsequently).

The syntax for the SHOW statement is as follows:

```
SHOW x_min,x_max,y_min,y_max
```

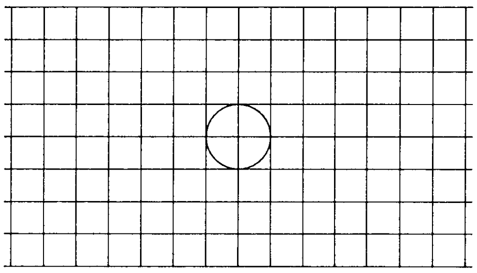
The x_min and x_max parameters specify the minimum bounds in the X direction. The y_min and y_max parameters specify the minimum bounds in the Y direction. The parameters can be numeric constants, variables, or expressions.

To use the SHOW statement, replace the SCALE statement in the previous example with the SHOW statement. Because of equal unit scaling, the figure drawn is now a circle instead of an ellipse. Line 140 of your program should look like this:

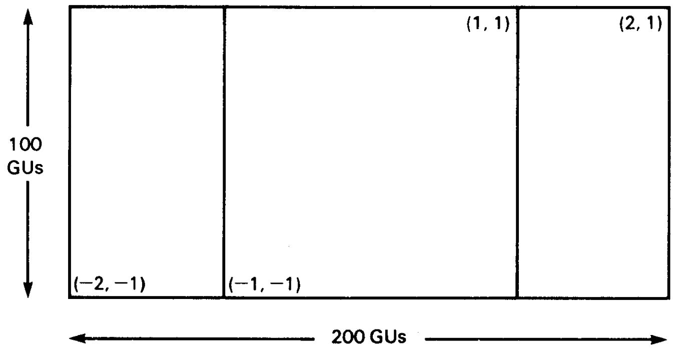
```
140 SHOW -2,2,-4,4
```

and your changed program should be as follows:

```
100 ! *** Scale ***
110 PLOTTER IS 1 !                The display is the plotter.
120 GCLEAR !                      Clears the graphics display.
130 !
140 SHOW -2,2,-4,4 ! <<<------- Specifies 'isotropic' UU scale.
150 GRID 1,1,0,0 !                Draws a grid with  1 UU spacing.
160 !
170 DEG !                         Sets degrees mode.
180 MOVE 1,0 !                    Moves to the start of the ellipse.
190 FOR Degrees=0 TO 360 STEP 10 !  10 degree increments.
200   DRAW COS(Degrees), SIN(Degrees) ! Draws in UU's.
210 NEXT Degree
220 END
```



The SHOW statement sets up UU's such that the coordinate system is as large as possible and is centered within the graphics limits (or within the plotting boundaries, if specified). For example, if the LIMIT rectangle is twice as wide as it is high (e.g., LIMIT 0,100,0,50), then SHOW -1,1,-1,1 is equivalent to SCALE -2,2,-1,1.



Millimetre Scale The MSCALE statement sets millimetres as user units and specifies the location of the origin. MSCALE is useful when correspondence between an image and a physical object is desirable, as in drafting applications. The accuracy of the scale depends entirely on the graphics device in use; for this reason, the MSCALE statement cannot be used to establish millimetre user units for an unsupported peripheral monitor.

The MSCALE statement parameters are different than parameters in the SCALE and SHOW statements.

```
MSCALE x_offset,y_offset
```

MSCALE specifies user units equal to millimetres, and offsets the origin (0,0) in millimetre spacing, from the lower-left graphics limits corner by the specified distance, in millimetres. The MSCALE parameters can be numeric contants, variables, or expressions.

For example, the following statement specifies that 1 user unit equals 1 mm; the origin is offset 10 mm to the right and 15 mm up from the lower-left corner of the plotting area.

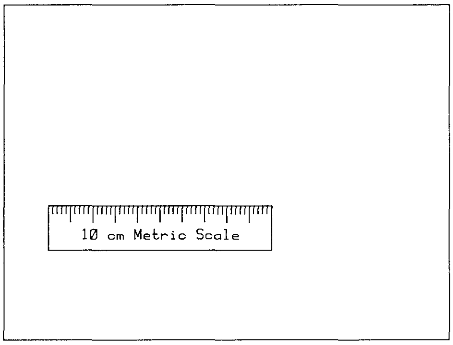
```
MSCALE 10,15
```

Like SCALE and SHOW, the MSCALE statement must follow operations that set or move the graphics limits or the LOCATE plotting boundaries in order to map the user units scale onto the current plotting area.

The following program uses the MSCALE statement to draw a metric ruler on the display.

```
100 ! *** Metric Ruler ***
110 PLOTTER IS 1
120 GCLEAR
130 FRAME
140 MSCALE 10,40 !             Specifies metric user units with
150 !                          10 mm x_offset and 40 mm y_offset.
160 CLIP 0,100,0,10 !          Clips plotting area in millimeters.
170 FRAME !                    Frames the plotting area (ruler).
180 AXES 2,10,0,10,5,10,5 !    Draws the ruler's metric scale.
190 MOVE 30,3
200 LABEL USING "K"; "10 cm Metric Scale"
210 END
```

The results from executing this program are as shown in the following picture:

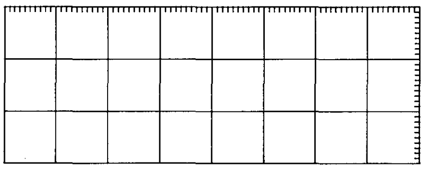


The specified MSCALE origin need not be located inside the graphics limits; you can plot, for example, in negative millimetre units by specifying the origin of your MSCALE beyond the maximum X and Y dimensions of the graphics limits.

The following program draws a metric grid; the MSCALE origin is offset to the upper-right corner of the plotting area.

```
10 ! *** Negative Mscale ***
20 PLOTTER IS 1
30 GRAPHICS !                    Sets the display to graphics mode.
40 LIMIT 0,160,0,60 !            Specifies the graphics limits.
50 MSCALE 160,60 !               Specifies metric UUs and places
51 !                             the origin at the upper-right
52 !                             corner of the plotting area.
60 FRAME !                       Frames the plotting area.
70 GRID 2,2,0,0,10,10 !          Draws a metric grid with 10mm
71 !                             spacing and 2mm tic marks on the
72 !                             x and y axes.
80 END
```

Execution of the previous program results in the following display:



Changing Units: SETGU and SETUU

The two types of units used by the computer in plotting operations are graphics units (GUs) and user units (UUs). The current units mode refers to the type of units in use during plotting. At entry to the BASIC system, the computer is set to user units mode and the current user units scale is GUs, by default. However, you can switch modes at any time and access the current UU's and GU's scales by executing the mode change statements: SETGU, and SETUU.

The SETGU statement sets the system to graphics units mode, establishing GU's as the current scale. Executing SETGU does not disturb the current user units scale, and allows you to plot outside the plotting boundaries set by the LOCATE and CLIP statements (discussion of plotting boundaries appears later in this section). The SETGU statement is the only means by which the computer is set to graphics mode. Unless SETGU is executed, the computer plots according to the current user units scale as defined by SCALE, SHOW, MSCALE, or by default (GU's). The syntax for setting the graphics unit mode is:

```
SETGU
```

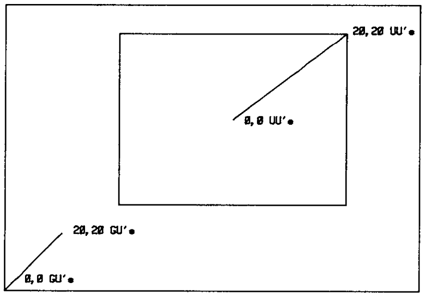
SETUU sets user units (UUs) as the current units mode. User units mode is also set by the SCALE, SHOW, MSCALE, LIMIT, and PLOTTER IS statements, and by default. The syntax for setting the user units mode is as follows:

```
SETUU
```

If the system is set to graphics units mode, you need to return it to user units mode in order for the plotting boundaries set by LOCATE or CLIP to be active. SETGU establishes the area bounded by the graphics limits as the current plotting area.

The following program makes use of both scales: UU's and GU's. The GU's scale is determined by the graphics limits, and is recalled by the SETGU statement.

```
100 PLOTTER IS 1 !        Sets to UU's (=GU's now).
110 GCLEAR
120 FRAME !               Show display limits.
130 CSIZE 3
140 !
150 LOCATE 60,120,20,80 ! Move plotting bounds (GU's).
160 FRAME
170 !
180 SCALE -20,20,-20,20 ! Scale in UU's.
190 MOVE -0,0
200 LABEL " 0,0, UU's"
210 DRAW 20,20
220 LABEL "20,20 UU's"
230 !
240 SETGU !               Change to GU's
250 MOVE 2,2
260 LABEL "0,0 GU's"
270 MOVE 0,0
280 DRAW 20,20
290 LABEL " 20,20 GU's"
300 END
```



Once a scaling statement is executed, the current user-defined scale is active until one of the following occur:

- A new scaling statement is executed (SCALE, SHOW, or MSCALE).
- The system is exited and re-entered, in which case UU's default to GU's.
- A LIMIT or PLOTTER IS statement is executed (UU's set equal to GU's).
- The system is switched to graphics units mode by executing SETGU.

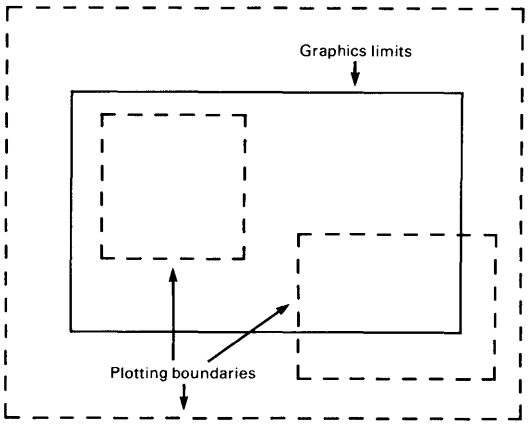
13 Plotting Boundaries

Plotting is restricted either to the default graphics limits or those specified by a LIMIT statement. The LOCATE and CLIP statements specify plotting boundaries. Like the graphics limits, plotting boundaries mark the limits of the plotting area. **However, plotting boundaries differ from graphics limits in that they represent conditional limits.** Plotting ouside the plotting boundaries is possible while the system is set to graphics units (GU) mode or while labeling.

Plotting boundaries can be used for reserving space within the graphics limits for labeling. Plotting boundaries can also be used to create windows for showing portions of a plot.

User units can be mapped onto the LOCATE-defined plotting area but not the CLIP-defined plotting area.

The diagram below shows different ways in which plotting boundaries can be set with respect to the graphics limits. Although the plotting boundaries can extend beyond the graphics limits, or for that matter, the physical limits of the plotting device, you can't plot or label outside the graphics limits.



LOCATE Boundaries The LOCATE statement enables you to relocate the plotting area within the graphics limits by specifying the plotting boundaries. This allows you to leave space for labels outside of the plotting area, but within the graphics limits. The parameters in the LOCATE statement are expressed in GU's. Thus, LOCATE defines the plotting boundaries as a percentage of the graphics limits. The syntax for the LOCATE statement is as follows:

```
LOCATE Xmin,Xmax,Ymin,Ymax
```

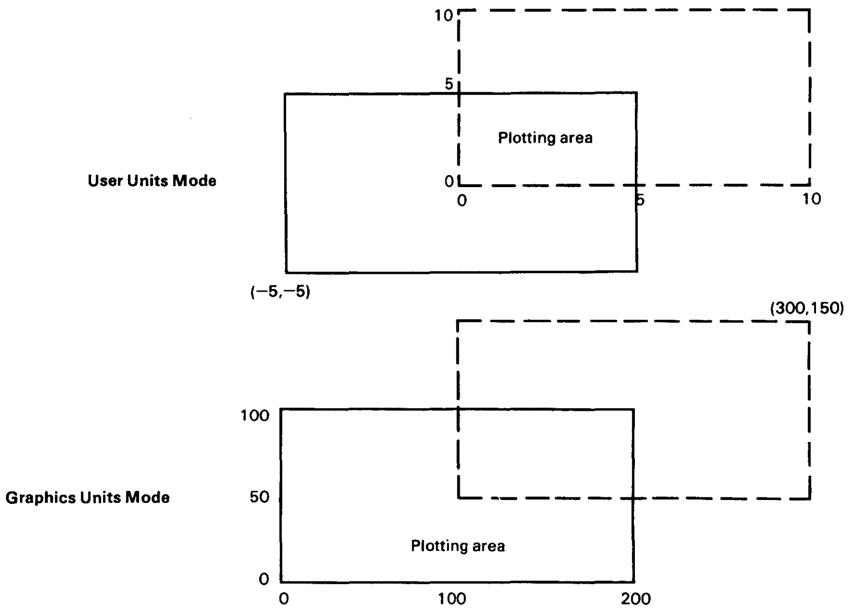
The first two parameters specify the left and right boundaries, and the last two parameters specify the lower and upper boundaries. Like the LIMIT statement, the parameters can be exchanged to reflect the plot (refer to the section of this chapter entitled "Reflecting Plots" for further details). The parameters can be numeric constants, variables, or expressions.

When the LOCATE statement is executed prior to a scaling statement (SCALE, SHOW, or MSCALE), the user units scale is mapped onto the area defined by LOCATE rather than the graphics limits.

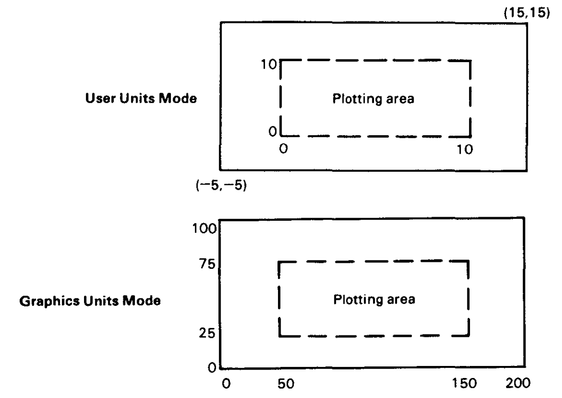
The plotting boundaries specified by the LOCATE statement replace any previously defined plotting boundaries. In turn, the LOCATE-defined plotting boundaries are redefined by the CLIP statement. The LIMIT, UNCLIP, and PLOTTER IS statements default the plotting boundaries to the graphics limits. The plotting boundaries are also set to the graphics limits whenever display memory is reapportioned or the computer is turned on. When the computer is set to graphics units mode by executing SETGU, the graphics limits define the current plotting area. Executing SETUU restores the LOCATE or CLIP-defined plotting boundaries.

The following drawings show the available plotting area and the current scale in user units mode and graphics units mode for the given LIMIT, LOCATE, and SCALE statements. Labeling is allowed anywhere within the graphics limits, regardless of the current units mode. The graphics limits are drawn in solid lines; the plotting boundaries are drawn in dotted lines. The plotting area is the shaded portion.

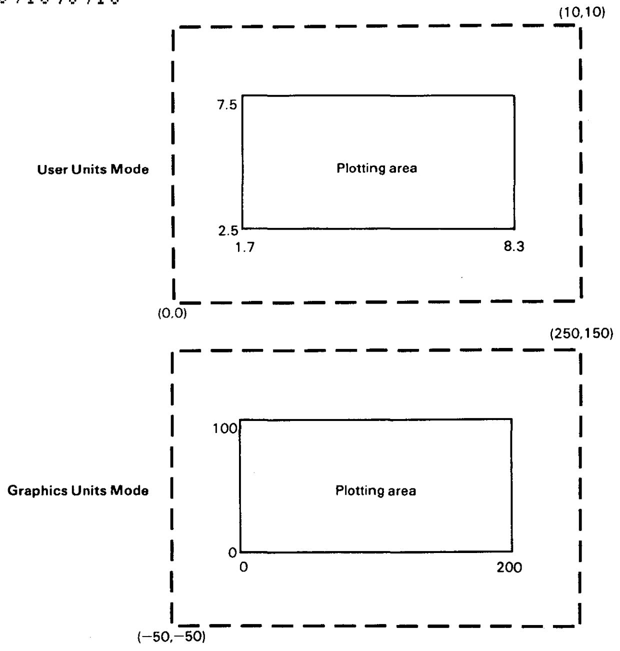
```
LIMIT 0,120,0,60
LOCATE 100,300,50,150
SCALE 0,10,0,10
```



```
LIMIT 0,120,0,60
LOCATE 50,150,25,75
SCALE 0,10,0,10
```



```
LIMIT 0,120,0,60
LOCATE -50,250,-50,150
SCALE 0,10,0,10
```

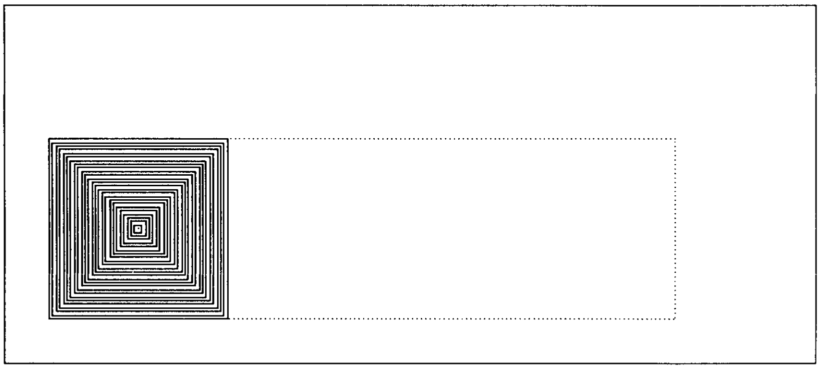


The following program sequentially frames the default graphics limits, the graphics limits specified by a LIMIT statement, and the LOCATE-specified plotting boundaries.

```
10 ! *** Locate ***
20 PLOTTER IS 1
30 GRAPHICS
40 FRAME !                  Frames the default graphics limits.
50 LIMIT 10,150,10,50
60 LINE TYPE 3,2            Specifies a dotted line type.
70 FRAME !·                 Frames the specified graphics limits.
80 FOR I=0 TO 50 STEP 2
90   LOCATE 50-I,50+I,50-I,50+I ! Plotting boundaries in
91 !                              increments of 2 GUs.
100  LINE TYPE 1 !          Specifies a solid line type.
110  FRAME
120 NEXT I
130 END
```

13

Execution of the previous program results in the following display:



CLIP Boundaries The CLIP statement specifies the plotting boundaries according to the currents units: GU's, or UU's. Previously set plotting boundaries are replaced by the CLIP-defined boundaries. Plotting boundaries set by LOCATE or CLIP affect lines plotted in user units mode, but have no effect on lines plotted in graphics units mode or labels. The syntax for the CLIP statement is as follows:

```
CLIP Xmin,Xmax,Ymin,Ymax
```

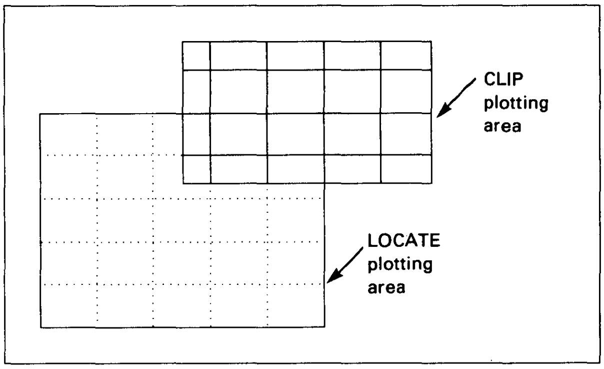
The first two parameters specify the left and right plotting boundaries, respectively, and the second two parameters specify the lower and upper plotting boundaries, respectively. The parameters can be numeric constants, variables, or expressions.

The CLIP parameters are interpreted according to the current units, in contrast to the LOCATE statement which always uses GU's. The plotting area defined by the CLIP statement cannot be scaled by any of the three scaling statements: SCALE, SHOW, and MSCALE. If a scaling statement is executed after the CLIP statement, the user units scale is mapped onto the current plotting area as defined by the graphics limits or, if specified, the LOCATE plotting boundaries.

The graphics units scale is mapped onto the plotting area defined by the graphics limits. For example:

```
10 ! *** Clip ***
20 PLOTTER IS 1
30 GCLEAR
40 FRAME !                  Frames the default plotting area.
50 LOCATE 10,90,10,70 !     Locates the plotting boundaries.
60 FRAME !                  Frames the LOCATE plotting area.
70 CLIP 50,120,50,90 !      Specifies new CLIP plotting boundaries.
80 FRAME !                  Frames the CLIP plotting area.
90 SCALE 0,5,0,5 !          Scales the LOCATE plotting area.
100 GRID 1,1 !              Draws a grid within the CLIP-defined
101 !                       plotting area according to the scale
102 !                       mapped onto the LOCATE-defined area.
110 LOCATE 10,90,10,70 !    Returns plotting boundaries to original
111 !                       LOCATE-defined position.
120 LINE TYPE 3 !           Specifies a dotted line type.
130 GRID 1,1 !              Draws a grid on the LOCATE plotting area.
140 END
```

Execution of the previous program results in the following display:

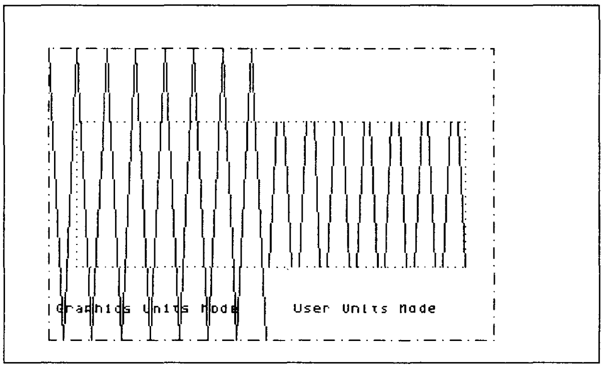


The following program uses the CLIP statement to specify plotting boundaries and demonstrates plotting in graphics units mode and user units mode.

```
10 ! *** Clip-Plot ***
20 GCLEAR
30 PLOTTER IS 1 @ FRAME
40 LIMIT 10,110,5,70 !                  Graphics limits
50 LINE TYPE 6 @ FRAME
60 CLIP 10,RATIO*100-10,25,75 !         Plotting boundaries in
61 !                                    GUs - the current user
62 !                                    units scale.
70 LINE TYPE 3 @ FRAME
80 LINE TYPE 1
90 MOVE 0,100
100 FOR X=5 TO RATIO *100 STEP 5
110  IF X<50*RATIO THEN SETGU ELSE SETUU ! Sets graphics units mode
120  !                                   for left half of plot,
121  !                                   user units mode for right
122  !                                   half of plot. Plotting
123  !                                   scale is GUs for both
124  !                                   modes.
130  IF (-1)^(X/5)=1 THEN Y=100 ELSE Y=0
140  PLOT X,Y,-1
150 NEXT X
151 ! *** Labeling is not restricted by the plotting boundaries ***
152 !
```

```
160 MOVE 3,10 @ LABEL "Graphics Units Mode"
170 MOVE 85,10 @ LABEL "User Units Mode"
180 END
```

Execution of the previous program results in the following display:



Unclipping Plotting Boundaries The UNCLIP statement sets the plotting boundaries equal to the graphics limits, establishing the area bound by the graphics limits as the current plotting area. The syntax of this statement is as follows:

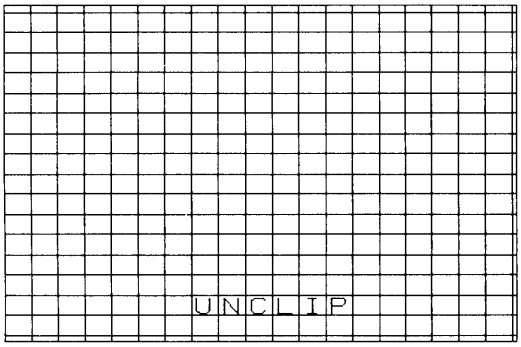
```
UNCLIP
```

UNCLIP doesn't disturb the current units; the system remains in the current scaling units mode. The SETGU statement also establishes the area within the graphics limits as the current plotting area, but without resetting the plotting boundaries. SETGU sets the system to graphics units mode.

The UNCLIP statement is used in the following program to reset the LOCATE-specified plotting boundaries to the graphics limits. The user units scale is preserved.

```
10 ! *** Unclip ***
20 PLOTTER IS 1
30 GCLEAR
40 LIMIT 0,115,0,75 @ FRAME ! Specifies and frames graphics limits.
50 LOCATE 40,120,20,80 @ FRAME ! Locates and frames the plotting
51 !                            boundaries.
60 SCALE 0,10,0,10 !            Scales the LOCATE plotting area.
70 GRID 1,1 !                   Draws a grid on the LOCATE area.
80 CSIZE 9,.9 !                 Specifies character size.
90 MOVE 2,2,-1.9 !              Moves the pen outside the plotting
91 !                            boundaries.
100 LABEL "UNCLIP" !            Labels the character string "UNCLIP".
110 UNCLIP !                    Sets the plotting boundaries equal to
111 !                           the graphics limits.
120 GRID 1,1 !                  Draws a grid on the plotting area
121 !                           bound by the graphics limits; UUs
122 !                           are unchanged.
130 END
```

Execution of the previous program results in the following display:



The following table summarizes the statements and conditions which affect the position and scale of the plotting area.

Condition or Statement	Parameter Units	Effect on Scaling Mode	Effect on Scaling Units	Effect on Graphics Limits	Effect on Plotting Boundaries
Power-on	–	Set to UU's mode.	UU's = GU's.	Set to default graphics limits of the graphics display.	Set to default graphics limits of the graphics display.
`PLOTTER IS`	–	Set to UU's mode.	UU's = GU's.	Read from device.	Set to graphics limits.
`LIMIT`	millimetres	Set to UU's mode.	UU's = GU's.	Set to specified limits.	Set to graphics limits specified by `LIMIT`.
`LOCATE`	GU's	No effect.	No effect.	No effect.	Set to boundaries specified by `LOCATE`.
`CLIP`	current units	No effect.	No effect.	No effect.	Set to boundaries specified by `CLIP`.
`UNCLIP`	–	No effect.	No effect.	No effect.	Resets to current graphics limits.
`SCALE`	UU's	Set to UU's mode.	UU's specified by `SCALE`.	No effect.	No effect.
`SHOW`	UU's	Set to UU's mode.	UU's specified by `SHOW` (in equal x & y units).	No effect.	No effect.
`MSCALE`	millimetres	Set to UU's mode.	UU's specified by `MSCALE` in millimetres.	No effect.	No effect.
`SETGU`	–	Set to GU's mode. Plotting area is defined by graphics limits.	GU's	No effect.	Temporarily set to graphic's limits.
`SETUU`	–	Set to UU's mode. Plotting area is defined by plotting boundaries.	Current UU's as specified by the above statements and conditions.	No effect.	Restores plotting boundaries.

Reflecting Images

The normal sequence of parameters in the LIMIT statement puts the origin of your graph in the lower-left corner of the graphics output. By changing the order of parameters, you can produce a reflected image of the plot (except labels) without any additional changes in the program. Three kinds of reflected images are possible:

Exchanging the x_min with the x_max parameter reflects the image across the y axis.

```
LIMIT x_max,x_min,y_min,y_max
```

13

Exchanging the y_min with the y_max parameter reflects the image across the x axis.

```
LIMIT x_min,x_max,y_max,y_min
```

Exchanging the x_min with the x_max parameter, and the y_min with the y_max parameter reflects the image across the origin.

```
LIMIT x_max,x_min,y_min,y_max
```

The SCALE, SHOW, and LOCATE statements can also be used to reflect plots by exchanging parameters similarly.

Note that these procedures do not reflect labels or BPLOT data. Labels are reflected by the CSIZE statement.

Using BPLOT and BREAD

The BPLOT, or byte plot, statement performs a type of plotting operation in which the system addresses individual pixels on the graphics raster, turning them on or off according to the parameters in the BPLOT statement. The plotting area is set up using the same procedures as for plotting data, axes, and labels. You can BPLOT anywhere within the default graphics limits. However, the pen cannot be positioned outside the current graphics limits (as specified by LIMIT or by default) prior to BPLOT unless the system is set to graphics units mode (unlike pen positioning for data and labels). The PEN statement is ignored during BPLOT operations. BPLOT plots white or black dots according to the BPLOT statement parameters.

The BREAD statement allows you to read the on/off states of pixels on the graphics raster. The raster is read dot by dot and placed into a character string. BREAD performs the opposite function as BPLOT; the two statements are often used cooperatively for creating and storing display dot graphics.

Byte Plotting: BPLOT

BPLOT address pixels from the current pen position, plotting across the row from left to right. Successive BPLOT statements plot rows of pixels from top to bottom, unless the pen is repositioned by another plotting or positioning statement. (Note that byte plots can't be reflected, as can other plotted images.)

The BPLOT statement reads the character string expression and interprets each character's code (an eight-digit binary number) as the on/off status of eight pixels. A "1" in the binary code specifies that a pixel is to be turned on, while a "0" indicates that the pixel is to be turned off. The syntax for BPLOT is as follows:

```
BPLOT string_expression , bytes_per_row
```

With the BPLOT statement, characters and bytes are synonymous. One character specifies the on/off status of eight pixels on the graphics raster. The string expression contains multiple bytes of information that translates into patterns of pixels. The bytes_per_row parameter specifies the number of characters (bytes) per row; it can be a numeric constant, variable, or expression. If the bytes_per_row parameter is positive, the BPLOT statement performs an exclusive or with the existing dots on the display screen; if it is negative, the pixels are inclusive or'ed with existing pixels on the graphics display. When the specified number of bytes per row are plotted, BPLOT repositions the pen to the left edge, one row below the previous byte-plotted row. BPLOT continues to plot pixels until the entire character string is converted to pixels.

For example, the following statement plots 16 characters (16 groups of eight pixels) per pixel row on the raster, until all of the characters in A$ have been plotted.

```
BPLOT A$,16
```

If A$ contained 64 characters, then the statement would produce a byte plot of four rows of 128 (16×8) pixels per row.

This statement would produce eight rows of 64 (8x8) pixels per row.

```
BPLOT A$,8
```

BPLOT can begin at any pixel location. The starting location for BPLOT is determined in two ways:

- If the most recent pen movement was directed by a BPLOT statement, then the next BPLOT string begins at the left edge of and one row below the last byte-plotted string.
- If the most recent pen movement was directed by any statement other than BPLOT, then the BPLOT begins at the current pen position (the closest pixel).

The BPLOT statement doesn't affect the pen position for other plotting operations. However, all of the other plotting statements which move the pen affect the location of the byte plotted information.

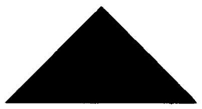
Building the BPLOT String

The procedure for building a BPLOT string is summarized below.

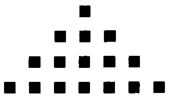
1. Draw the figure you wish to plot.
2. Redraw the figure in matrix form, using dot patterns instead of lines. Graph paper is useful for this task; let each square equal one pixel (one bit of information).
3. Divide the dot figure into columns of dots and spaces; each row must be an even multiple of eight squares wide (e.g., 16, 32, etc.). View each group of eight dots as a byte of information where each dot specifies a bit in the byte. If a dot is to be plotted, the value of the corresponding bit is one; if no dot is to be plotted, then the bit's value is zero. Each group of eight dots or spaces specifies a binary number that determines a particular character.
4. Convert each binary number to its decimal equivalent.
5. Build the character string by assigning the character of the specified decimal value (using the CHR$ function) to the appropriate character position in the string. One approach is to write a program that accepts and appends the character to the string through INPUT statements or READ and DATA statements.
6. Use this string with the BPLOT statement to plot the figure.

Use the procedure just given to create a BPLOT string for a triangle.

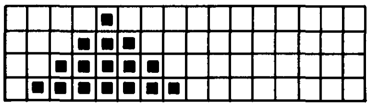
1. Draw the figure.



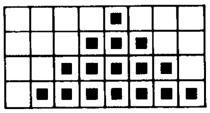
2. Represent the figure with dots.



3. Since the base of the triangle is seven dots wide, place it in a 4×8 matrix.



4. Convert each row of the matrix to a decimal value (note that all bytes on right side of drawing will be 0).



Binary Representation	Decimal Value
0 0 0 0 1 0 0 0	8
0 0 0 1 1 1 0 0	28
0 0 1 1 1 1 1 0	62
0 1 1 1 1 1 1 1	127

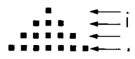
5. Build the character string using the CHR$ function and this program:

```
10 DIM T$[8] !              Dimensions the string variable.
20 FOR I=1 TO 4 !           Loop reads the decimal values into the
21 !                        appropriate character position in the
22 !                        string.
30 READ V
40 T$[2*I-1,1]=CHR$(V)
50 NEXT I !                 End loop.
51 ! Data statement contains the decimal codes for the BPLOT string.
60 DATA 8,28,62,127
70 END
```

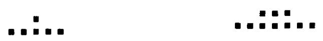
6. Use the string with the BPLOT statement to plot the triangle. BPLOT T$,2 produces a triangle because it plots two characters per line (and the second character of each line is all 0's).

   The drawings below represent the outcome of the listed BPLOT statement using four different bytes per row parameters.

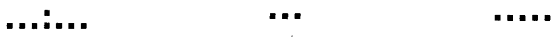
```
BPLOT T$,2
```



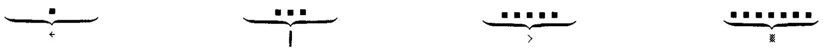
```
BPLOT T$,4
```



```
BPLOT T$,6
```



```
BPLOT T$,8
```



Byte Reading: BREAD

You can read the current states of pixels on the raster by using the BREAD statement. Note that the BREAD (byte read) statement performs the opposite of BPLOT: it reads even multiples of groups of eight dots from the graphics display and stores them as characters in a string variable. The byte reading begins at the current pen position and moves down one row of dots after reading the specified number of bytes per row. The BREAD statement continues to read bytes across and down – building the character string until the string variable has reached its allocated length. Recall that strings longer than 18 characters must be allocated memory through a DIM statement. The syntax for a BREAD statement is as follows:

```
BREAD string_variable,bytes_per_row
```

The bytes_per_row parameter can be a numeric constant, variable, or expression; negative values are interpreted as their absolute value.

BREAD does not effect the pen location for any plotting operation other than BREAD and BPLOT. An example of using BREAD is as follows:

```
BREAD String$,32
```

where String$ is the character string that is to receive the pixels to be read, and 32 is the number of bytes per row.

By dimensioning a string variable to the proper size, it is possible to BREAD the entire graphics display. In the following program, a figure is drawn on the raster of an Integral PC using the IDRAW statement in a FOR...NEXT loop. (The string size and bytes_per_row parameters are machine-dependent.) The graphics raster is then read into a string variable, and after a short pause the program BPLOTs the string variable.

```
100 ! *** Bread ***
110 PLOTTER IS 1
120 GCLEAR
130 !
140 SCALE -120,120,-50,50 !          Scales the plot to GUs.
150 MOVE 0,0
160 DEG !                            Trigonometric mode is DEG.
```

```
170 FOR Ang=0 TO 3600 STEP 100 !      Loop for plotting figure.
180   IDRAW 50*COS(Ang),50*SIN(Ang)
190 NEXT Ang !                        End of loop.
200 !
210 DIM D$[16320] !                   Dimensions string to graphics
220 !                                 display size.
230 MOVE -120,50
240 BREAD D$,64 !                     Byte reads the graphics display.
250 PAUSE
260 !
270 GCLEAR !                          Clears the graphics display.
280 MOVE -120,50
290 BPLOT D$,64 !                     Plots the byte read string.
300 END
```

Other Output Devices

Now that you have used most of the features of graphics raster devices, it is time to expand this knowledge to include other devices like pen plotters.

Graphics Defaults Restored

When you change plotting devices (with PLOTTER IS), the system sets up certain default conditions on the device and in the BASIC graphics system itself. The following operations also set up default graphics conditions:

- Executing a LIMIT statement.
- Exiting and re-entering BASIC.
- Resetting the system.

For a complete list of graphics default conditions, refer to the *HP-UX Technical BASIC Reference.*

Specifying a Plotter

The PLOTTER IS statement is used to specify the device that is to receive subsequent graphics output. Before using the PLOTTER IS statement, you need to assign an *interface select code* to an interface card, (and determine the *primary address* of the plotting device, if using an HP-IB interface and plotter). Combining these values properly results in the *device selector* for the PLOTTER IS statement.

1 For a list of select codes available on your particular system, refer to the *Implementation Specifics* appendix for your Technical BASIC system.

Interface Select Code For instance, select code 7 is used for the internal HP-IB interface on the Integral Personal Computer. This value can be assigned to the internal HP-IB interface by using the ASSIGN statement as follows:

```
ASSIGN 7 TO "hpib"
```

where "hpib" identifies the internal HP-IB.

Primary Address Next, you can determine the primary address of a plotter by looking at switch settings on the back panel. Refer to your particular plotter manual for the switch locations and settings. You will find that address 05 is the factory default switch setting for most plotters.

Device Selector With select code 7 assigned to an HP-IB interface, and plotter primary address switches set to address 05, you can use a device selector of 705 in the PLOTTER IS statement.

```
PLOTTER IS 705
```

Executing this statement from the keyboard or from a program will cause graphics to be sent to the plotter whose device selector is 705.

Considerations

There are several special considerations when using an external plotter.

- For instance, PENUP lifts a plotter's pen from the surface of the paper. The logical pen used with raster graphics could care less where it sits, but real pens with real ink make a real mess unless lifted from the paper. Thus programs that use pen plotters should make a point of lifting the pen whenever it is not moving. After plotting, cap the pen (or execute PEN 0 to put the pen away).
- The aspect ratio of an external plotter is often different than the aspect ratio of a graphics raster. Character size is also affected by this difference.
- Lines drawn by the LINE TYPE statement will differ from those defined for the display. Check the plotter's manual for descriptions of line type parameters.

Graphics Using HPGL Commands

To simplify communicating with the wide variety of HP graphics devices, a standard set of graphic commands has been adopted. The Hewlett-Packard Graphics Language (HPGL) consists of about sixty, two-letter commands that can be used to control the operation of most HP plotters. If fact, when BASIC statements are used to control an external plotter, they are converted (by the system) into a series of HPGL commands which are then sent to the plotter. Refer to the plotter's manual for details concerning which HPGL commands the plotter can recognize.

While most plotting applications can be accomplished by using BASIC statements, some plotters have capabilities that can only be accessed by using HPGL commands. When it is necessary (or desirable) to communicate directly with the plotter, the OUTPUT statement can be used to send HPGL commands. For example,

```
ASSIGN 7 TO "hpib"
OUTPUT 705;"DF;";
```

This statement sends the HPGL command to restore the default conditions of the plotter. Many of the HPGL commands have one or more parameters. For instance:

```
ASSIGN 7 TO "hpib"
OUTPUT 705;"LT 6;";
```

This statement sets the line type to pattern number 6. The line type is plotter dependent and not likely to be the same pattern displayed on the CRT by the LINE TYPE statement in Technical BASIC.

In general, an HPGL command is terminated by either a semicolon or a line-feed. The parameters in commands are usually separated by commas. In the previous example statement, a semicolon is included in the string sent to the external plotter to indicate the end of a command. The OUTPUT statement's trailing semicolon suppresses the current end-of-line sequence from being sent to the plotter.

Some HPGL commands request the plotter to send back information to the computer. The ENTER statement is used to receive the information. The following example interrogates the plotter for the coordinates of the lower-left (P1) and upper-right (P2) graphics limit.

```
100 ! This program determines the coordinates of
110 ! the lower-left corner (P1) and upper-right corner (P2)
120 ! of the "plotting area" (in "absolute device units").
130 !
140 ! Ask plotter to "Output Points P1 and P2".
150 OUTPUT 705; "OP;";
160 !
170 ! Now input the points.
180 ENTER 705 ; P1x,P1y,P2x,P2y
190 !
200 ! Now show the coordinates.
210 CLEAR @ DISP
220 DISP "Lower-left corner, P1: (";P1x;",";P1y;")"
220 DISP "Upper-right corner, P2: (";P2x;",";P2y;")"
230 END
```

The results of executing this program on an HP 7475 plotter are as follows:

```
Lower-left corner, P1: ( 250  , 596 )
Upper-right corner, P2: ( 10250  , 7796 )
```

The OUTPUT statement sends the "Output Points P1 and P2" command to the plotter, and the ENTER statement accepts the X and Y coordinates for P1 (lower-left corner) and P2 (upper-right corner) sent by the plotter. The values returned are in "absolute device units," not in GU's or UU's. One absolute device unit is equal to 0.025 millimetre.

You might wonder how the mapping of GU's, UU's, and absolute device units is accomplished. Consider the following statement.

```
PLOTTER IS 705
```

This statement actually sends several HPGL commands to the plotter and accepts the current setting of P1 and P2 for use by the computer in converting the values used by Technical BASIC statements into the values needed for HPGL commands.

If you wish to change the locations of P1 and P2, it will be necessary to re-execute the PLOTTER IS statement (after changing P1 and P2). This allows Technical BASIC to become aware of the new graphics limits, and set up the correspondence between UU's (or GU's) and absolute device units.

Graphics with Printers

There are printers capable of reproducing the image on the graphics raster. For a list of printers that support these operations, see the *Implementation Specifics* appendix for your particular system. Usually there is a one-for-one correspondence between a pixel on the screen and a dot on the printed paper.

The image on the graphics raster can be sent to the external printer by entering the statement,

```
DUMP GRAPHICS
```

The contents of the alpha display can be sent to the PRINTER IS device by the statement,

```
DUMP ALPHA
```

Interactive Graphics

Technical BASIC has the ability to accept inputs from a graphics device. The following capabilities are supported.

- You can digitize individual points on the PLOTTER IS device. Each point is selected by first positioning the device's locator (typically a pen on plotting devices or stylus on input devices) and pressing the "digitize" button (typically the "ENTER" button on plotters or the stylus on input devices).
- The ability to "continuously" monitor the locator's coordinates.
- The ability to monitor the Digitize button to determine whether or not it is currently being pressed.

Compatible Devices

The PLOTTER IS statement is used for designating the input or output device. The device selector used in conjunction with the PLOTTER IS statement determines which device you are selecting. A list of the input and output devices supported on your system can be found in the *Implementation Specifics* appendix for your system.

Digitizing Graphics Images

Digitizing is essentially the inverse of plotting. During plotting operations, the computer sends x,y coordinate values and pen status instructions (pen up or down) to the plotter, directing the pen to the specified location on the plotting area. To digitize a point, you can use the plotter's front-panel controls to move the pen to the desired point and then press a key to tell the system to determine the coordinates of (i.e., digitize) the point.

The digitizing process enables you to convert graphics information into digital information. For example, you could trace the outline of a drawing or photograph with the plotter's pen or digitizing sight, digitizing the coordinates along the way. The pen coordinates and status are read into computer memory in a format identical to the PLOT statement – as numeric x,y coordinates. This information can then be used with plotting statements to create a reproduction of the original graphics image.

This section begins with a discussion of digitizing graphics limits and plotting boundaries. It is followed by a discussion of digitizing pen position and up/down status.

Digitizing Graphics Limits and Plotting Bounds

When executed without parameters, the LIMIT, LOCATE, and CLIP statements allow you to manually move the graphics limits or plotting boundaries on the plotting device. Executing these statements without parameters suspends program execution.

- With the LIMIT statement, the system waits to receive a message from the plotter containing the location of the lower-left and upper-right **graphics limits**.
- With LOCATE and CLIP, the system waits to receive the location of the lower-left and upper-right CLIP or LOCATE **plotting boundaries**.

The procedure for digitizing the graphics limits or the plotting boundaries is as follows:

1. Execute LIMIT, LOCATE, or CLIP, which suspends program execution.
2. Move the pen to the desired lower-left limit or boundary and press the **ENTER** key on the plotting device. The pen's location is sent to the system, where it is interpreted as the lower-left limit or plotting boundary. The system beeps when the **ENTER** key is pressed to signify that the digitized information has been received.
3. Move the pen to the desired upper-right limit or boundary and press the **ENTER** key again. The pen's location is sent to the system and interpreted as the upper-right graphics limit or plotting boundary. The computer beeps once again after pressing the **ENTER** key to signify that the digitized information has been received.
4. The digitized graphics limits or plotting boundaries are now active, and program execution continues.

Normally you would want to enter the lower-left limit or boundary first and the upper-right limit or boundary second. However, you can also digitize the graphics limits or plotting boundaries in different orientations to get a reflected image of your plot. For example, if you enter the upper-right limit first and the lower-left limit second, your plot will appear as if it was reflected through the origin. The procedure is analogous to exchanging parameters in the LIMIT or LOCATE statement. The sequence (first or second) and location (lower-left, upper-right, upper-left, or lower-right) of the digitized graphics limit or plotting boundary corner determines the type of reflection. The three types of reflections are summarized in the table below. (Note that digitized CLIP boundaries cannot be used to reflect plots.)

LIMIT and LOCATE Reflected Plots

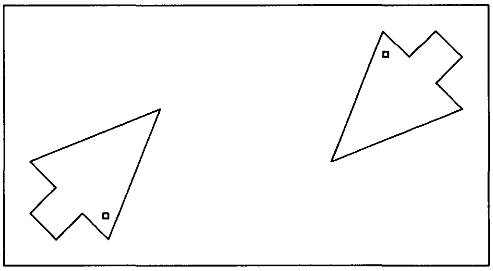
	Unreflected plot	**Reflection across origin**	**Reflection across x-axis**	**Reflection across y-axis**
Location of first digitized limit or boundary	lower-left corner	upper-right corner	upper-left corner	lower-right corner
Location of second digitized limit or boundary	upper-right corner	lower-left corner	lower-right corner	upper-left corner

The following program digitizes the graphics limits, frames the plotting area, and then draws an arrow; the arrow points from the first digitized corner to the second digitized corner of the graphics limits. Experiment with your plotter by digitizing different graphics limits, and note how the shape and orientation of the figure changes.

```
program listing digitize goes here
```

```
100 ASSIGN 7 to "hpib"
110 PLOTTER IS 705 !       Specifies the plotting device.
120 CLEAR
130 DISP "DIGITIZE THE GRAPHICS LIMITS."
140 LIMIT !                Computer waits while you digitize the
150 !                      graphics limits from the plotter.
160 CLEAR
170 DISP "PLOTTING !"
180 FRAME !                Frames the digitized plotting area.
190 READ X,Y
200 MOVE X,Y
210 FOR I=1 TO 7 !         Loop plots figure.
220   READ X,Y
230   DRAW X,Y
240 NEXT I !               End of loop.
250 !
260 READ X,Y
270 MOVE X,Y
280 FOR I=1 TO 4
290   READ X,Y
300   DRAW X,Y
310 NEXT I
320 PENUP
330 !
340 RESTORE !   Restores DATA pointer.
350 GOTO 120 !  Repeats digitizing loop.
360 !
370 DATA 20,10,10,20,20,30,10,40,60,60,40,10,30,20,20,10
380 DATA 38,18,40,18,40,20,38,20,38,18
390 !
400 END
```

The example output below shows the image of two arrows, each of which is the reflection (through the origin) of the other. It was produced by first digitizing the lower-left and upper right corners, followed by digitizing the upper-right and lower-left corners.



Digitizing Pen Locations

You have already seen an example of digitizing by executing the LIMIT statement without parameters and entering the graphics limits from the plotter. This feature is just one application of your computer's digitizing capability. You can also digitize any point on the plotting area and store it for later use. In order to better understand these operations, however, you may need a little background.

Physical and Logical Pens The ink pen on a pen plotter and the thermal print head on a thermal printer are both considered "physical pens" in the sense that they draw the lines, points, and curves which constitute plotter graphics.

The BASIC system has a pen of its own, known as the "logical pen". The X and Y coordinates and up/down status of the logical pen reside in memory and are determined by the most recently executed statement affecting logical pen location and status. For example, executing PLOT 10,20,1 lowers the logical pen at the coordinates 10,20 (according to the current scaling units).

On some devices, the physical pen location can be changed at the device. For example, you can move a plotter pen by using the front-panel pen-movement controls. The physical pen location can also be altered by executing a plotting statement (for example PLOT, MOVE, LABEL, or AXES). The physical pen is always located within the physical limits of the plotting device, but not necessarily within the current plotting area.

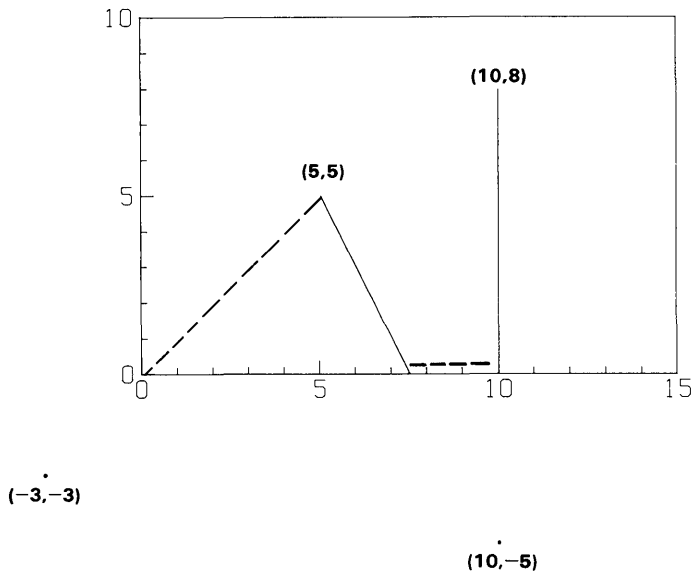
The location and status of the logical pen are unaffected by the pen movement controls on the plotting device. The logical pen can be located anywhere inside or outside the physical limits of the device.

Although the physical and logical pens coincide with each other during most plotting operations, they are each recognized individually by the system. Listed below, are some instances where the logical and physical pens have different locations.

- Whenever the graphics default conditions are activated, the logical pen moves to the lower-left corner of the plotting area. However, the physical pen location is unaffected.
- When a plotting statement directs the pen to a point outside the current plotting area, the physical pen stops short of the intended point, at the current graphics limit or plotting boundary and is lifted (refer to the diagram below). In contrast, the logical pen location and status always coincide with the destination point and status specified by the plotting statement, regardless of whether or not the point lies within the current plotting area and whether or not it was actually plotted.
- Whenever the plotting device is changed, the physical and logical pens may have different locations depending on the initial physical pen position and the last executed plotting statement.
- Whenever the physical pen is moved using the pen movement controls at the external plotting device, the physical and logical pen locations differ.

13

The following diagram shows the location of the physical and logical pens during the sequence of plotting statements listed in the table below. The framed plotting area is scaled from 0 to 15 in the X direction and from 0 to 10 in the Y direction. The solid black line indicates a line drawn during physical pen movement. The dashed black line indicates physical pen movement without line drawing. Note that the computer plots successive points according to the logical pen location.



Execute:	Resulting physical pen location and status	Resulting logical pen location and status
`PLOT 10,8,1`	(10,8) down	(10,8) down
`DRAW 10,-5`	(10,0) up	(10, −5) down
`PLOT 5,5`	(5,5) down	(5,5) down
`MOVE -3,-3`	(0,0) up	(−3, −3) up

Digitizing the Physical Pen Location Digitizing the plotter's physical pen is an operation which involves both the plotter and the computer. Here is an example statement that digitizes the PLOTTER IS device's *physical pen* location:

```
DIGITIZE Xpos,Ypos,PenStatus
```

The system first asks the plotter for the pen's coordinates and up/down status. When the "ENTER" button is pressed, the plotter sends the information (as numbers) to the computer. The BASIC system then stores the information in the three numeric variables specified. The first two variables identify the coordinate location of the physical pen; the third variable identifies the pen status. The variables are assigned values according to the current scaling units. The optional third variable parameter is assigned the pen status information. If the pen is up, 0 is assigned to the variable. If the pen is down, 1 is assigned to the variable. All three variables must be numeric variables.

There is also another statement which can be used to digitize the physical pen's location: CURSOR. The syntax is the same as for the DIGITIZE statement, but the statements use different methods for entering the digitized information into computer memory:

- **The DIGITIZE statement suspends program execution** while you position the plotter's pen to the desired location and waits until the **ENTER** button is pressed on the plotter. The physical pen's coordinates and up/down status are read into computer memory only when the **ENTER** button is pressed. When the computer receives the digitized information, program execution continues. Here is an example that uses the DIGITIZE statement:

  ```
  DIGITIZE Xvar,Yvar,PenStatus
  ```

  where Xvar and Yvar are the coordinates of the point that was plotted and PenStatus is the pen status which tells whether the pen is in the up or down position.

- **The CURSOR statement does not suspend program execution.** The physical pen's coordinates and up/down status are read into the specified variables immediately – without pressing the plotter's **ENTER** button. Here is an example that uses the CURSOR statement:

  ```
  CURSOR Xvar,Yvar,PenStatus
  ```

  where Xvar and Yvar are the variables that receive the coordinates of the point that was plotted and PenStatus is the pen status which tells whether the pen is in the up or down position.

Keep in mind that the pen must be positioned at the desired location for digitizing prior to executing the CURSOR statement. The DIGITIZE statement allows you to position the pen and enter the digitized information after DIGITIZE is executed.

Digitizing Images on the Integral PC Raster You can digitize points on an Integral PC's graphics raster by using the procedure explained in this section.

1. Execute this statement:

   ```
   DIGITIZE X,Y,P
   ```

   Your display will enter the graphics mode.
2. Next, move the graphics pointer into the lower-left corner of your active (BASIC graphics) window by pressing the **CTRL** key and pressing one of the cursor-arrow keys (on the lower-right portion of the keyboard).
3. Press the **User** key and then the **Select** key to obtain the set of softkeys used for digitizing. You will know that you have the right set of softkeys when you see the word *basic_g* appears in the softkey labels area of the display.
4. Press the softkey labeled **graph** and a new set of softkey labels will appear. Look for the softkey labeled **FastPen**. Press this softkey and an asterisk should appear in its softkey label.

5. You are now ready to move the digitize arrow to any point on the display that you wish to digitize. To do this, press the arrow keys until you have moved the digitize arrow over the point you wish to digitize. Next, press the **Enter** key to store this point as the variables X, Y, and P.
6. Return to the Technical BASIC window after digitizing the point on your display. To return to this window hold down the **Shift** key and press the **Select** key until the Technical BASIC window appears with your DIGITIZE statement in it.
7. Now that you are in the Technical BASIC window enter the following statement:

   ```
   DISP X,Y,P
   ```

   This displays the coordinates for the digitized point. Your result should look similar to this:

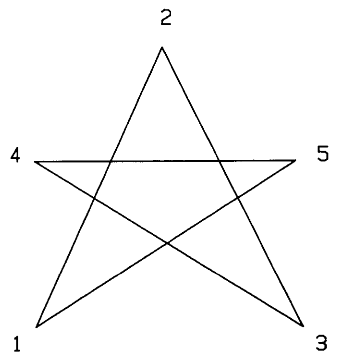
   ```
   60.2362204724400      37.7952755905512      0
   ```

   The final number show is either 0 or 1, depending on whether you pressed the softkey labeled **Pen Down** or **Pen Up**, respectively.

The following program allows you to digitize 5 points on the display and draw lines that connect them. Enter and run the program, and use the arrow keys to move the digitize arrow and the **Enter** key to digitize the points. Note that once you have entered 5 points on the display, the program will draw lines connecting these points. After the drawing is complete, you will be returned to the Technical BASIC window.

```
100 ! *** Digitize ***
110 OPTION BASE0
120 GCLEAR
130 !
140 FOR I=0 TO 4 !   Digitize 5 points.
150   DIGITIZE X(I),Y(I),P(I)
160 NEXT I !
170 !
180 MOVE X(0),Y(0) ! Move back to starting point.
190 FOR I=1 TO 5 !   Draw lines to connect the 5 points.
200   DRAW X(I MOD 5),Y(I MOD 5)
210 NEXT I
```

A drawing of a star is given below as an example of using this program. Run the program, and then digitize points 1 through 5. The program then draws the star.



Using DIGITIZE and CURSOR To use the CURSOR statement, lift the plotter's pen and move it to any point using the pen movement controls and then execute this statement:

```
CURSOR X,Y,P
```

Look at the coordinates stored and the pen status, by executing the following statement:

```
DISP X,Y,P
```

The physical pen's coordinates and status are shown on the display. They look similar to this:

```
53.4722222222222      81.0555555555556      0
```

To use the DIGITIZE statement, execute:

```
DIGITIZE Xvar,Yvar,PenStatus
```

Lift the pen and relocate it on the plotter. Lower the pen and press the plotter's **ENTER** button. Next, execute the following statement:

```
DISP Xvar,Yvar,PenStatus
```

The physical pen's new X,Y coordinates and status are shown on the display:

```
64.6527777777778      31.7916666666667      1
```

Be sure to lift the pen after you have finished digitizing.

Tracing Graphics Images Digitizing operations are commonly used for tracing drawings or other graphics images, which can then be reproduced using the PLOT statement. Note the similarity between the DIGITIZE statement and the PLOT statement.

```
DIGITIZE x_variable,y_variable,pen_status_var

PLOT x_coordinate,y_coordinate,pen_control
```

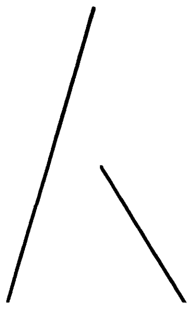
If the pen is in the appropriate up/down position while digitizing, the PLOT statement can use the pen status variable for the pen control parameter.

The digitized pen status variable takes on the value 0 or 1 depending on whether the pen is up or down, respectively. When the PLOT statement interprets a digitized pen status value as input for pen control, there are two possible results.

1. Pen status = 0 (for example, PLOT 4,-6,0): The pen is directed to the specified x,y coordinate and lifted *after movement*. The pen maintains its initial up or down status until relocated at the specified x,y coordinate. If the pen is initially down, a line is drawn to the specified point and lifted. If the pen is initially up, the pen is moved to the specified point and remains up.
2. Pen status = 1 (for example, PLOT 10,14,1): The pen is directed to the specified x,y coordinate and lowered *after movement*. The pen maintains it's initial up or down status until the pen is relocated.

The outcome of both PLOT statements is determined by the pen's up or down status preceding execution of PLOT. Therefore, when you digitize a point, have the pen set to the correct up/down status for the next digitized point.

An Exercise in Tracing This section takes you through the steps necessary for tracing the image shown below. Keep in mind how pen status affects pen control when the digitized data is plotted using the PLOT statement.



The drawing consists of two line segments, requiring you to digitize four points (the endpoints of the two lines).

The following group of statements enables you to digitize four points on the current PLOTTER IS device:

```
FOR I= 1 TO 4 @ DIGITIZE X(I),Y(I),P(I) @ NEXT I
```

The coordinates variables X(I),Y(I) are assigned the physical pen locations, according to the current scale. Pen status information is assigned to the variable P(I).

To digitize the example drawing, follow this sequence of steps:

1. Place a copy of the drawing onto the plotter. Your plotter should already be turned on.
2. Set the graphics limits manually, ("P1" and "P2" on the plotter), so that the drawing is located within the plotting area. Execute the PLOTTER IS statement to read the manually set graphics limits.
3. Execute the above multi-line statement that digitizes four points.

4. Digitize the four points in the sequence shown in the previous drawing, using the indicated up/down pen status. It is easiest to position the pen at the desired point while the pen is up. When the pen is properly positioned and in the correct up/down status, press the plotter's **ENTER** button and move on to the next point.

The digitized coordinates and pen status information are stored in the numeric arrays X(I), Y(I), and P(I). To reproduce the digitized image, execute the following multi-statement line. Be sure your plotter is equipped with paper and a pen.

```
FOR I = 1 TO 4 @ PLOT X(I),Y(I),P(I) @ NEXT I
```

If the physical pen coordinates and status were entered as shown, the plotter duplicates the original drawing.

Keep in mind that the DIGITIZE and CURSOR statements digitize points according to the current scaling units. To recreate a digitized image accurately, the scaling units and plotting area dimensions in effect while plotting must match those in effect while digitizing.

Digitizing the Logical Pen Location The WHERE statement assigns the current logical pen coordinates and status to the specified variables. The parameters are the same as the parameters in the CURSOR and DIGITIZE statements.

```
WHERE x_variable,y_variable,pen_status_var
```

The location and up/down status of the logical pen is determined by the most recently executed statement which changes pen status or location. All of the plotting statements which direct pen movement also affect the logical pen location. In addition, statements and conditions which activate the default graphics conditions also lift the logical pen and move it to the origin (0,0). However, the physical pen's location and status are unaffected by activating the default graphics conditions.

The logical and physical pens often have the same location and status; any plotting statement which directs pen movement inside the current plotting area moves the physical pen as well as the logical pen.

The following program demonstrates the difference between the physical and logical pen positions as read by the CURSOR and WHERE statements. When program execution is suspended, move the pen (using the plotter's front panel controls) to a new location, lower the pen, execute CONT. The computer displays the resulting physical (CURSOR) and logical (WHERE) pen coordinate locations and pen status. In the example output below, the physical pen was moved to the coordinate location x = 76.6, y = 68.0, and lowered.

```
100 ASSIGN 7 to "hpib"
110 PLOTTER IS 705 !  Specifies the plotting device.
120 MOVE 50,50 !      Moves the pen to the point (50,50)
130 PAUSE !           Pauses the program while you move
140 !                 the plotter pen to a new position.
150 WHERE WX,WY,WP !  Assigns logical pen position and
160 !                 status to the variables WX,WY,WP.
170 CURSOR CX,CY,CP ! Assigns physical pen position and
180 !                 status to the variables CX,CY,CP.
190 CLEAR
200 DISP USING "8A,2X,2(3D.D),3X,D";"WHERE",WX;WY;WP
210 DISP USING "8A,2X,2(3D.D),3X,D";"CURSOR",CX;CY;CP
220 END
```

The results from the programs execution are:

```
WHERE          50.0   50.0   0
CURSOR         76.6   68.0   1
```

Index

HEWLETT
PACKARD

Reorder Number
97068-90000
Printed in U.S.A. 5/85

97068-90600
Mfg. No. Only