Panasonic

Friends Amis™ HHC Software

MICROSOFT BASIC

VOLUME I: TUTORIAL

A PROGRAMMING LANGUAGE FOR THE HHC™

RL-S6001S7

MICROSOFT BASIC

a programming language
for the HHC[T.M.]

VOLUME I: TUTORIAL

by

FRIENDS AMIS, INC.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

CHAPTER 2: GETTING STARTED

CHAPTER 3: VARIABLES

CHAPTER 4: WRITING A STORED PROGRAM

CHAPTER 5: MORE ABOUT BASIC

CHAPTER 6: MORE ABOUT EDITING PROGRAMS

CHAPTER 7: MANAGING PROGRAM FILES

CHAPTER 8: REMARKS IN BASIC PROGRAMS

CHAPTER 9: FLOW OF CONTROL

CHAPTER 10: ARRAYS

CHAPTER 11: DEVELOPING PROGRAMS; SOME EXAMPLES

CHAPTER 12: THE FOR/NEXT STATEMENT

CHAPTER 13: MORE ABOUT I/O

CHAPTER 14: FUNCTIONS

CHAPTER 15: STRINGS

CHAPTER 16: SUBROUTINES

CHAPTER 17: PEEKS AND POKES

CHAPTER 18: USING THE FUNCTION KEYS

CHAPTER 19: ADVANCED I/O TECHNIQUES

CHAPTER 1: INTRODUCTION

This book is your introduction to computer programming with Microsoft Basic for your HHC™.

Whether you are a new user learning to program for the first time or an experienced programmer just beginning to use the HHC, you will find that programming in Microsoft Basic is a pleasant, challenging and rewarding experience.

WHAT IS BASIC?

The unique power of a computer is based on the fact that it is **programmable**. This means that it not only can store data and do calculations, as a pocket calculator can do, it also can run under the control of a set of instructions which defines the steps in a calculation that is far more complex than you could perform by hand. Such a set of instructions is a **computer program**.

There are two ways you can run programs on the HHC. First, you can buy pre-packaged programs in **HHC capsules**. To run such a program, all you need do is plug the HHC capsule into your HHC.

Often, however, the program you want to run is not available in an HHC capsule. In that case you can write the program yourself, or obtain it from another person who has written it.{1}

Basic is a **programming language** -- a convenient form of notation for writing computer programs. Your HHC can run Basic programs with the help of another program, called a **Basic interpreter**, which is contained in the HHC capsule that accompanies this book.

WHAT CAN BASIC DO?

Basic is a programming language that is often used on personal computers and time sharing systems. Its major benefits are:

- It is easy to learn and use. This makes it a good first language if you are just learning to program.
- It is convenient for many tasks that involve mathematics or manipulating strings of characters. It is often used to

{1} - In many cases it is possible to store a completed Basic program in a HHC capsule. This protects the program from being changed or erased accidentally, and makes it cheap and easy to distribute to large numbers of users. If you are interested in having a program stored in ROM capsules, contact Friends Amis Incorporated, 505 Beach Street, San Francisco, CA 94133; telephone (415)928-2800.

write accounting programs and similar programs for business or personal use.

- It is available on many different computers. Once you are familiar with Microsoft Basic, you will find that you can learn to use many other kinds of computers, large and small, with little effort.
- Many computer programmers know it. More users of small and medium-size computers know Basic than any other single language.
- Many computer programs have been written in it. When you need a program to solve a problem, you will often find that someone has already written it, and you can adapt it to the HHC with relatively little work. If you are using the HHC as a portable device for communicating with a larger computer, you will often find that programs already running on the larger computer can easily be adapted to the HHC.

While most types of applications can be programmed in Basic, another language will sometimes make your programming work easier because of:

- Features that make large programs easy to write and change.
- Features that make it easy for a program to detect certain types of user errors, such as entering a word when the program expects a number.
- Features that allow you to use the HHC's file system. Basic programs cannot read and write files, making Basic an awkward choice for writing a program that must deal with large amounts of data.
- Greater efficiency. Some programs written in Basic will run less efficiently than equivalent programs written in another language.

If you are interested in writing programs for which Microsoft Basic is not the best choice, consider learning to program in one of the following languages:

- SNAP Basic, a more comprehensive Basic system that is distributed in ROM capsules for the HHC.
- SNAP, a very powerful language designed for professional programmers who want to develop packaged programs. SNAP was used to write most of the HHC's internal software. It is the language of choice for developing new HHC capsule programs.

Your HHC's distributor can provide you with information about these and other HHC programming languages as they become available.

HOW TO USE THIS BOOK

This book is organized as a ***tutorial*** guide that teaches you how to program in Microsoft Basic.

This book's companion volume, the ***Microsoft Basic Reference Guide***, contains detailed information that you can refer to as you write programs. We will refer to it as the ***Reference Guide***.

If you are new to Basic and the HHC, read every part of this book carefully. Try all the examples. Start writing your own programs as soon as you can; you'll find that there is nothing like practice for increasing your programming skill.

If you are familiar with Basic, but not the HHC, study the sections of the book that are marked with this symbol: {**H**}. These sections describe the mechanics of using the HHC. Also study the sections of the book that are marked with this symbol: {**B**}. These sections describe features of Basic that differ greatly from other versions of Basic, and/or will cause you trouble if you do not understand exactly how they work. Look at the ***Reference Guide*** to see what functions, operators and keywords Basic recognizes.

If you are familiar with the HHC, but not with Basic, you can skim over the sections of the book that are marked with {**H**}.

If you are familiar with both Basic and the HHC, and only need to know how Microsoft Basic on the HHC differs from other Basics, you can skim over most of the book; pay attention to the sections that are marked with {**B**}.

This book presents examples of the characters exactly as they appear on the HHC display. We have used the ▶ symbol to indicate when a one-line display is continued on the following line in the example.

CHAPTER 2: GETTING STARTED

PREPARING THE HHC TO RUN BASIC

If you have not yet used your HHC at all, there are some very simple set-up procedures you must carry out to make it ready for use.

Turn the HHC so that its keyboard is facing away from you.

The rectangular panel along the bottom of the back can be lifted off easily by prying up the tab next to the legend 'OPEN'. Lift off the panel and look at the three **ROM sockets** behind it. Each of these sockets can hold an HHC capsule.

Plug the HHC capsule containing Basic into any of the three sockets. The flat side of each capsule should face inward; the arrow on the outward-facing side should point down, to match the arrow on the bottom of the socket. (This is the only way a HHC capsule fits in the socket; you can't insert one incorrectly!)

After inserting the Basic HHC capsule, replace the panel that covers the sockets.

The HHC's **ALL OFF switch** is located in a recessed slot above the left end of the panel. Use a slender object such as a pencil to move this switch to the ON position.

The HHC's ALL OFF switch should stay ON at all times -- even when you put the HHC away at the end of the day. In this way, data that you tell the HHC to preserve can be preserved. Because of the HHC's unique power-conserving design, leaving the main power switch ON does not cause an undue drain on the HHC's battery.

With the keyboard still facing away from you, find the circular hole on the left end of the HHC. This hole is a socket for plugging in the HHC's **AC Adaptor**. Plug the AC adaptor's jack into this socket, and plug its power cord into an electrical outlet.

The AC Adaptor can be set to operate on either 110 volt or 220 volt power. Be sure your AC Adaptor is set for the voltage that your electrical system provides. If you try to use the AC Adaptor at the wrong voltage setting you may damage it seriously.

The HHC has built-in rechargeable batteries that let you operate it for many hours without the AC Adaptor. To keep the batteries fully charged, however, we suggest that you use the AC adaptor whenever it is convenient to do so. The AC Adaptor charges the HHC's battery whenever it is plugged in, unless the HHC's ALL OFF switch is set to OFF.

If you try to use your HHC when the batteries' charge is uncomfortably low, the LCD displays the message 'BATT LOW'. This is a signal that you had better plug in the AC Adaptor to run the HHC and recharge the batteries.

GETTING ACQUAINTED

Hold the HHC with its keyboard right side up and facing toward you. Find each of its features as we describe them.

- To the right of the main keyboard are the **ON** and **OFF** keys, which you use to turn the power on and off as you use the HHC. Unlike the ALL OFF switch on the back of the HHC, the OFF key preserves the contents of the HHC's internal storage. Below the ON and OFF keys is the **CLEAR** key, which you use with many HHC programs (***but not with Basic***) to reset the HHC when you have finished a task.{1}
- Above the keyboard is a long rectangular frame containing a **liquid crystal display** (**LCD** for short). The LCD is the HHC's primary means of displaying information. It is capable of displaying one line, 26 characters long.
- Just below the LCD is a line of words that say '. . . SHIFT . . . LOCK . . . 2nd SFT . . . DELETE . . . INSERT . . . ALARM . .. ON LINE.' The HHC can display a row of triangular dots called **blips** on the LCD, above these words. The HHC uses blips to give you information about the status of the HHC. For example, the SHIFT blip goes on when you press the SHIFT key (which is similar to the case shift key on a typewriter).
- The keyboard occupies most of the HHC's face.
- The keys in the central part of the keyboard type characters like 'a', '4', or '?' when you press them. The symbols inscribed next to these keys show that the keyboard's layout is similar to that of a typewriter. (Many of the keys are labelled with two symbols, and some of the symbols in the second set may not be familiar to you; you'll learn how to enter these symbols later in this chapter.)
- Other keys are labelled with arrows, with words like 'HELP' or abbreviations like 'STP/SPD', or with codes like 'f1'. These keys are used to control the operation of the HHC rather than to type characters.

{B} {1} - **Note to old HHC hands:** you return from Basic's programming mode to Basic's menu by entering the command BYE. Then you return to the primary menu by pressing CLEAR. Chapter 4, "Writing a Stored Program," describes this process fully.

The following control keys are especially important:

- **ENTER**, located near the lower right corner of the keyboard. It is similar to a typewriter's RETURN key.
- **SHIFT**, just to the right of ENTER. It is similar to a typewriter's SHIFT key. Notice that there is only one SHIFT key on the HHC.
- **2nd SFT** (short for "**second shift**"), to the left of ENTER. This key is similar to SHIFT, but gives you a ***second*** set of upper case characters. With the aid of this key you can type a total of 96 different characters.

Look at the HHC's left side. The slot you see is a socket for plugging in a **peripheral device** such as a printer or a telephone coupler (modem).

STARTING BASIC

Attach the AC Adaptor to the power socket on the HHC's side, and plug the AC Adaptor into an electrical outlet. Remember, the AC Adaptor not only provides power to the HHC, but also keeps its battery charged so that you can use it without plugging it in when you want to.

Turn the HHC on by pressing the ON key. Look at the LCD. It should display some information in black letters on a clear background.

If the LCD displays the word 'RESTART', press the CLEAR key once or twice to make the word go away.

Now the LCD should display the following messages, one line at a time, over and over:

```
1=CALCULATOR
2=CLOCK/CONTROLLER
3=FILE SYSTEM
4=RUN SNAP PROGRAMS
5=Microsoft BASIC
```

This display is a **menu**. It is the HHC's way of asking you what you want it to do. You pick a selection from the menu by pressing the corresponding number key. For example, to make the HHC run the calculator, you would press the '1' key.

The HHC displays many different menus at different times. The one you are looking at now is called the **primary menu**, since it leads you to all the other functions that the HHC can perform.

You want to run Microsoft Basic, so you should respond to

the primary menu by pressing the '5' key. (If you should happen to press the wrong key, press CLEAR once or twice to get back to the primary menu.)

The HHC displays the message

```
Microsoft BASIC
```

to confirm your choice. Then it begins running the Basic interpreter.[2]

Basic displays another menu that looks like this:

```
1=New file
No files
```

There's only one numbered selection on this menu: #1, "New file". This selection allows you to create a new Basic program. The message says "New file" because when Basic saves a program, it sets aside part of its storage for the program, and this part of its storage is called a **file**.

Pick the "New file" selection now by pressing the '1' key.

The Basic interpreter displays the message

```
New file
```

to confirm your choice. (It works just like the primary menu. Most of the HHC's menus work the same way!) Then the interpreter displays the message

```
Program name:
```

followed by a flashing black square. This **prompt** asks you to give a name to the file that your program will be stored in. The flashing black square is a **cursor** which invites you to enter characters on the keyboard.

The name you give the file may be of any reasonable length, and may contain any character you can type on the keyboard, including 'space'. For convenience, we suggest that you use names that are fairly short, but describe your programs well, and contain no strange characters like '?' or '@'.

Enter the program name of your choice through the keyboard, and press ENTER. For example, if you decide to

---

{B} [2] - **Note to old HHC hands:** now is the time to turn on any peripheral devices and define any function keys you intend to use in Basic. You will not be able to do so later, because the I/O key and the HELP key do not have their usual functions while you are editing and running Basic programs. They duplicate the function of the ENTER key.

name your program 'program 1', type

```
Program 1
```

and press ENTER. Don't use the shift key yet; let your program name be all in lower case.

As soon as you enter the first character of your program name, the 'Program name:' prompt disappears, and the name you are entering appears on the LCD. The cursor is always over the next position where a typed character will be displayed, like the type element of a typewriter.

When you press ENTER, your program name disappears, and is replaced by the symbol

```
>
```

followed by the cursor. Now Basic is waiting for you to start programming.

HOW BASIC HANDLES ERRORS

Before you begin learning to write correct Basic programs, let's see what happens when you write an incorrect one. You're going to write plenty of those while you're learning, so you might as well get used to it now!

Enter something that is obviously nonsense, such as 'qwerty', and press ENTER. Basic beeps at you and displays the **error message**

```
SN error
```

This means "**Syn**tax error," an error in the way you wrote something. Basic is telling you what you did wrong.

That wasn't so bad, was it? There is nothing you can type into your HHC that will harm it physically, and there are few errors you can make in a Basic program that will have any effect more serious than this one did. (The few "dangerous" errors can happen only in rarely used parts of Basic that we will point out clearly when we get to them.)

So don't worry about damaging your HHC by typing the wrong thing. You can't.

THE PRINT STATEMENT

A computer program consists of steps which accomplish some desired result when the computer **executes** them in the proper sequence. In Basic, each step in a program is called a **statement**.

We're going to start programming in Basic by executing some simple statements on the HHC. We will begin with the PRINT statement, which displays information on the LCD.

Type in the following statement:

```
Print 15
```

Press the ENTER key and watch what happens.

When you press ENTER, Basic clears the LCD (that always happens when you start a program) and displays the number '15' (that is the function of the statement you entered).

The statement 'print 15' has two parts. The first part is the word 'print'. 'print' is a **reserved word** -- a word that has a special meaning to Basic.

The second part of the statement is the word '15'. We call '15' a **constant**, because it represents a value that does not change. This particular constant is a **numeric** constant, because its value is a number, 15. We will learn about other kinds of constants later.

Try entering the following statement:

```
Print 15.3
```

Basic obediently displays '15.3'. Basic is not limited to calculations involving integers; it can deal with fractional numbers, too. Computer people call numbers of this sort **real numbers** or **floating point numbers**.

Try the following statement:

```
Print 15.300000
```

Basic displays '15.3', not '15.300000'. This is because Basic processes the statement 'print 15.300000' in two steps:

1. Basic processes the the number '15.300000'. Basic processes the number by converting it to an internal format that it uses to do arithmetic, in which '15.300000', '15.3', and '0015.3' are all represented the same way.
2. Basic processes 'PRINT'. The function of PRINT is to convert the following number from internal format to external format, and display it on the LCD. When Basic displays a number it always omits trailing zeros after the decimal point.

Try the following statement:

```
Print 1.01234567890
```

{**B**} Basic doesn't display exactly what you entered; it rounds the number to '1.01234568'. Basic stores numbers in a form that limits their precision to nine decimal places. If a number can't be represented exactly in nine decimal places, Basic rounds it to the nearest nine-place number.

Try displaying other numbers with zeroes before the decimal point, as well as after it. (For the moment, avoid very large numbers and very small decimal fractions.) You should be able to predict how Basic will display each of the numbers you enter.

Fingers Tired? {B}

Here's a useful hint: Basic accepts '?' as an abbreviation for the word 'print'. For example, in place of 'print 229.5', you can type '? 229.5'.

Knowing this will save you a lot of key strokes as you learn to program.

INTERLUDE: USING THE SHIFT KEYS, AND OTHER MATTERS {H}

In the next section we're going to start using the PRINT statement to display the results of calculations. First, to enter arithmetic symbols like '+' and '-', we must learn how the HHC's shift keys work.

The HHC has two shift keys. They are labelled 'SHIFT' and '2nd SFT' ("second shift").

SHIFT Key {H}

The SHIFT key is meant to be used only to capitalize the alphabetic keys, 'A' through 'Z'.

You use the SHIFT key by pressing it, releasing it, and then pressing the key you want to shift. SHIFT affects only the next key that is pressed after it.{3}

Press the SHIFT key, release it, and then press the 'P' key. You should get an upper case 'P' instead of a lower case 'p'.

Now press the 'R' key. You get a lower case 'r', since you didn't press the SHIFT key first.

Press SHIFT again, then 'I'. Notice that the 'SHIFT' blip goes on when you press SHIFT, and stays on until you press 'I'. This blip tells you that the next key you press will be shifted.

{3} - You can also hold the SHIFT key down while you press several other keys, as you would on a typewriter. But beware -- this will shift every character up to and including the first one entered ***after*** the SHIFT key is released! Do you see why this is consistent with the way the SHIFT key works on a single character?

By now you should have the following characters on the LCD:

```
>PrI
```

{B} Finish the rest of the word 'print', type in a number, and press ENTER. See how the HHC accepts 'PrINT' or 'PrINt', just as happily as it accepts 'print'. **Basic statements may be entered in upper or lower case; the HHC does not care**.

What if you press the SHIFT key, and then decide you don't want to shift the next character? Just press SHIFT again. The SHIFT blip will go off and the shift will be cancelled. Try it.

Now that you know how the SHIFT key works (and now that you know you don't need it) you can ignore it until you get to later chapter where we learn about situations in which SHIFT is useful.

{H} The 2nd SFT Key

The 2nd SFT ("second shift") key produces the characters inscribed in the upper-right area above each key.

2nd SFT works the same way as SHIFT: you shift one key by pressing 2nd SFT, then the key that is to be second-shifted.

Let's use the 2nd SFT key to perform a simple calculation. Enter

```
Print 15+5
```

The '+' sign is on the 'Y' key. To enter '+', press 2nd SFT, then 'Y'. End the statement by pressing ENTER, and watch the HHC display the result: 20.

Enter another PRINT statement with a '+' in it. When you press 2nd SFT, notice that the '2nd SFT' blip goes on. This blip tells you that the next key you press will be second-shifted.

If you press the 2nd SFT key and then decide that you don't want to second-shift, you can press 2nd SFT again to return to lower case; or , you can press the SHIFT key to go directly into upper case. (You can also use the 2nd SFT key to go directly from upper case to second-shifted case.)

NEGATIVE NUMBERS

Now that you know how to enter a minus sign (it is the second-shifted 'U' key on the HHC keyboard), you can enter negative numbers, as well as positive ones. Try the following

statement:

```
Print -25.3
```

The HHC obediently displays '-25.3'.

DOING SOME CALCULATIONS

Now let's learn to do arithmetic with the PRINT statement. We've already tried addition,

```
Print 15+5
```

and seen that it works. Let's try subtraction:

```
Print 15-5
```

This gives the result you would expect, too.

To do multiplication, use the '*' symbol (the second-shifted 'M' key), ***not*** the 'x' symbol (on the 'I' key[4]). The '*' is almost universally used for multiplication in computer programming languages:

```
Print 15*5
```

To do division, use the '/' symbol (the second-shifted '?' key), ***not*** the '÷' symbol (on the 'O' key[5]). Again, this use of '/' is almost universal.

```
Print 15/5
```

EXPRESSIONS

Statement parts like '15+5' and '15/5' in the examples above are called "expressions." An **expression** is a group of values (like '15' and '5') joined together by **operators** (like '+' and '/') in a valid way.

You can write more complicated expressions if you wish. For example, you can write an expression that adds several numbers:

```
Print 15+293.17+5+82.3
```

or multiplies several numbers:

```
Print 195*67*44
```

You can write an expression that subtracts or divides several numbers, or does any combination of operations in one

[4],[5] - The '×' and '÷' symbols are used for multiplication and division in the HHC's calculator program. In most other programs, including Basic, they duplicate the function of the SPACE key.

statement:

```
Print 25*8-4-2
```

You can perform exponentiation with the operator '^'. For example,

```
Print 7^2
```

displays 49, which is 7^2, or 7*7.

```
Print 2^4
```

displays 16, which is 2^4, or 2*2*2*2.

Non-integer exponents are allowed; for example:

```
Print 2^4.3
```

displays 19.6983106, which is the approximate value of $2^{4.3}$.

Negative exponents are also allowed; for example:

```
Print 2^-4
```

displays .0625, which is the value of 2^{-4}(1/16).

The Order of Arithmetic Operations

Look at the PRINT statement in the last example, above. Notice that the result depends on the order in which the operations are done.

Basic has rules that determine the order of operations for every valid statement. Computer programmers call these rules **operator precedence** rules, since they determine which operators ('+', '/', '*', etc.) take precedence in a calculation.

If you are familiar with mathematics, you will find that Basic's operator precedence rules are the same as the ones you are used to. Here are the rules:

1. Negation ('-', used to express a negative number) has the highest precedence; that is, it is performed first.
   Example: in '5*-3', '-' is performed before '*'.
2. Exponentiation, '^', has the next precedence.
   Example: in '5*2^{-4}', '-' is performed first, then '^', then '*'.
3. Multiplication and division have the next precedence.
   Example: in '5+3*8', '*' is performed before '+'. (The result is 29.)
4. Addition and subtraction have the next precedence.
5. If two or more consecutive operations have equal precedence, they are performed left-to-right.

Examples: in '15-2-1', 2 is subtracted from 15, then 1 is subtracted from the difference. (The result is 12.) In '15/3*2', 15 is divided by 3, then the quotient is multiplied by 2. (The result is 10.)

Overriding the Normal Operator Precedence

If you want an expression to be evaluated in some order other than the one defined by Basic's operator precedence rules, put parentheses around the part of the expression Basic is to evaluate first.

For example, consider the following statement:

```
Print 15*12+1/2-3-5
```

Here is how we make Basic evaluate '12 + 1/2' first, getting '12.5', and then evaluate '15*12.5-3-5':

```
Print 15*(12+1/2)-3-5
```

Note that ***inside*** and ***outside*** the parentheses, normal precedence rules apply. In the expression above, Basic would evaluate '1/2' first, then add 12 to the quotient. ('/' has higher precedence than '+'.) Then Basic would multiply the sum by 15 ('*' has higher precedence than '-'), then subtract 3 (equal-precedence operations are performed left-to-right), then subtract 5. The result would be 179.5.

Where would you insert parentheses in the statement

```
Print 15+3*15-3/5*3.1415926
```

so that Basic will do the addition, the subtraction, and multiplication by pi first, then do the other two multiplications, then do the division? Try it on your HHC; does your solution work? The result it should produce is 13.7509873.

You can write expressions that use parentheses inside parentheses if you wish. For example, suppose you want to calculate the value of this expression:

$$1+\cfrac{1}{2+\cfrac{1}{3+\cfrac{1}{18}}}$$

You can do it like this:

```
Print 1+1/(2+1/(3+1/18))
```

We say that this expression has two **levels** of **nested** parentheses.

But be cautious: expressions with too many levels of parentheses are usually very hard to read (as this one is!). Your programs will be clearer if you divide complex expressions into several simple steps, and perform each step in a separate statement. You will learn how to do this in the next chapter.

{H} TURNING THE HHC OFF

We have covered a lot of ground in this chapter, and you have learned almost everything there is to know about writing numeric expressions. Before we end the chapter, let's take a look at one more remarkable feature of the HHC.

Enter ***part*** of a PRINT statement. Then turn the HHC off by pressing the OFF key at the right edge of the keyboard.{6} On any other computer, this would erase everything in the computer's storage; when you turned the machine on the next time, it would be just as if you were turning it on for the first time.

But turn the HHC back on by pressing the ON key. There's your partially entered statement, just as it was when you pressed OFF! You can turn the HHC off at ***any*** time, and whatever operation the HHC is performing will continue undisturbed when you turn it back on.

THE AUTO-SHUTOFF FEATURE

The HHC will automatically turn itself off ten minutes after the last program input or output, or ten minutes after the last key is pressed. If a Basic program runs continuously with no I/O or key interrupt for longer than ten minutes, the HHC will not automatically shut off.

The **auto-shutoff** feature is designed to preserve the HHC's batterys in case you absentmindedly leave the HHC on when you are done usng it.

To turn the HHC on again, just press the ON key.

{6} - Not the ALL OFF switch on the back of the HHC. Remember, this switch stays ON at all times.

CHAPTER 3: VARIABLES

INTRODUCTION TO VARIABLES

So far we've been writing expressions that consist entirely of constant numbers like 517 and 3.1415926. Clearly, this is limiting. The computer programs we're going to write have got to manipulate different values at different times.

Let's see how we can do this in Basic. Turn your HHC on; type in the following Basic statement and press ENTER:

```
x=5
```

('=' is the second-shifted 'P' key.)

When Basic executes this statement, it creates a **variable** named "X" and assigns it the value 5.

Now enter the following statement:

```
Print x
```

and Basic displays the value of the variable X, which is 5.

Every variable has a ***name*** and a ***value.*** For example, we have just looked at a variable whose name is X, and whose value is 5.

When we write programs, we refer to variables by name. When we ***run*** programs, Basic manipulates the ***values*** of the variables, as if we had entered those values as constants. In our example above, since the value of X is 5, the statement 'print x' has the same result that the statement 'print 5' would have.

Try entering the two statements:

```
x=-8.2
Print x
```

Now Basic displays '-8.2', instead of '5'. By executing the statement 'x = -8.2', we have changed the value of X to 8.2. The former value, 5, is gone without a trace. This is why we call X a "variable;" its value changes -- varies -- every time we execute a new 'x = ...' statement. (This kind of statement is called an **assignment statement**, because it assigns a new value to a variable.)

NAMING VARIABLES

Whenever you write a program, you must choose names for the variables you plan to use. In naming Basic variables, you

must observe the following rules:

1. The first letter of a variable name must be a letter. (Upper and lower case, remember, are equivalent.)
2. Every following letter of variable name must be a letter or a numeral.

{**B**} 3. A name may be any reasonable length, but Basic ignores all characters beyond the first two. For example, the variable names HE, HEAP, HERKIMER, and HEFFILTRUP are all the same as far as Basic is concerned, since they all begin with the same two characters.

{**B**} 4. A variable name may not be or contain a reserved word, such as 'print'.

Here are some reserved words that are liable to appear in your variable names if you don't guard against them:

ABS	IF	OR
AND	INT	POS
CONT	LEN	REM
DATA	LET	RUN
DEF	NEW	TAB
END	NOT	TO
FOR	ON	VAL

You can find a complete list of Basic reserved words in the ***Reference Guide,*** Chapter 4.

You can write more readable programs if you give your variables names that are connected with their uses in the program. For example, in a program that computes compound interest over a period of time, a variable to hold the length of time in months would be better named MNTHS, or at least MN, rather than just M or (horrors) X.[1]

THE INITIAL VALUE OF A VARIABLE

Try PRINTing a variable you have never assigned a value to, such as ZQ. You should get 0 (zero).

Every variable that you use in a Basic program has an **initial value** of zero. You can often take advantage of this fact. For example, if you are writing a program that computes the sum of a series of numbers, you can safely assume that the variable you use to accumulate the sum has a value zero before you add the first number to it. Thus, you need not write a statement like 'SUM = 0' at the start of your program.

---

[1] - A variable couldn't be named MONTHS, because MONTHS contains the reserved word ON!

VARIABLES AND EXPRESSIONS

You can assign the value of an expression to a variable, just as you can PRINT the value of an expression. Try this:

```
x=(15+3)*(15-3)/5
Print x
```

these two statements display exactly the same value as:

```
Print (15+3)*(15-3)/5
```

Variables can appear in an expression, as well. Try this:

```
a=15
b=3
x=(a+b)*(a-b)/5
Print x
```

It should give you the same result again.

X = X + 1. . .?

Consider the following statement:

```
x=x+1
```

This statement takes the current value of X, adds 1 to it, and assigns the sum to X as its new value.

Notice that as a mathematical statement, 'x = x + 1' is absurd. There is no way a mathematical variable could possibly be equal to itself plus 1. This points up an important difference between the meaning of the symbol '=' ***in mathematics*** (particularly in algebra) and its meaning ***in Basic.***

In mathematics, '=' represents ***statement of fact.*** 'x = y + 1' means, "X is equal to Y + 1; whatever the value of Y happens to be, the value of X is 1 greater."

In Basic, '=' represents an ***assignment operation.*** 'x = y + 1' means, "take the value of Y, add 1 to it, and assign the sum to X."

There is also an important difference between the concept of a variable in algebra and in Basic. In algebra, we deal with equations like '2x + 1 = 25', where the "variable" X has some fixed, pre-existing value. Our task, if we wish to solve the equation, is to ***find the value.***

In Basic, we deal with statements like 'x = 2*X + 1', where X has one value before the statement is executed, and a

different value afterward. A computer's task, if it is commanded to execute such a statement, is to ***create a new value.***

In order to avoid being confused about what a computer program does, keep these differences clear in your mind as you progress through this book.

PERFORMING A CALCULATION IN STEPS

Remember the mathematical formula and the corresponding Basic statement that we used to illustrate the idea of nested parentheses?

$$1+\cfrac{1}{2+\cfrac{1}{3+\cfrac{1}{18}}}$$

corresponds to:

```
Print 1+1/(2+1/(3+1/18))
```

Let's use our knowledge of variables to calculate the value of that formula in a more readable way.

We begin at the place where we would begin computing the value by hand: the innermost part of the formula, '1/18'. We will work outward from there until we have calculated the value of the whole formula.

What is the largest part of the formula that we can write in a thoroughly legible way? For most of us it will be '3 + 1/18'. We write a statement that computes that part of the formula:

```
x=3+1/18
```

Now we can simplify the formula by substituting X for the part of the formula that X will represent after the statement above has been executed:

$$1+\cfrac{1}{2+\cfrac{1}{X}}$$

What is the largest part of the new, simplified formula that we can write legibly? Probably '2 + 1/X'. Let's write this as a

statement after the first one:

```
x=3+1/18
x=2+1/x
```

The formula is simplified again:

$$1+\cfrac{1}{X}$$

Think about what we have here. There are two Basic statements, which we intend to execute ***in order.*** The first statement computes an intermediate value representing part of formula. The second statement computes a second intermediate value representing a larger part of the original formula. The new formula is equivalent to the first one, with X substituted for the expression whose value it will contain after our second statement is executed.

Advancing one more step, we can represent the whole formula with this set of three Basic statements:

```
x=3+1/18
x=2+1/x
x=1+1/x
```

If we execute these three statements in order, and then

```
Print x
```

we should get the same value that we got from

```
Print 1+1/(2+1/(3+1/18))
```

(Do we? Try it.)

WHAT IS PROGRAMMING ABOUT?

There are two important lessons to learn from the exercise we just went through.

1. A complicated expression (or statement, or procedure) can usually be broken down into a series of simpler expressions, or statements, or procedures. This makes each part easier to understand, and so makes the whole easier to understand.
2. A complicated problem can often be broken down in the same way. If you can figure out the right way to break a problem down, it becomes easy to solve, and its solution is easy to understand.

After you master the "ABC's" of Basic, most of your programming time will be spent figuring out how to describe the solutions to problems so that a Basic program can solve them for you. The best way to do this is to break a problem down systematically into smaller and smaller parts, until the solution to each part of the problem becomes clear.

Smart computer programmers are not those who can write complicated programs; they are those who can write ***simple*** programs, even when they are solving problems that seem complicated.

CHAPTER 4: WRITING A STORED PROGRAM

WRITING A PROGRAM

Up to now, we have used Basic as a sophisticated calculator. We have typed in numbers and operators, and it has calculated a result when we pressed ENTER.

Now we're going to start writing computer programs. We're going to type in statements that Basic will ***hold in storage*** rather than ***execute.*** When we have typed in several statements, we will tell Basic to execute them all at once. Those statements will make up a program. We will be able to execute the same program as many times as we want, with or without changes.

Memory, Storage, and Some Other Terms

Before we begin, we will introduce some terms that relate to the HHC's facilities for remembering things.

We will use the term **memory** to describe any part of the HHC that can store information. We may also apply the term "memory" to information storage in a peripheral device.

A computer has two kinds of memory. One is **random access memory** (**RAM** for short), which can be used to store and retrieve information. The other is **read-only memory** (**ROM** for short), in which the computer's manufacturer stores unchangeable information that is needed to run the computer.

We will use the term **storage** to describe RAM in which you can store, edit and run Basic programs. The HHC has some built-in storage, called **intrinsic RAM**. Depending on the model of HHC you have, the intrinsic RAM may have room for approximately 1,000 to approximately 3,000 characters of storage. You can give your HHC an even larger, additional storage area by plugging in a **Programmable Memory Peripheral**.

Kinds of memory that are ***not*** "storage" are the ROM that holds the HHC's fundamental operating programs; the ROM in HHC capsules; and any memory in a peripheral device other than a Programmable Memory Peripheral.

Getting Started

Let's write a simple computer program on paper. This is actually something we've already done. Any two Basic

statements that perform a meaningful task when executed in order, make up a program. Here's the very simple program we're going to work with:

```
x=67
Print x
```

Turn on your HHC. Enter the two-statement program we just wrote -- not as it is shown above, but with the numbers 10 and 20 in front of the two lines, like this:

```
10 x=67
20 Print x
```

Notice that when you finish entering the program, the LCD does not display a 67. Basic hasn't executed either statement yet.

Next, type the command

```
run
```

and press ENTER. Now Basic executes the program and displays the value 67.

As you entered the statements, Basic ***stored*** them. When you entered the command RUN, Basic executed (ran) the program.

Enter RUN a few more times. Watch Basic execute the program each time you enter RUN.

What Happened

Basic stored the statements instead of executing them because of the **line numbers**, 10 and 20, that preceded the statements. A line number before any Basic statement means, "don't execute this statement now; store it in the program."

When Basic is executing a stored proram, we say it is running in **deferred mode**, because execution is deferred from the time you enter the program until the time when you enter RUN. When you enter a statement without a line number, Basic executes it in **immediate mode**; that is, Basic executes the statement immediately when you press ENTER.

Notice that we called RUN a **command** rather than a statement. That was because RUN is customarily used only in immediate mode. We reserve the term **statement** for lines that customarily may be used in programs.

We often refer to the statements in a program, collectively, as **code**. The process of writing a program, as distinct from designing one or correcting errors in one, is called **coding**.

Adding Statements To a Program

Enter the following two statements:

```
8 Print x
6 x=63
```

and type RUN again. Basic displays 63, then 67.

When you entered statements preceded by the line numbers 8 and 6, Basic stored the statements in the program in line number order. Thus, the program ended up looking like this:

```
6 x=63
8 Print x
10 x=67
20 Print x
```

Do you see why the program did what it did?

Limit On the Length Of a Line

You cannot create a line in a Basic program that is more than about 80 characters long. If you try, Basic will truncate the line.

Long program lines are hard to enter and read in any case, so it is good practice to keep your program lines much shorter than the limit.

LIST: REVIEWING THE CONTENTS OF A PROGRAM

If you make many changes to a program after entering it, you will have trouble remembering what statements are in the program. Therefore, Basic has the **LIST** command, which lists the current contents of a program on the LCD, one line at a time.

To list your program, simply enter

```
list
```

and press ENTER. The first line of your program, '6 X=63', appears on the LCD.

To display the next line of the program, press the ➧ key. (It is {**B**} two rows above the ENTER key.) Basic displays the next line

of your program, '8 PRINT X'. Press ➧ twice more to see the remaining two lines of your program. Press ➧ one more time to get back to Basic's "enter a statement" prompt, '_'.[1]

Notice that although you typed your program in lower case, Basic lists it in upper case. Basic does this to almost all parts of a Basic program. From here on we're going to show Basic programs in upper case in this book, too.

Basic also inserts blanks in some places in your program,and deletes them in other places. For example, whether you enter

```
20 Print x , y , z
```

or

```
20Print x,y,z
```

Basic will display the statement as

```
20 Print x,y,z
```

This feature of Basic is intended to make your programs efficient to store, and at the same time give them a uniform, readable appearance. You can take advantage of it, if you wish to do so, by entering a program with no blanks at all to minimize the number of keystrokes you have to enter. Basic will put blanks in all the right places as it stores the statement.

Listing Part Of a Program

You need not start listing a program at the very beginning. You can start at any line.

To list a program starting at some line after the beginning, enter LIST followed by the number of that line. For example, to start listing your program at line 8, enter

```
list 8
```

Try this. After line 8 appears, press ➧ three times to see line 10, line 20, and the '>' prompt.

If you enter LIST with a line number that does not exist, Basic will start listing your program at the next larger line number that does exist.

{B} You can also ***stop*** listing a program after any line, not just at the end. To stop listing a program, press ENTER instead of ➧. Basic will return to the '_' prompt immediately.

Try this. Enter LIST; after line 6 appears, press ➧ to see line 8, then press ENTER to stop the listing.

[1]- **Note to old HHC hands**: you cannot use the ➧ key to display the preceding line of the program, as you can in the file system editor. To see a preceding line, return to the '>' prompt by pressing ENTER, and type in an appropriate LIST statement.

Replacing a Line

To replace a line in a program, simply type in a new line with the same line number.

Changing the Contents Of a Line {B}

Basic allows you to change the contents of any line in a program. To change the contents of a line, LIST the line, move the cursor left to the part you want to change, and type new characters over the characters that are already there.

Let's try this. Enter LIST to list line 6 of your program:

```
6 X=63
```

cursor is here

Use the ⬅ key to move cursor left two spaces:

```
6 X=63
```

cursor is here

Change the line to '6 X = 53' by pressing the "5" key:

```
6 X=53
```

cursor is here

Press ENTER. Now list your program again. Line 6 says 'x = 53'. Run your program. Line 8 displays '53'; line 20 displays '67', as before.

Enter "LIST 10' to list line 10 of your program, and change the '67' to some other number. Now end your editing of this line by pressing ➧, not ENTER. List your program again; you will find that ***the contents of line 10 has not changed.*** When you end an editing operation with '➧', that means, "I've changed my mind; I don't want to modify this line after all." Only ENTER stores the changed line in your program.

{B} Moving the Cursor Right

You can move the cursor to the right without changing the text

by pressing the ▶ key, just as you can move the cursor to the left by pressing the ◀ key. (But you can't move the cursor right of the rightmost character in a statement with ▶, any more than you can move the cursor left of the leftmost character with ◀. Try it and see what happens.)

Deleting a Line

Sometimes you don't want to change the contents of a line; you want to take the line out of your program completely. You can do this by replacing the entire contents of the line (except for the line number) with spaces.

Try deleting line 6 in this way. Now list your program. Line 6 is gone, line number and all.

Since you can replace a line by entering a new line with the same line number, you can delete a line by entering a "line" that consists of a line number and nothing more. This is convenient if you happen to know the number of the line you want to delete without using LIST.

Note: to delete a line, you must leave the line number, and the line number only. If you replace the line number with spaces, too, Basic will ***not*** delete the line. (Try it.)

Copying a Line

You can copy a line from one place in your program to another by changing its line number.

For example, consider your program as it now stands, with line 6 deleted. Let's get line 6 back, not by typing it in again, but by making a copy of line 10.

Enter 'LIST 10'. Basic will display line 10 like this:

```
10 X=67
```

cursor is here

Move the cursor back to the line number, and change it from 10 to 6:

```
10 X=67
```

cursor is here

```
6  X=67
```

cursor is here

Press ENTER. List your program again. Now it should look like this:

```
6 X=67
8 PRINT X
10 X=67
20 PRINT X
```

LIST your program again, and change line 6 from 'X = 67' to 'X = 63'. Make Basic accept the change you have made by pressing ENTER. LIST your program again. Is it back in its original form?

The Auto-Repeat Feature {H}

You don't have to press the ◀ or ▶ key over and over to move the cursor a long way through a line. Just hold a key down, and after a half second or so it will **auto-repeat**; that is, it will enter characters until you let go.

All of the HHC's character-typing keys can auto-repeat, too. This is useful, for example, when you want to replace a long statement with spaces.

More About Editing

You will learn more advanced ways of editing a program in a later chapter. For now, you know enough to write and edit any program you want.

SAVING A PROGRAM IN A FILE {B}

Basic allows you to work on only one program at a time. When you are done working on a program, you must **save** it before you can work on another program, or use your HHC for some other task.

To save a program, enter the command **BYE**:

```
bye
```

When you enter BYE, Basic saves your program in its file system, under the name that you chose when you first entered Basic. Then it returns you to the Basic menu. Notice that the menu now says

```
1=New file
2=Program 1
```

or whatever you named your program

The 'No file' line in the menu is gone, since there now is a file: the one containing your program.

There are several things you can do next:

1. You can press the '1' key to create another program with a different name.
2. You can press the '2' key to resume using the program you just saved.
3. You can press the CLEAR key to leave Basic's menu and return to the HHC's primary menu, so that you can use some program other than the Basic interpreter. (Press CLEAR only when you see Basic's menu, ***not*** when you see the '>' prompt!)

Try pressing '2' to resume using "program 1". Do a LIST to verify that you do, indeed, have your program back again. Enter BYE again to return to the Basic menu.

Pick selection 1, if you wish, and create another program. Get some practice in writing and editing Basic programs, and try switching back and forth between your two programs. Then return to the Basic menu and then press CLEAR to return to the HHC's primary menu.

{B} On the Consequences Of Not Using BYE

If you are already familiar with the HHC, you may be tempted to return from your program to the Basic menu by pressing CLEAR, as you would do in most other HHC programs. Resist this temptation at all costs!

Pressing the CLEAR key once returns you to Basic's '>' prompt, not to the Basic menu. Pressing CLEAR twice returns you to the primary menu, as you would expect; but you will then find that the HHC's file storage appears to be full. You will be unable to edit any file.

In a later chapter, you will learn how to restore file storage to a healthy state if you should accidentally press CLEAR. (If you need to learn that technique fast, look it up in the index under BYE!)

{B} On the Relation Between Basic and the File System (A Note For Old HHC Hands)

The file system maintains a **file type** field for every file that it stores. A text file created by the file system is one type of file; a Basic program file is another type.

The Basic menu shows only Basic program files. Thus, you will not see any files you have created with the file system when you look at this menu.

The file system can edit only text files. You can edit a Basic program only with the editor built into Basic.

VARIABLE VALUES AND RUN

It seems logical to pass data to a program by assigning values to variables the program will use, and then executing the program with RUN. Unfortunately, that doesn't work. When you RUN a program, Basic sets all the variables to their initial value of 0 before executing the first statement.

This habit of Basic is actually quite useful. It means that if a program depends on the initial value of a variable being 0, you can run the program twice in a row, and the second run will produce the same results as the first.

THE INPUT STATEMENT

There's an easy way to get data into a program's variables. In fact, it's easier than typing an assignment statement. It is the **INPUT** statement.

Here is a program that uses the INPUT statement to read two numbers, and displays the difference:

```
10 INPUT N1
20 INPUT N2
30 PRINT N1-N2
```

When this program is executed, the INPUT statement in line 10 halts and displays '?' on the LCD. This is INPUT's prompt. INPUT waits for you to type a number and press ENTER. When you do, INPUT assigns the value of the number you type to N1.

Similarly, the INPUT statement in line 20 halts and prompts you with '?' to type a number, and assigns the value of the number you type to N2.

Finally, the PRINT statement in line 30 calculates the difference between the two numbers you entered, and displays it.

Input gets information into your HHC, and output gets information back out. Input and output are closely related in many ways. For example, they are both concerned with peripheral devices, and they both are concerned with converting numbers between external (keyboard or LCD) and internal (storage) formats. When we discuss such matters, we often refer to input and output as a single topic called input/output, or **I/O** (pronounced "eye-oh") for short.

INPUT With Multiple Variables

One INPUT statement may input data into any number of variables. List the variables after INPUT with commas between them, like this:

```
10 INPUT N1,N2
```

When INPUT prompts you for data, type the proper number of values with commas between them, like this:

```
63,17
```

If you enter ***too few*** numbers, INPUT assigns the numbers you do enter to the first variables in the list. When INPUT runs out of numbers, it prompts you again with '??', meaning "that's not enough; give me more numbers." It assigns the values you enter to the next variables in the list, and prompts you with '??' again until you have entered enough numbers to fill all the variables in INPUT's list.

If you enter ***too many*** numbers, INPUT simply discards the excess ones.

If you enter a line of numbers that includes ***an invalid number,*** INPUT discards the whole line. It does not assign a value to any variable. It displays the message

```
Reenter
```

and prompts you again.

INPUT Works In Deferred Mode Only

You cannot use INPUT in immediate mode, only in deferred mode.

If you try to use INPUT in immediate mode, Basic gives you the error message

```
ID error
```

meaning, "immediate error; you can't use this statement in immediate mode."

INPUT With a Prompt

It would certainly be useful if you could make an INPUT statement that could prompt you with a message of your choice in place of '?', which tells you nothing about what INPUT wants. You can write an INPUT statement that prompts you, like this:


```
10 INPUT "1st & 2nd values";N1,N2
```

This statement displays the following prompt:

```
1st & 2nd values?
```

Notice that INPUT adds a '?' to the end of the prompt.

The two quotation marks indicate the start and end of the text of the prompt.

The quotation marks and the characters between them form a **string constant**. It is a ***constant*** because it has a fixed value, just like 5 or 3.141592. It is a ***string*** constant because its value consists of a string of characters, rather than a number. In a later chapter you'll learn more about what Basic can do with string values.

Start a new file and type in the difference-of-two-numbers program. Use the prompting version of the INPUT statement, above. Note the ';' that separates the string constant from the list of variables; you must use a ';' here, not a ',', or Basic will give you an error message! Run the program.

Notice that the characters in the prompt are displayed in lower case, the way you entered them. String constants are one place where Basic does not force letters to upper case or add and delete blanks according to its own rules.

CONCLUSION

Now you know enough to write fairly complex calculator-like programs. It's time for you to get some practice in writing and editing Basic programs, if you haven't begun to do so already.

Choose a few tasks that involve doing non-trivial calculations on one or more variables, and displaying the results. Write, enter and execute a program to perform each task. If any of your programs does not work the way it should, figure out why, and correct it.

If you don't have any interesting tasks that you want to use as programming exercises, we suggest the following ones:

- Given four numbers, calculate and display the average.
- Calculate your car's fuel efficiency in miles per gallon, given the odometer readings at two consecutive trips to the pump, and the amount of gas pumped at the second reading.

CHAPTER 5: MORE ABOUT BASIC

Now you have been introduced to most of the fundamental concepts of programming in Basic. This chapter discusses some more advanced "nuts and bolts" aspects of programming that will come in handy as you work with Basic.

ERROR MESSAGES IN A BASIC PROGRAM

You have already encountered error messages. When you entered an invalid statement in Chapter 2, you got an error message that said 'SN error' (for "syntax error"). You've probably gotten this message more than once, and you may have gotten other messages too, when you entered erroneous statements by accident. (Remember, everyone does this frequently while learning to write programs, and it doesn't hurt anything.)

Let's see what happens when Basic finds an error while executing a program in deferred mode. Turn on your HHC and enter Basic, if necessary; select 'program 1' from the Basic menu. If your version of 'program 1' still matches the one developed in this book, it looks like this:

```
6 X=53
8 PRINT X
10 X=67
20 PRINT X
```

We're going to introduce an error into this program on purpose. List line 10 and change it to say

```
10 X=&'
```

('&' and' 'are the second-shifted characters on the '6' and '7' keys, respectively.)

When you press ENTER, telling Basic to accept this "statement," Basic doesn't object. Try listing your program; Basic has stored the erroneous statement in the program, all right. But now try to execute the program. Lines 6 and 8 execute properly, and line 8 displays the value of X, 53. But when Basic gets to line 10, the one containing the error, it gives you the error message

```
SN error in line 10
```

The error code, SN, tells you what happened.

The line number, '10', tells you where in your program the

error happened. (You didn't get this when you were entering statements in immediate mode, because the error was always in the statement you had just entered; and there were no line numbers.)

This information gives you a good start on finding the cause of the error.

Here are some error codes you are likely to encounter, and their meanings:

OM-***Out of memory.*** Your program is too large, and/or you are keeping too many programs in the HHC's file system. Before you can continue, you must shorten or delete at least one program, or move the program you are working on to a Programmable Memory Peripheral and continue working on it there. Chapter 7, "Managing Program Files," explains how to move a program to a Programmable Memory Peripheral. (***Note:*** if you decide to delete a program to make space, you can preserve a copy of it in a Programmable Memory Peripheral before doing so.)

OV-***Overflow.*** The result of a calculation is too large to be represented in Basic's internal numeric format. (Such a result must be very large indeed, as you will see when we deal with the range of a variable's value, later in this chapter.)

SN-***Syntax.*** A Basic statement is incorrectly written: parentheses do not match, an operator is used in an improper context, a reserved word is misspelled, etc.

/0 – ***Division by zero.*** You tried to divide a number by zero.

Basic can display several other error codes in situations that you haven't encountered yet. As we present more features of Basic, we will also discuss the error codes that are likely to come up in connection with them.

For a complete list of Basic's error codes, see the ***Reference Guide,*** Chapter 10.

{B} THE RANGE OF A NUMERIC VALUE

You already know that numeric values in Basic are limited in precision to nine digits. They are also limited in range, although the limits are very broad.

The largest number that Basic can represent is about $1.70141*10^{38}$ (that is, 170141 followed by 33 zeroes). The smallest number that Basic can represent is the negative of the largest: $-1.70141*10^{38}$.

There is also a limit to the tiniest ***absolute*** value that Basic can represent: about $2.93874*10^{-39}$ (293874 preceded by a decimal point and 38 zeroes). The tiniest negative value that

Basic can represent is the negative of that: $-2.93874*10^{-39}$

HOW BASIC PRINTS VERY LARGE AND SMALL NUMBERS

Basic does not represent very large and small numbers, like 32,963,000,000,000,000 and .00000000000002911, in ordinary numeric format. If it did, the numbers would be unreadable; who could keep track of all those zeroes?

Basic displays very large and small numbers in **scientific notation**, like this:

```
3.2963E+16
```

(represents 32,963,000,000,000,000)

```
2.911E-14
```

(represents .00000000000002911)

To interpret such a number, multiply the part to the left of the 'E' (the **mantissa**) by 10 to the power to the right of the 'E' (the **exponent**). For example,

```
3.2963E+16
```

(represents $3.2963*10^{16}$)

```
2.911E-14
```

(represents $2.911*10^{-14}$)

An easy way to interpret a number in scientific notation is to shift the mantissa's decimal point right by the number of digits given by the exponent. If the exponent is negative, shift the decimal point left.

The rules that Basic follows when displaying a number are explained in the ***Reference Guide,*** Chapter 5.

You can use scientific notation to describe any numeric constant in a Basic program. You can also use it to enter any numeric value in response to an INPUT statement; ***but note,*** in this situation you must use an upper case 'E' before the exponent. A lower case 'e' is not valid.

MORE ABOUT THE PRINT STATEMENT

Printing Several Values On a Line

So far you have used PRINT to display a single numeric value at a time. Now you're going to learn how to use PRINT for much more than that.

Try entering the following statement in immediate mode:

```
PRINT 25,43
```

Basic displays the numbers 25 and 43 ***at the same time,*** with some space between them.

You can display any number of values on a line by writing a PRINT statement that lists them all, in order, with commas between them:

```
PRINT 1,2,3,4,5,6,7,8,9,0
```

This statement displays ten values with spaces between them. Try executing it. Notice that since it is too long to fit on the LCD all at once, the HHC **scrolls** it past the LCD a character at a time, like a message displayed on a lighted sign.

The values displayed by this kind of PRINT statement may be of any type: constants, variables, or expressions:

```
X=9.999
PRINT 1,X,2*X+5,7.5,7*X+2
```

How Basic Spaces Values

When you display several values on a line, Basic spaces them according to the following rules:

{B} 1. It divides the line into **zones**, each 16 characters wide.

{B} 2. It displays the first value beginning at the left edge of the first zone: that is, in LCD position 1. (Note that the value begins with a sign position, which is '-' if the value is negative and "space" if the value is non-negative.)

{B} 3. It adds one space after the value.

4. It skips to the beginning of the next zone.

5. It displays the second value beginning at the left edge of the second zone, and so forth until all the values have been displayed.

PRINT always displays a numeric value in a field no more than 15 characters wide, so rule #3 might seem a meaningless complication. But hold on; we're about to encounter another kind of displayable value that can be more than 15 characters long, and will give meaning to the rule.

Printing String Constants

You can display string constants with PRINT. A string constant can appear anywhere in a the list of values that PRINT displays. This can be useful if you want to display a

message like

```
Fuel efficiency = 28.6 mpg.
```

When PRINT displays a string value, it displays no leading "sign" character and no trailing space, as it does when it displays a numeric value.

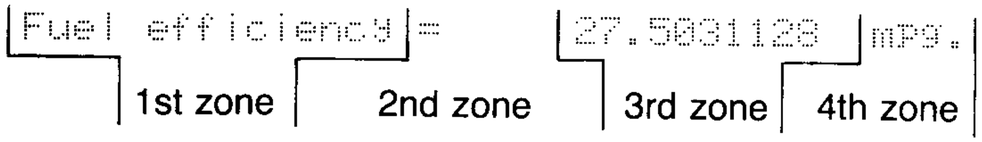
Here is a program that can calculate your car's fuel efficiency program and display a message like the one above. It uses a variable named SR to hold the odometer's starting reading, a variable named ER to hold the ending reading, and a variable named GA to hold the number of gallons of gasoline consumed. A variable named MP holds the calculated result.

```
10 INPUT ""Start & end odometer";SR,ER
20 INPUT "Gallons used";GA
30 MP=(ER-SR)/GA
40 PRINT "Fuel efficiency = ",MP,"mpg."
```

Enter this program into your HHC and run it. Notice that it displays its result with a lot of spaces around the number, something like this:

```
Fuel efficiency =      27.5031128   mpg.
```

Why did this happen? Remember how Basic divides a line of output into 16-character zones! If we printed this program's output and marked the divisions between zones, the reason for the odd spacing would become clear:



The first value displayed by line 40, 'Fuel efficiency =', is just long enough to extend into the second zone. Since PRINT always skips to the start of a new zone when it finds a comma between two values, it puts the number in the third zone. (Why the blank at the start of the zone? Remember that a negative number starts with a '-' and a non-negative number starts with a space.) For the same reason, PRINT puts the third value, 'mpg.', in the fourth zone.

How To Avoid PRINT's Zones

PRINT's habit of aligning values in zones is a nuisance when you want to display a message like 'Fuel efficiency = 27.5 mpg.' Is there a way to make PRINT ignore the zones, and place the beginning of each value right after the end of the

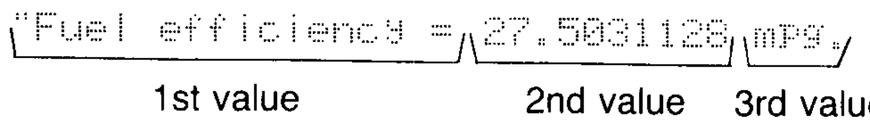
preceding one? Yes, there is. Simply replace the commas in the PRINT statement with semicolons:

```
30 PRINT "Fuel efficiency =";MP;"mpg."
```

Modify your program in this way and try executing it. Does it produce the desired result?

Combining Strings and Numbers In a PRINT Statement

Let's mark off the divisions between the values in the second version of our program's result, as we marked off the divisions between zones in the first version:



The spaces on either side of the number are actually part of the number. The leading space is the "sign" character, which is a space for a non-negative number. The trailing space is the space that follows every numeric value.

SEVERAL PRINT STATEMENTS, ONE PRINTED LINE

Up to now, every PRINT statement you have seen has displayed exactly one line. Sometimes this is not what you want; you want to "build up" a line of displayed output from pieces displayed by several different PRINT statements.

Basic lets you do this easily. To end a PRINT statement without ending the line it displays, simply end the statement with a comma or semicolon instead of a value, like this:

```
PRINT X,Y,Z,
```

or

```
PRINT X;Y,Z;
```

When you write a PRINT statement like this, Basic displays and spaces each value as it usually would. After displaying the last value, it spaces to the start of the next zone if you ended the statement with a comma, or does not space at all if you ended the statement with a semicolon.

The following mileage calculator displays its result in exactly the same format as the one we just tried:


```
10 INPUT "Start & end odometer";SR,ER
20 INPUT "Gallons used";GA
30 MP=(ER-SR)/GA
40 PRINT "Fuel efficiency =";
50 PRINT MP;
60 PRINT "mpg."
```

This suggests one use for ending a PRINT statement with a semicolon or comma; it lets you break a long, complex PRINT statement into several shorter ones. If you were constructing a message out of a dozen or so values, this sort of simplification would be invaluable. You could divide the message among several PRINT statements that each displayed a few logically related values.

By the way, you can use PRINT with ***no*** values to print an empty line, or to end a line built up by several PRINT statements that end with ';'s:

```
30 PRINT "Fuel efficiency =";
40 PRINT MP;
50 PRINT "mpg.";
60 PRINT
```

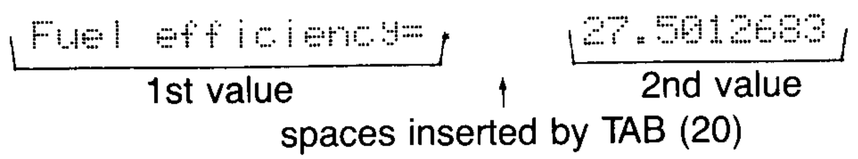
The TAB Function

The TAB function is a special kind of "value" that you can use to format PRINT's output in particular ways. TAB displays enough spaces to make the next PRINTed value appear in a specific position on the LCD. It works very much like the TAB key on a typewriter.

For example, consider this statement:

```
PRINT "Fuel efficiency = ";TAB (20);MP
```

'TAB (20)' displays spaces through LCD position 19, so that the number will appear in position 20.



If Basic would have to move ***left*** to place the next value in the desired LCD position, it ignores the TAB.

TAB is particularly useful when you are printing data to an actual printer, or a multi-line device like a TV Adaptor, instead of the LCD. TAB makes it easy for you to align a column of numbers exactly where you want them, rather than wherever a print zone happens to begin. For example, the following

sequence of statements

```
40 PRINT "Start reading =";TAB (20)▶
   ;SR;TAB (32);"mi"
50 PRINT "End reading =";TAB (20);ER;▶
   TAB (32);"mi"
60 PRINT "Gallons used =";TAB (20);GA
70 PRINT "Fuel efficiency =";
80 PRINT TAB (20);MP;TAB (32);"mpg"
```

might display output that looks like this:

```
Start reading =    35585        mi
End reading =      35822        mi
Gallons used =     11.8
Fuel efficiency =  26.8644068   mpg
```

Note: when you use TAB, you usually will want semicolons, not commas, as value separators. Commas would just mess up the alignment that TAB is intended to produce!

The SPC Function

SPC makes PRINT display a specified number of spaces in a line of output. For example, consider the following statement:

```
PRINT "Fuel efficiency =";SPC (8);MP
```

In this statement, 'SPC (8)' means, "display 8 spaces at this point in the output line." The following statement does exactly the same thing (we are using the symbol 'ø' for "space" so that you can see how many spaces there are):

```
PRINT "Fuel efficiency =øøøøøøøø";MP
```

THE STP/SPD KEY

You can use the STP/SPD ("stop-speed") key to change (1) the LCD's rotation speed, (2) the minimum time that a line of output will remain on the LCD before being replaced, and (3) the speed of the keyboard's auto-repeat feature.

To change the HHC's speed, press STP/SPD. Whatever the HHC is doing, it will "freeze." To set the speed and unfreeze the HHC, press one of the number keys.

The '1' key sets the HHC to its slowest speed when used with STP/SPD; the '2' key sets the HHC to its second-slowest speed, and so on. The '9' key sets the HHC to its second-fastest speed, and the '0' key sets the HHC to its fastest speed.

You can use STP/SPD to "freeze" the HHC if you want to look at something displayed on the LCD for a longer time than the HHC would otherwise display it. To unfreeze the HHC without changing its speed setting, simply press STP/SPD again.

CHAPTER 6: MORE ABOUT EDITING PROGRAMS

So far you've learned how to insert or delete a line in a Basic program, and how to edit a line by moving the cursor left and right. As we noted, there's more to editing than that.

In this chapter you'll see many new editing functions. Unless you have a photographic memory, you are not likely to remember them all. That's OK; you don't have to remember ***any*** of them if you are content with the simple (and time-consuming) procedures you learned in earlier chapters.

As you become more experienced with Basic, you will want more sophisticated editing functions to speed up your work. You can return to this chapter and learn how to use a new editing functions each time you decide that you want to use one. That's a perfectly acceptable way to use this book.

INSERTING A CHARACTER IN A LINE {B}

Suppose you have a line in a program that looks like this:

```
TX=TX+.06*(PP-EP)+ET/PP
```

Suppose '.06' is the wrong number; it should be '.065'. You could change it by typing over everything from the '*' to the end of the statement, but it would be much easier to ***insert*** a '5' between '.06' and '*', pushing the rest of the line one character to the right.

Basic lets you do this by using the **INSERT key**. This key is located just above the ENTER key on the HHC's keyboard. When you press INSERT, the next character you type is inserted under the cursor. The rest of the line moves one position to the right.

Let's try this. Enter the line above into Basic, with a line number before it, so that Basic will hold it as a statement in a program.

Now list the line. Move the cursor over the '*' with the ▶ key. Press INSERT.

The INSERT blip goes on, and the cursor changes from a solid rectangle to a checkerboard one. These are both signals that you are about to insert a character.

Press the '5' key. Watch a 5 appear under the cursor, while the '*' and all following characters move to the right.

Now notice that the INSERT blip has gone off, and the cursor looks like a solid box again. INSERT mode applies to only one character at a time; the next character you type will

replace the '*' unless you press INSERT again.

{B} Inserting Several Characters In a Line

Suppose you want to change a line like

```
PRINT "Fuel ="MP"mPg."
```

to say

```
PRINT "Fuel efficiency ="MP"mPg."
```

You could press the INSERT key once for each character you want to insert, but it would be more convenient to tell Basic, "start inserting characters, and keep on inserting them until I tell you to stop."

Basic lets you do this by using the INSERT key with the **LOCK key**.

Let's try this. Enter the line above into Basic (with a line number). List the line and move the cursor to the '=' in the first string constant. Press LOCK, then INSERT.

The INSERT blip goes on again, and the cursor changes its appearance again.

Type in 'efficiency ' (with a space at the end). Watch the HHC insert the text as you type, pushing the '=' and following characters to the right.

To get out of lock-insert mode, just press the INSERT key again. The INSERT blip goes off, and you are back in "non-insert" mode.

{B} The ◖ and ◗ Keys In Insert Mode

When Basic is in insert mode (or lock-insert mode), the ◖ and ◗ keys do not have their usual functions. They insert spaces at the cursor, pushing the rest of the line to the right.

◗ moves the cursor to the right. ◖ leaves the cursor unmoved, so that a space appears to open up to the right of the cursor as you insert spaces.

Practice using the ◖ and ◗ keys in insert mode to become accustomed to them.

{B} DELETING A CHARACTER IN A LINE

What about the reverse of inserting characters: deleting characters? The HHC lets you do that, too.

To learn how to delete characters, let's reverse the change

we made in the assignment statement above: we'll change '.065' back to '.06'.

List the line we're going to change, and move the cursor to the '5'. Press the **DELETE key** (located to the right of the INSERT key) and then the ◗ key.

Notice that when you press DELETE, the cursor changes from a solid box to a empty box, and the DELETE blip goes on.

When you press ◗, the '5' is deleted; the following characters move one position left to fill the gap. The cursor becomes a solid box again, and the DELETE blip goes off. DELETE, like INSERT, affects only one character at a time.

Re-insert the '5' at the cursor. Move the cursor back to the '5'. Press DELETE and then the ◖ key. Again, the '5' is deleted and the following characters move in to fill the gap; but in addition, the cursor moves ***left one position.***

If this seems odd, remember the following rules:

- You delete a character by pressing DELETE, then ◗ or ◖.
- If you press ◗, the cursor ends up on the character that was to the ***right*** of the deleted character. If you press ◖, the cursor ends up on the character that was to the ***left*** of the deleted character.

Can you lock Basic in delete mode, as you can lock it in insert mode? Yes, you can. Press LOCK, then DELETE. Now, each time you press ◗ or ◖, the character under the cursor will be deleted, and the cursor will go to the next character to the right or left of the deleted character.

Try this. Press LOCK, then DELETE, then play with the ◗ and ◖ keys. Watch Basic delete one character after another from the line.

Notice how DELETE ◖ and DELETE ◗ function when you use them this way. DELETE ◖ deletes the character at the cursor, and then characters to the left, toward the beginning of the line. DELETE ◗ deletes the character at the cursor, and then characters to the right, toward the end of the line. (Try it again.)

To get out of lock-delete mode, press DELETE again.

By experimenting with DELETE, we've thoroughly messed up the line. Can we undo what we've done? Yes; press ◂, as you learned to do when we first introduced LIST, and Basic will leave this statement unchanged. (List it again to assure yourself of that.) When you are done entering a line, you must press ENTER to make Basic accept the changes you have made. Pressing ◂ will always make Basic ignore the changes.

{B} EDITING A LINE LONGER THAN THE LCD

In your experiments with the editing keys, you may have created a line so long that all of it could not fit on the LCD at one time. Basic's editor lets you handle a line up to 80 characters long with little difficulty.

Let's enter a line too long to fit on the LCD. Create a long Basic statement of your own, or enter this one:

```
PRINT "Fuel efficiency =";MP;"mpg.";▶
  GA;"gallons used."
```

After the cursor reaches the last character position on the LCD, it does not go off the LCD; it remains at the right edge, and all the characters you had have entered so far are pushed left to make room for more. The beginning of the line is pushed off the left edge of the LCD. But it isn't lost; it is just invisible for the moment.

Now, suppose you want to edit the first part of the line. Move the cursor left by pressing the ◀ key repeatedly. (You can just hold it down to make the auto-repeat feature work for you.) When the cursor reaches the left edge of the LCD, Basic starts pulling the beginning of the line back onto the LCD, and shifting characters off the right edge to make room.

You can think of the LCD as a little window, and of the line as a big wheel that you can turn back and forth behind the window, allowing you to see the whole wheel a piece at a time. This sort of activity is called **rotation**.

Notice that when the right end of a line is rotated off the right edge of the LCD, the symbol '≡' appears just beyond the last character on the LCD. (There is no corresponding indication that the beginning of a line is rotated off the left edge of the LCD.)

{B} REVIEWING A LINE

To stop the rotation of a line before Basic reaches the end of the line, press ROTATE again, or press any "character" key such as 'L', '9', or 'space.' The line stops where it is, and you can edit it in the usual ways.

To review the entire contents of a long line, just press the **ROTATE key**. Try this now, and watch the results.

Basic treats the line like a ring, with the beginning joined to the end. It rotates the entire statement to the left, bringing the beginning back onto the right edge of the LCD after the end appears. To make Basic stop rotating and reposition the beginning of the line at the left edge of the LCD, press any key.

SOME ADDITIONAL EDITING OPERATIONS {B}

Here are some more editing operations that are less frequently used than those above, but are still useful to know:

- ***Move cursor to start of line:*** press LOCK, then ◀. The cursor moves left at auto-repeat speed. (At high STP/SPD settings the cursor disappears momentarily.) When the cursor reaches the start of the line, the HHC beeps and the cursor stops.
- ***Move cursor to end of line:*** press LOCK, then ▶. The cursor moves right at auto-repeat speed. When the cursor reaches the end of the line, the HHC beeps and the cursor reappears.
- ***Insert multiple spaces left of cursor:*** press LOCK, then INSERT, then LOCK, then ◀. Basic inserts spaces left of the cursor at auto-repeat speed. It stops when you press INSERT again.
- ***Insert multiple spaces right of cursor:*** press LOCK, then INSERT, then LOCK, then ▶. Basic inserts spaces right of the cursor at auto-repeat speed; it stops when you press INSERT again.
- ***Delete from character under cursor to start of line:*** press LOCK, then DELETE, then LOCK, then ◀. Basic deletes characters at auto-repeat speed until the cursor reaches the start of the line.
- ***Delete from character under cursor to end of line:*** press LOCK, then DELETE, then LOCK, then ▶. Basic deletes characters at auto-repeat speed until the cursor reaches the end of the line.
- ***Stopping any auto-repeat operation:*** you can stop any of the "auto-repeat" operations described above at any point, by pressing the key that started it (◀, ▶, INSERT, etc.), or by pressing any key that normally displays a character (including SPACE, ENTER, INSERT, DELETE, and ROTATE).
- ***Going directly from insert mode to delete mode:*** just press DELETE ◀ or DELETE ▶ (or LOCK DELETE ◀, etc). You don't have to cancel insert mode by pressing INSERT first.
- ***Going directly from delete mode to insert mode:*** similarly, just press INSERT, or LOCK INSERT, or etc. You don't have to cancel delete mode by pressing DELETE first.

SHIFTING CASE WITH THE LOCK KEY {H}

You can also use the HHC's LOCK key with the SHIFT key and the 2nd SFT key.

To lock the HHC in upper case, press LOCK, then SHIFT. Notice that when you press SHIFT, the SHIFT blip ***and*** the LOCK blip go on. The LOCK blip is reserved for indicating that the HHC is locked in upper case or in second-shifted mode.

To get the HHC out of upper case, press the SHIFT key again. The SHIFT and LOCK blips go off, and the HHC returns to lower case.

To lock the HHC in second-shifted mode, press LOCK, then 2nd SFT. The 2nd SFT and LOCK blips go on. To get out of second-shifted mode, press 2nd SFT again.

{B} NOTE ON EDITING IN IMMEDIATE MODE

All of the editing functions that we have described in this chapter are usable in immediate mode, too.

It's best, however, to keep immediate-mode statements fairly short. Why put a lot of effort into entering something that will be used once, and then will disappear?

CHAPTER 7: MANAGING PROGRAM FILES

Your HHC has a finite amount of space for storing program files. Sooner or later you will fill that space up. When you do, you will get the message 'NO ROOM, DELETE FILE' when you try to create or modify a program, or you will get the message 'OM error' when you try to run a program.

When you must delete a file, you can avoid losing a useful program by copying the file to a Programmable Memory Peripheral before you delete it. A later section of this chapter explains how to do that.

KEEPING AN EYE ON YOUR FILES

We suggest that you periodically review the files that are stored in your HHC, and delete any that you no longer want. In this way you can delay or avoid the 'NO ROOM, DELETE FILE' message. You can also make your active files easier to manage by eliminating irrelevant things from the Basic menu.

You can tell roughly how much file space you have left by going to the Basic menu or the primary menu, and pressing the **I/O key.** The I/O key interrupts whatever task the HHC is performing, and presents a menu showing the file storage in the HHC's intrinsic RAM, and all the peripherals presently attached to the HHC. The first selection on the I/O key's menu represents the intrinsic RAM; it looks like this:

```
1=INT RAM, 515 FREE
```

The number '515' in this selection is the number of characters of memory still available for you to store programs in.{1}

To leave the I/O menu, press the I/O key again.

HOW TO DELETE A FILE {H}

To delete a file, return to the primary menu. (Remember, enter BYE to save your program, then press CLEAR to leave Basic.) Then pick selection 3, 'File system'. This is the name of an **intrinsic application** program, built into the HHC, that you can use to delete, copy, and rename files.

The file system presents a menu that begins with the following two items:

{1} - You actually cannot use all of this space to store programs; if you did, Basic would have no space left to hold the values of variables when you wanted to run a program!

```
1=NEW FILE
2=COPY FILE
3= . . . . . name of first file
4= . . . . . name of second file
5= . . . . . etc.
```

Find the file you want to delete, and enter its menu selection number. The file system displays the file name, first in normal letters, and then in **inverse video** (clear letters on a black background). It leaves the cursor one character past the end of the name.

Delete the file's name by pressing DELETE, and then ◄. (Or you can delete the name as if it were text on a line in a Basic program by pressing LOCK, DELETE, ◄, ◄, ◄ . . . until the name is all gone.)

Now press ENTER. The file system deletes the file, and then returns you to its menu. The menu is the same as before except that the file you deleted is no longer in it.

{H} {B} HOW TO RECOVER FROM CLEAR WHILE EDITING A BASIC PROGRAM

If you should happen to press CLEAR twice in a row while editing a Basic program, the HHC will act as if its file storage space were completely full. You won't be able to create or edit any file -- not even the one you were working on when you pressed CLEAR.

To recover from this condition, do the following:

1. Press CLEAR to leave Basic.
2. Select the file system from the primary menu.
3. Delete the file named 'B'.{2} B is a file that Basic creates and uses while it is editing your program.
4. Press CLEAR to leave the file system.
5. Return to Basic.
6. Select the file that you were working with when you pressed CLEAR. If you have further editing to do to the file, you may do it now. If you have no more editing to do, just enter the BYE command.

If you were editing a line when you accidently pressed CLEAR, anything you did to that line is lost. Otherwise, your program is in the same condition it was in before you pressed CLEAR.

{2} - **Note to old HHC hands:** this step is usually unnecessary if the current file space is in a PMP rather than the intrinsic file space. If you need or want to delete the 'B' file, however, you will find it in the intrinsic file space, not the current file space.

HOW TO RENAME A FILE {H}

To **rename** a file, go to the file system. Select the file from the file system's menu, just as if you were going to delete it. Use the familiar editing functions to change the name to whatever you want it to be.

HOW TO COPY A FILE {H}

You can use the file system to **copy** a file, that is, to create a duplicate of the file.

Let's make a copy of one of the files you have created while practicing Basic programming.

To copy a file, go to the file system and pick selection 2, 'COPY FILE', from its menu.

The file system presents a new menu containing only the names of files you can copy. Select the file you want to copy from this menu.

Next the file system displays the prompt 'SELECT DESTINATION RAM', followed by another menu. We'll come back to what this menu means in a minute. For now, pick menu selection 1.

The file system copies the file, displays the message 'COPY DONE', and returns to its top-level menu (the one that starts with '1=NEW FILE'). Look at that menu, and notice that it contains ***two*** entries with the name of the file you just copied. One of these is the original file; the other is the copy.

It would be confusing to have two files with the same name; therefore you should immediately rename one of these files (it doesn't matter which one) so that you can tell them apart.

INTRODUCING THE PROGRAMMABLE MEMORY PERIPHERAL {H}

Most computers allow you to attach **peripheral devices** ("peripherals" for short) to them. Peripherals are accessories that you can use to read information from sources other than the computer's built-in keyboard, and write it to destinations other than the computer's built-in display.

The HHC is no exception. It can accept several peripherals, such as printers, TV Adaptors, and modems, which you can use to transfer information in and out of the HHC.

One useful peripheral is the **Programmable Memory Peripheral** ('PMP' for short). This peripheral contains additional storage of the same sort that the HHC uses to store your Basic programs. The Programmable Memory Peripheral, like the HHC's intrinsic RAM, can hold Basic programs in files.

A PMP has many uses. Among them are:

- Holding and running larger programs than your HHC's intrinsic RAM can hold.
- Transferring programs from your HHC to one owned by a friend.
- Storing programs that you do not currently need. This is an important use if you have more programs than the HHC's intrinsic RAM can hold.
- **Backing up** a file you want to preserve; that is, making a copy of the file in a safe place, so that the file will not be lost if you make a mistake in editing it, or if some accident should damage the files stored in your HHC's intrinsic RAM.

{H} Copying a File To a Programmable Memory Peripheral

You can use the file system to copy, rename and delete files in a PMP, just as you use it to copy, rename and delete files in the HHC. Let's learn the procedure by making a copy of one of the files you created while practicing Basic programming.

Save your Basic program, if you are editing one, and return to the primary menu. Press the OFF key, plug your PMP into the HHC's bus socket, and press the ON key.{3}

Go to the file system and pick selection 2, 'COPY FILE'. Pick the file you want to copy from the file system's menu.

As before, the file system displays the prompt 'SELECT DESTINATION RAM', followed by another menu. Look at this menu now. It has two entries, which look like this:

```
1=INT RAM,986 FREE
2=EXT RAM,8183 FREE,SLOT=0
```

(The exact numbers you see in this menu will depend on the amount of intrinsic RAM in your HHC and the amount of RAM in your PMP.)

Each of these menu selections represents one "memory area" that is available for you to copy a file to. (A **memory area** is simply a name for an area of storage, which is capable of storing files.)

{3} - **Be sure to save your Basic program before plugging in the PMP!**

Plugging in a peripheral has the same effect on the HHC as pressing CLEAR twice. Recall that pressing CLEAR twice while you are editing a program will have unpleasant effects on the file system. Plugging in a peripheral has the same effects.

The first entry, 'INT RAM', is your HHC's intrinsic RAM. This is the memory area where you have been storing all your files up to now.

The number after 'INT RAM' is the number of bytes (characters) of RAM that is free for storing new files in the intrinsic RAM's memory area.

The second entry, 'EXT RAM', is the **extrinsic** (not built-in to the HHC) **RAM** in the PMP. The number following is the number of bytes of free RAM in the PMP's memory area.

You want to copy your file to the PMP, so pick selection 2. The file system copies the file to the PMP, displays 'COPY DONE', and returns to its top-level menu.

When You Run Out Of Space

When you run out of space in the memory area, you get the message 'OM error' (if you are in Basic) or 'NO ROOM, DELETE FILE' (if you are not). This is a signal that you had better free up some space in the memory area if you want to continue your work.

The "NO ROOM" message is followed by a menu showing the files in the memory area. You can delete any file by selecting it from the menu. This will usually solve the immediate problem.

If there are no files you are willing to lose, you can return to the primary menu and use the file system to copy one or more files to a Programmable Memory Peripheral, then delete the files.

If you get the "NO ROOM" message right after connecting a peripheral to the HHC, you may have to disconnect the peripheral before you can run the file system. This is because a peripheral uses space in the memory area, and can trigger the "NO ROOM" message if the space it needs is not available.

Managing Files In a Programmable Memory Peripheral {H}

You can do all of the same things to a file in the PMP that you can do to a file in the HHC's intrinsic memory area: you can RUN it, LIST it, copy it, delete it, rename it, and edit it.

To do these things, you must first make the PMP's memory area the HHC's **current memory area:** that is, the memory area that the HHC uses when you perform any operation on a file.

The file system's "DESTINATION RAM" menu shows which memory area is the current one by displaying its name in inverse-image characters.

To select a new current memory area, press the I/O key.{4} Find your PMP on the I/O key menu. Pick the corresponding selection to make the PMP the current-memory-area device. Press the I/O key again to return the HHC to the file system, or to whatever other task it was performing when you pressed the I/O key the first time.

Once you have made the PMP the current device, return to the file system (if you have left it) and look at the menu that shows files. Notice that it contains only the file(s) that you copied to the PMP. None of the files in the HHC's intrinsic memory area are shown, since that is not the current memory area.

You can copy the file(s) in the PMP, rename them, or delete them. If you leave the file editor and enter Basic, you can edit them directly with Basic's program editor.

If you leave Basic by pressing CLEAR while you are editing a program in a PMP, you usually will not have to delete the 'B' file to restore the file space to a usable state.{5}

{H} Recovering a File From a Programmable Memory Peripheral

To copy a file from a PMP back to the intrinsic memory area, simply make the PMP the current-memory-area device, and copy the file, selecting intrinsic RAM as the "DESTINATION RAM."

Note On Recovering From CLEAR

Basic records its 'B' and 'B.' files in the intrinsic file space, regardless of whether or not that is the current file space. Thus, if you should happen to leave Basic by pressing the CLEAR key instead of entering the BYE command, you must make the intrinsic file space the current one in order to delete these files.

{H} Note On Multiple Peripherals

You can attach several peripherals to your HHC at one time if you want to. For example, you can attach a PMP, a printer, and a modem. You can also attach more than one Programmable Memory Peripheral.

{4} - ***Note:*** you ***cannot*** use the I/O key while you are running Basic. You ***can*** use the I/O key while you are running the file system, or most other HHC application programs, or while the HHC is displaying the primary menu.

{5} - see Note 2.

To attach multiple peripherals, you must get an **I/O adaptor**. This is a peripheral which plugs into the HHC's bus socket and has several bus sockets of its own that can hold other peripherals. You can use as many peripherals simultaneously as the I/O adaptor has sockets. (The current model of the I/O adaptor has six sockets.)

For information on how to use other peripherals such as printers with Basic, see the index of this book.

For more information on I/O adaptors, PMP's, and other peripherals, see the HHC owner's manual.

CHAPTER 8: REMARKS IN BASIC PROGRAMS

THE REM STATEMENT

As you begin to write more complex programs in Basic, you will reach a point where it becomes hard for you to remember what they do. Sooner or later you will progress to a point where it is difficult to remember all the details about one of your programs even while you are working on it.

You can deal with this difficulty by keeping notes on what your program does, and updating the notes whenever you change the program. Such notes are generally called **documentation.**

One very convenient way to keep notes is to incorporate them right into the program file. You can do this in Basic with the **REM** statement. REM stands for "remark." A REM statement contains a remark about the program.

You write a REM statement like this:

```
REMremark
```

where "remark" can be any text up to 76 characters long (the 80-character length of a statement, less the length of 'REM' and one blank).

Note: Basic inserts a space between REM and the remark, so that if you enter

```
REMremark
```

LIST will show you

```
REM remark
```

If you enter

```
REM remark
```

LIST will show you

```
REM  remark
```

(with two spaces after 'REM').

Here is a version of our fuel efficiency calculator that illustrates how the REM statement may be used:

```
10 REM Fuel efficiency calculator.
20 REM In: start & end odometer▶
   readings, fuel used.
30 REM Out: miles/gallon.
40 REM Variables: SR=start odom.,▶
   ER=end odom.,
50 REM GA=gallons.
60 INPUT "Start & end odometer";SR,ER
70 INPUT "Gallons used";GA
80 PRINT "Fuel efficiency =";
90 PRINT (ER-SR)/GA;
100 PRINT "mpg."
```

You can put any number of REM statements anywhere in your program. They have no effect on the program's operation when it is executed.

Basic does not squeeze out blanks from REM statements, nor does it translate letters to upper case (except in the word REM itself). You can use spacing and capitalization to make your remarks more readable.

ADVANTAGES IN USING REM STATEMENTS

REM statements have several advantages over other kinds of program documentation.

- Since they are right in a program, they are impossible to lose -- unless you lose the program too.
- They are easy to remember to update, since you can't work on your program without seeing them.
- Since you can put them anywhere you want in a program, you can easily use them to describe what particular parts of a program are doing.

DISADVANTAGES IN USING REM STATEMENTS

The main disadvantage of using REM statements is that they make your program file larger, and so use up more file space.

This disadvantage can be controlled by being making your comments concise, using spaces economically and abbreviating where possible.

If you have to write a lot of information about a program, don't include it in the remarks. Store it elsewhere, and write a remark that ***refers*** to it.

WHAT TO WRITE IN REMARKS

Whenever you write a program, consider including each of the following pieces of information in remarks:

1. The program's purpose.
2. Who wrote the program, and when. (This can be important if other people will be using the program.)
3. How to use the program. (But if the users of the program don't know how to program in Basic, this kind of information should be kept in a separate document!)
4. What variables the program uses. What it uses each variable for.
5. What the overall function is of each part of the program? Begin each part of the program with one or more remarks explaining the program's function.
6. If there are any "tricky" parts in the program where it is not easy to figure out what is happening (or why it is happening), include remarks explaining those parts.

CHAPTER 9: FLOW OF CONTROL

INTRODUCTION TO FLOW OF CONTROL

Up to now we've been working with programs that start executing at the first statement, continue to the last statement, and stop. In computer terms, **flow of control** has gone from the first statement in a program to the last.

Now we're going to look at programs in which the flow of control is more complex.

THE GOTO STATEMENT

Consider the mileage calculator that we developed earlier:

```
10 REM Fuel efficiency calculator.
20 REM In: start & end odometer▶
   readings, fuel used.
30 REM Out: miles/gallon.
40 REM Variables: SR= start odom.,▶
   ER=end odom.,
50 REM GA=gallons.
60 INPUT "Start & end odometer";SR,ER
70 INPUT "Gallons used";GA
80 PRINT "Fuel efficiency =";
90 PRINT (ER-SR)/GA;
100 PRINT "mpg."
```

We're going to modify this program to do any number of calculations in one run. We can do this by adding a **GOTO** statement at the end of the program. (We'll use boldface type to emphasize the parts of the program that we're changing.)

```
10 REM Fuel efficiency calculator.
20 REM In: start & end odometer▶
   readings, fuel used.
30 REM Out: miles/gallon.
40 REM Variables: SR=start odom.,▶
   ER=end odom.,
50 REM GA=gallons.
60 INPUT "Start & end odometer";SR,ER
70 INPUT "Gallons used";GA
80 PRINT "Fuel efficiency =";
90 PRINT (ER-SR)/GA;
100 PRINT "mpg."
110 GOTO 60
```

When Basic executes line 110, it ***goes to*** line 60. That is, the function of the GOTO statement is to transfer control to line 60.

We say that the statements from line 60 through line 110 form a **loop**. When flow of control reaches the end of the loop, the GOTO statement sends it back to the beginning.

Enter the program above into your HHC and run it. See how the the program transfers control back to line 60 every time line 110 is executed.

{B} ENDING EXECUTION OF THE PROGRAM

Now that control goes back from the end of our program to the beginning, how can we make the program stop?

You can halt a program at any time by pressing the **C1** key (in the lower left corner of the keyboard). Basic responds with the message

```
Break in 60
```

where '60' is the line number of the line Basic was executing when you pressed C1.

The C1 key is particularly useful when you run a program that contains an error, and the program does not stop when it should. After you halt the program by pressing C1, you can collect information about what made the program misbehave by noting what part of the program was executing and displaying the values of variables with PRINT.

You can also interrupt your program's execution by pressing the CLEAR key ***once*** (remember not to press it twice while in BASIC). The C1 key is preferable, however. Form the habit of never using the CLEAR key in Basic unless the C1 key does not work for some reason.

SOME GENERAL NOTES ABOUT GOTO

The number in a GOTO statement must be the number of a line that exists in the program. For example, the statement 'GOTO 11' in the program above would produce the error message:

```
US error
```

which is short for "**U**ndefined **S**tatement error." The program has no statement with the line number 11.

But the number in a GOTO statement may refer to a line that contains a REM statement rather than an executable statement. If it does, the REM will do nothing, but control will proceed to the next statement that is not a REM.

INTRODUCING THE IF STATEMENT

Consider the following refinement of our fuel efficiency calculator. We want to calculate fuel efficiency several times, and then calculate an ***average*** fuel efficiency. How could we make our program do this?

First, we need two more variables: one to accumulate the sum of the miles-per-gallon results, and one to count them. When we're done, we'll get the average fuel efficiency by dividing the sum by the count.

Let's call our new variables SU (sum of results) and CO (count of results). Our program looks like this:

```
10 REM Fuel efficiency calculator.
20 REM In: start & end odometer▶
   readings, fuel used.
30 REM Out: miles/gallon.
40 REM Variables: SR=start odom.,▶
   ER=end odom.,
50 REM GA=gallons, SU=sum of results,
52 REM CO=count of results.
60 INPUT "Start & end odometer";SR,ER
70 INPUT "Gallons used";GA
80 PRINT "Fuel efficiency =";
90 PRINT (ER-SR)/GA;
100 PRINT "mPg."
110 SU=SU+(ER-SR)/GA
120 CO=CO+1
130 GOTO 60
```

This doesn't quite accomplish our goal, since it doesn't calculate and display the average. To finish the program we'll need a new statement, the **IF** statement.

```
10 REM Fuel efficiency calculator.
20 REM In: start & end odometer▶
   readings, fuel used.
30 REM Out: miles/gallon.
40 REM Variables: SR=start odom.,▶
   ER=end odom.,
50 REM GA=gallons, SU=sum of results,
52 REM CO=count of results,▶
   CT="continue?" switch.
60 INPUT "Start & end odometer";SR,ER
70 INPUT "Gallons used";GA
80 PRINT "Fuel efficiency =";
90 PRINT (ER-SR)/GA;
```

```
100 PRINT "mPg."
110 SU=SU+(ER-SR)/GA
120 CO=CO+1
130 INPUT "Type 0 to stop, 1 to▶
    continue";CT
140 IF CT=1 GOTO 60
150 PRINT "Average mPg=";SU/CO
```

If CT is 1 when Basic executes line 140, Basic goes to line 60. If CT is not 1, Basic continues executing statements in top-to-bottom order; that is, it goes on to statement 150 and displays the average fuel efficiency.

IF is a very powerful statement. It enables your program to test whether a certain assertion is true, and take two different courses of action depending on the result of the test.

{B} RELATIONAL OPERATORS

In earlier chapters we learned about a variety of operators such as '+' (for addition) and '*' (for multiplication).

Now we're going to consider another group of operators called "relational operators." A **relational operator** makes an assertion about a relationship between two values. If the assertion is true, it produces the result '-1'. If the assertion is not true, it produces the result '0'.

To make this idea clear, take the statement:

```
IF A>B GOTO 45
```

We read the statement like this: "If A is greater than B, go to line 45." The statement is executed like this:

1. The expression 'A>B' is evaluated. '>' is a relational operator that means 'greater than.' If the assertion "A is greater than B" is true, the result is -1; if not, the result is 0.
2. If the value of the expression 'A>B' is non-zero (***i.e.,*** if A is greater than B), the IF statement transfers control to line 45. If the value of 'A>B' is zero, the statement allows control to pass to the next statement in the program.

When we talk about expressions that use relational operators, we often refer to a non-zero value (particularly the value -1) as a "**true**" value, and a zero value as a "**false**" value. Remember that when we speak of ***true*** and ***false*** values, we are really talking about numeric values.

As far as the syntax of a Basic statement is concerned, 'A>B' is interchangeable with 'A+B'. Any place you can use

one, you can use the other. Thus, the following statements are valid:

```
IF A+B GOTO 45
```

and

```
C=A>B
```

Do you see what these statements do?

More Relational Operators

Basic recognizes six relational operators, corresponding to the six possible relations between two values:

operator	meaning
=	equal to
>	greater than
<	less than
<>	not equal to
<=	less than or equal to
>=	greater than or equal to

Notice that '=' is both a relational operator (as in 'IF A=B GOTO 45') and an assignment operator (as in 'B=B+1'). The meaning Basic gives to '=' depends on the context '=' appears in.

All six relational operators are of equal precedence. Their precedence is below that of all the arithmetic operations. For example, in the statement

```
IF A+1=B-C GOTO 45
```

the addition and subtraction are performed first, then A+1 is compared to B-C.

A Little Quiz

Here is a quiz that will help you see if you understand what we've said about relational operators so far.

Suppose A=5, B=4, and C=3. Then:

Q: What will 'IF A=B+C GOTO 45' do, and why?

A: Pass control to the next statement. A is 5; B+C is 7; 'A=B+C' is untrue.

Q: What will 'IF (A=B)+C GOTO 45' do, and why?

A: Go to line 45. A = B is ***false,*** and so returns 0; but 0 + C is 3, which is non-zero, and hence true.

Q: What will 'A = B = C + 1' do, and why? (Be careful; remember the distinction between '=' as a relational operator, and '=' as an assignment operator!)

A: Assign A the value -1. C + 1 is 4; B is also 4; so the value of the expression 'B = C + 1' is -1.

Q: What will 'IF A = B = C + 1 GOTO 45' do, and why? (Be careful again; consider ***all*** precedence rules!)

A: Pass control to the next statement. 'C + 1' is evaluated first; it is 4. The two ='s are both relational operators, and they have the same precedence. Equal-precedence operators are executed left-to-right, so 'A = B' is evaluated first. It returns a 0. '0 = 4' is evaluated next, and also returns a 0.

Note: in a real program, you would not want to write a statement like this one. You would try to think of something else that produced the same result, but was easier to understand.

IF . . . THEN

Basic supports another variety of IF statements that is even more powerful than the one you have just seen. Here is an example of it:

```
IF A=B THEN PRINT  "They're equal."
```

If 'A = B' is true, this statement displays the message 'They're equal.' If 'A = B' is not true, the statement displays nothing. In either case, it passes control to the next statement.

In place of 'PRINT "They're equal."', you can use ***any*** Basic statement. For example, all of the following statements are valid:

```
IF A=B THEN A=0            {an assignment}
IF A>B THEN INPUT C        {an INPUT}
IF A<>B+1 THEN GOTO 45     {a GOTO}
```

To clarify the usefulness of IF. . .THEN, let's add another refinement to our fuel efficiency calculator. We're going to calculate and display the average fuel efficiency only if we've done more than one set of calculations.

We could do this by replacing line 150 with the following:[1]

[1] - **END** is a statement that makes Basic stop running the program when it is executed.


```
150 IF CO<2 GOTO 170
160 PRINT "Average mPG=";SU/CO
170 END
```

But it is shorter and clearer to write the following:

```
150 IF CO>1 THEN PRINT  "Average mPG=
    ";SU/CO
```

SOME VARIATIONS ON THE PROGRAM

If you use the fuel efficiency calculator for a while, you will probably start to notice inconvenient things about it. For example, the program makes you enter the "end reading" for one calculation and the "start reading" for the next even though for several consecutive fill-ups, the two values presumably will always be the same.

If this bothered you, you could modify the program to eliminate the duplicate entry. You would make the program ask for a start reading before the first loop, and the end reading and gallons used for each loop. The end reading for each loop would automatically become the start reading for the next loop.

After such a modification, the program might look like this:

```
10 REM Fuel efficiency calculator.
20 REM In: start & end odometer▶
   readings, fuel used.
30 REM Out: miles/gallon.
40 REM Variables: SR=start odom.,▶
   ER=end odom.,
50 REM GA=gallons, SU=sum of results,
52 REM CO=count of results, CT=▶
   "continue?" switch.
60 INPUT "1st start reading";SR
62 INPUT "End reading this time";ER
70 INPUT "Gallons used";GA
80 PRINT "Fuel efficiency =";
90 PRINT (ER-SR)/GA;
100 PRINT "mPG."
110 SU=SU+(ER-SR)/GA
120 CO=CO+1
122 SR=ER
130 INPUT "Type 0 to stop, 1 to▶
    continue";CT
140 IF CT=1 GOTO 62
150 IF CO>1 THEN PRINT "Average mPG=▶
    ";SU/CO
```

Here's another possible refinement: eliminate the need to answer a 'Continue?' prompt by letting an "end reading" of 0 mean "there are no more calculations." After such a change, the program might look like this:

```
10 REM Fuel efficiency calculator.
20 REM In: start & end odometer▶
   readings, fuel used.
30 REM Out: miles/gallon.
40 REM Variables: SR=start odom.,▶
   ER=end odom.,
50 REM GA=gallons, SU=sum of results,
52 REM CO=count of results, CT=▶
   "continue?" switch.
60 INPUT "1st start reading";SR
62 INPUT "End reading this time (0▶
   ends run) ";ER
64 IF ER=0 GOTO 150
70 INPUT "Gallons used";GA
80 PRINT "Fuel efficiency =";
90 PRINT (ER-SR)/GA;
100 PRINT "mpg."
110 SU=SU+(ER-SR)/GA
120 CO=CO+1
122 SR=ER
140 GOTO 62
150 IF CO>1 THEN PRINT "Average mpg=▶
    ";SU/CO
```

Notice that this version of the program lets you run the program and make no calculations, because the 'IF' is ***before*** the calculation steps, not after. The previous versions forced you to make at least one calculation. In some situations a program must be able to handle a "do it zero times" calculation like this in order to function correctly.

PLANNING PROGRAMS FOR CHANGE

You will often find shortcomings in a program only after the program is in use; and you will often decide to modify a program in the light of such shortcomings, after the program is supposedly complete.

As you write a program, give thought to how it can be made easy to modify. Modify it you will, often many times, before you are satisfied with it!

Enter the final version of our fuel efficiency calculator into your HHC and run it. Do you understand how it works? Can you think of other useful refinements to it?

MULTIPLE TESTS IN ONE "IF"

Suppose you have to perform a two part test like, "If A equals 0 and B is greater than 0, then...?"

Basic lets you write such tests in a very natural way:

```
IF A=0 AND B>0 THEN . . .
```

AND is a **logical operator**. It operates on two values that can be ***true*** or ***false,*** and produces a value that is ***true*** or ***false***.[2] AND obeys the following rules:

true AND ***true***	results in ***true***
true AND ***false***	results in ***false***
false AND ***true***	results in ***false***
false AND ***false***	results in ***false***

In other words, "A AND B" is ***true*** if both A and B are ***true;*** otherwise it is ***false.***

OR is another logical operator. It obeys the following rules:

true OR ***true***	results in ***true***
true OR ***false***	results in ***true***
false OR ***true***	results in ***true***
false OR ***false***	results in ***false***

In other words, "A OR B" is ***false*** if both A and B are ***false;*** otherwise it is ***true.***

The third logical operator in Basic is NOT. NOT operates on ***one*** value:

NOT ***true***	results in ***false***
NOT ***false***	results in ***true***

The logical operators AND, OR and NOT have lower precedence than the relational operators ('<', '=', etc). Among the logical logical operators,

NOT has the highest priority,

AND has the next priority, and

OR has the lowest priority.

[2] - Logical operators actually operate on any integer values in the range -32,767 to +32,767. If one of the values is not 0 or -1, the result may not be 0 or -1. A complete description of these operators is beyond the scope of this book. If you use logical operators only on the results of relational operators, as in the examples above, the matter will not come up.

Examples

Suppose A = 5, B = 4, and C = 3. Then:

```
IF A=9 AND B=4 OR C=3 GOTO 45          reduces to
IF false AND true OR true GOTO 45      reduces to
IF false OR true GOTO 45               reduces to
IF true GOTO 45
```

The statement goes to line 45.

```
IF A=9 AND (B=4 OR C=3) GOTO 45        reduces to
IF false AND (true OR true) GOTO 45    reduces to
IF false AND ( true ) GOTO 45          reduces to
IF false GOTO 45
```

The statement does not go to line 45.

```
IF NOT A=9 OR B=4 GOTO 45              reduces to
IF NOT false OR true GOTO 45           reduces to
IF true OR true GOTO 45                reduces to
IF true GOTO 45
```

The statement goes to line 45.

```
IF NOT (A=9 OR B=4 ) GOTO 45           reduces to
IF NOT (false OR true ) GOTO 45        reduces to
IF NOT true GOTO 45                    reduces to
IF false GOTO 45
```

The statement does not go to line 45.

SEVERAL STATEMENTS ON A LINE

Up to now you have seen every Basic statement on its own line in your program file. You can store two or more statements on one line by putting a ':' between statements. For example,

```
110 SU=SU+(ER-SR)/GA
120 CO=CO+1
122 SR=ER
```

may be written:

```
110 SU=SU+(ER-SR)/GA:CO=CO+1:SR=ER
```

If one statement on a line is an IF. . .THEN, the outcome of the IF. . .THEN will determine whether ***all*** of the remaining statements are executed, or ***none*** of them are.

Consider the following line:

```
50 IF NC>0 THEN AC=AC+1:BC=BC+1:▶
   CC=CC+1
```

This line is equivalent to:

```
50 IF NC<=0 GOTO 90
60 AC=AC+1
70 BC=BC+1
80 CC=CC+1
90 . . .
```

If 'NC>0' is true, then all three of the following assignment statements are executed. If 'NC>0' is ***false,*** none of the assignment statements are executed.

':' makes this piece of code more compact and more readable than the equivalent statements with GOTO.

':' has some other advantages:

- It saves space in the program file.
- If several statements are logically very closely related, putting them on the same line emphasizes their relatedness.

Caution: use ':' sparingly. If you overwork it, your program will end up with a lot of long, confusing lines; the statements will run together, and the program will be very ***un***readable.

In any case, do not try to make a program line more than 80 characters long. Basic will not accept a longer line as input, and if you create one by inserting characters in a shorter line, Basic will chop off the end!

THE ON/GOTO STATEMENT: MULTI-WAY DECISIONS

Basic has another decision making statement, the ON/GOTO statement, that is useful when you want to execute one of several GOTO's, and you can base your choice on a variable whose value is 1, 2, 3, etc.

The ON/GOTO statement looks like this:

```
20 ON XY GOTO 110,120,130
```

'XY' is the name of a numeric variable. The numbers after 'GOTO' are line numbers.

In this example, if the value of XY is 1, the ON/GOTO statement passes control to line 110, the first line number. If the value of XY is 2, ON/GOTO passes control to 120, and so on.

If the value of XY is not an integer, it is truncated to the next lower integer.

If the truncated value of XY is less than 1 or greater than the number of line numbers in the list, ON/GOTO does nothing; control passes to the next statement.

CHAPTER 10: ARRAYS

WHAT IS AN ARRAY?

Up to now we have dealt with variables that have a single value. Basic also knows about a kind of "variable" that has more than one value. Such a "variable" is called an **array**.

An array consists of a group of **elements**, each of which can hold a value, just as a variable can. All the elements of an array have the same name -- the array's name -- but each element has a unique **subscript**. A subscript is a number identifying an element, like this:

```
A(5)=17.3
```

This statement assigns the value 17.3 to element #5 of an array named A. Similarly,

```
B=B*A(5)
```

multiplies B by element #5 of array A, and assigns the product to B. The subscript used with A in this statement is 5.

You can use an array element in any place that you can use a variable.

Basic distinguishes an array from a variable simply by the fact that an array has a subscript and a variable does not. For example, if you used the statement

```
B=B*A
```

and

```
B=B*A(5)
```

in the same program, the first statement would multiply B by a variable named A, and the second statement would multiply B by the 5th element of an array named A. Basic never confuses arrays with variables. (But since ***you*** may confuse them, we recommend that you never use the same name for an array and a variable in the same program.)

THE DIMENSION OF AN ARRAY

The number of elements in an array is called the **dimension** of the array. For example, if an array has 53 elements, its dimension is 53.

If your program simply starts referring to an array, Basic {**B**} assumes that the array's dimension is 10. The elements in the array have subscripts from 0 to 9. (The first element in a Basic array always is element #0).

To create an array with a dimension other than 10, execute a **DIM** ("dimension") statement. Here is an example of a DIM statement:

```
DIM A(53)
```

This statement creates an array named A, with a dimension of 53. The elements are subscripted from #0 to #52.

You can also use the DIM statement to create an array with ***fewer*** than 10 elements. This will make your program require less memory when it runs.

USES OF ARRAYS

Arrays are useful for processing any sort of data that comes in groups. Here are some examples of situations where arrays may come in handy:

- You are doing calculations on the sales records of a store. You have several sets of statistics for consecutive months of business, and you want to compare each month to the one before it. One way to approach the task is to create an array for each statistic, with an element for each month.
- You have a set of numbers that must be put into ascending order. You can put the numbers in an array, and then move them from one element to another until they are arranged the way you want them.
- You want to write a program that asks its user for a number, and then checks the number against a list of acceptable responses. If you keep the acceptable responses in an array, you can write a loop to search through the array elements until you find a match or until you reach the end of the array.

AN EXAMPLE: CALCULATING THE NUMBER OF A DAY IN A YEAR

As an example of how to use arrays, here's a program to compute the number of a day in a year. If you gave this program the input '2 1' (for "February 1"), it would display '32' ("that is the 32nd day of the year").

```
110 REM Convert a date to day-of-year.
120 REM In: month, day. A zero month▶
    means "end run.
130 REM Out: day of year.
140 REM Vars: MN=month of year. DM=day▶
    of month.
150 REM DR=day of year, the result.
160 REM M1=days Preceding 1st day of▶
    MN.
170 REM
180 REM MA=array of days in year▶
    before each month.
190 DIM MA(12)
200 MA(0)=0:MA(1)=31:MA(2)=59:MA(3)=90
210 MA(4)=120:MA(5)=151:MA(6)=181:▶
    MA(7)=212
220 MA(8)=243:MA(9)=273:MA(10)=304:▶
    MA(11)=334
230 REM Prompt user for month, day.
240 INPUT "Month, day";MN,DM
250 IF MN=0 THEN END
260 REM Calc M1, then DR.
270 M1=MA(MN-1)
280 DR=M1+DM
290 PRINT "Day of year=";DR
300 GOTO 240
```

Let's study what this program does, step by step.

Lines 110 through 180 are remarks that explain the purpose and use of the program, and the meaning of each variable in it.

Lines 180 through 220 create and initialize a 12-element array, MA. MA(0) is assigned the number of days in the year that precede January; MA(1) is assigned the number of days that precede February; and so forth.

Line 230 marks the beginning of the main processing part of the program. Line 240 prompts the user for the input: month-of-year (a number from 1 to 12) and day-of-month (a number from 1 to 31). Line 250 tests for a month number of 0, which is the value the user should enter to end the run.

Lines 260 through 280 calculate day-of-year. Day-of-year is the number of days in the year that precede this month (MA(MN-1)) plus the day-of-month (DM). Line 290 displays the day-of-year; line 300 loops back to the INPUT statement for another calculation.

Notice that the program produces an incorrect result for months 3 through 12 in a leap year. We'll see how to correct that in a later chapter.

ANOTHER EXAMPLE: RECORDING VALUES IN ORDER

Let's look at another simple program that uses arrays. This one asks the user for a set of ten values, then displays the

values in ascending order.

```
10 REM Read 10 values & Print them in▶
   order.
20 REM Values are stored in ascending▶
   order of value.
30 REM N=each value read: NN=number▶
   of this value (0-9);
40 REM RA is an array to hold values▶
   in order;
50 REM NP is a Pointer used to insert▶
   a new value in RA.
60 DIM RA(10)
170 REM Insert N after highest NP▶
    where N>RA(NP), or at NP=0.
180 IF NP<0 GOTO 210
190 IF RA(NP)<N GOTO 210
200 RA(NP+1)=RA(NP):NP=NP-1:GOTO 180
210 RA(NP+1)=N:NN=NN+1:IF NN<10 GOTO▶
    150
220 REM
230 REM Print the values.
240 NN=0:PRINT "The values are:";
250 PRINT RA(NN);
260 NN=NN+1:IF NN<10 GOTO 250
270 PRINT
```

Lines 10 through 50 are remarks that describe the program and the variables in it.

Line 60 creates the array. Since the array's dimension is 10 there is no need to create it explicitly, but it makes the operation of the program clearer to a reader.

Lines 120 through 210 read ten values and store them in the array. Each loop reads one value and inserts it in the array order. The procedure for inserting a value in the array is described in the next few paragraphs.

NN is the number of the value being inserted: 0 for the first value, 1 for the second value ... 9 for the tenth and last. We assign NP the value NN-1; thus it points to the subscript of the last element in RA that already contains a value.

Now we examine each value in the array, from element NP down to element 0, to see if it is less than N.

If the element is ***not*** less than N, N goes somewhere before this element; we copy the value of this element up to the next element and subtract 1 from NP in line 200, then loop back to examine the preceding element.

If the element ***is*** less than N, N goes in the element immediately ***after*** this one; we assign the value of N to that

element. (That element is "free," since we moved its value up one element in the preceding loop. If this is the first loop, that element is free because it hasn't been used yet.)

If N is less than every element stored in the array so far, we continue looping until NP = -1; but a special test in line 180 takes care of that case by going to line 210, which stores the value of N in RA(0). (This test also takes care of the "insertion" of the first number we read, when RA is empty and NN = 0.)

Once we have collected ten numbers in order, line 210 allows control to drop down to line 240. The loop in lines 240 through 260 displays the result in this format:

```
The values are: -8 -3 1 4 9 13 13 20▶
  99 223
```

The empty PRINT statement on line 270 ends the line of output that the loop built up. (Since the PRINT statements in lines 240 and 250 end with ';'s, the line of output would never be ended otherwise.)

MULTI-DIMENSONAL ARRAYS {B}

Basic lets you create arrays that have more than one dimension, that is, more than one subscript. For example, you can create a two-dimensional array like this:

```
DIM R2(5,6)
```

You can think of this array as a checkerboard, with each square representing an element.

We say that R2 has 5 **rows**, with subscripts from 0 to 4, and 6 **columns**, with subscripts from 0 to 5. When we display such an array, or draw a diagram of it, we customarily represent it like this:

R2(5,6): a two-dimensional array

	COLUMNS 0	1	2	3	4	5
0	RA(0,0)	RA(0,1)	RA(0,2)	RA(0,3)	RA(0,4)	RA(0,5)
1	RA(1,0)	RA(1,1)	RA(1,2)	RA(1,3)	RA(1,4)	RA(1,5)
ROWS 2	RA(2,0)	RA(2,1)	RA(2,2)	RA(2,3)	RA(2,4)	RA(2,5)
3	RA(3,0)	RA(3,1)	RA(3,2)	RA(3,3)	RA(3,4)	RA(3,5)
4	RA(4,0)	RA(4,1)	RA(4,2)	RA(4,3)	RA(4,4)	RA(4,5)

Two-dimensional arrays are useful for many kinds of programs where it is natural to represent data as a two-dimensional grid of values. For example, if you were writing a chess playing program, you might well use a two-dimensional array to represent the chess board, and store a value in each element to indicate what piece, if any, was sitting on the square that element represented.

You can define arrays with more than two dimensions, as well. Such arrays could use large amounts of memory when they are run, though. They could easily exceed the memory capacity of the HHC. For example, if you defined a five-dimensional array with six elements per dimension, the array would contain 7,776 elements, and this is considerably more than the HHC can manage, even with the largest-capacity Programmable Memory Peripheral.

INTEGER VARIABLES

So far you have encountered only one kind of Basic variable: the **numeric variable**, which holds a numeric (or real, or floating point) value like 519 and 3.141592.

Another kind of Basic variable is the **integer variable**. An integer variable can hold only integers, or whole numbers, such as 519, 3, 2, 1, 0, and -24.

You create an integer variable by putting a '%' sign after a variable name. For example, HE% is the name of an integer variable.

HEAP%, HERKIMER%, and HEFFILTRUP% are all the same as HE% as far as Basic is concerned. The variable naming restrictions that we described for numeric variables apply to integer variables, too, but the '%' doesn't count as one of the two significant characters.

You can operate on integer variables just the same way you do on real variables. For example, all of the following statements are valid:

```
X%=5
X%=(15-3)*(15+3)/5
X%=(X%+1)^2
PRINT X%
PRINT X%*(X%-1)
```

You can define integer arrays:

```
DIM MA%(12)
.
.
.
M1=MA%(MN-1)
```

You can mix integer and numeric values freely in a statement:

```
X%=HE
HE=X%
X%=3*HE
HE=2.5*X%
HE=(X%-3)*(X%+3)/HE
```

When Basic evaluates an expression containing integers, it converts all the integers' values to numeric form before it starts. When Basic assigns a value to an integer variable, it converts the value from numeric to integer form when it is done. Thus, the statement {B}

```
X%=X%+Y%
```

requires three conversions: two from integer form to numeric form, and one back from numeric form back to integer form. This makes the statement execute considerably slower than 'X = X + Y' would.{1}

An integer variable has an initial value of zero, just like a numeric variable.

Basic keeps similar-named integer and numeric variables distinct, just as it keeps similar-named variables and arrays distinct. For example, variables with names like HE and HE% are not related in any way.

But again, ***you*** are likely to confuse such variables, even though Basic doesn't; so we recommend that you avoid using them.

Characteristics of Integer Variables

The largest value that an integer variable may have is 32767.{2} The smallest value that an integer variable may have is -32768. (A negative number is considered "smaller" than zero; -2 is "smaller" than -1.)

Integer variables require less memory when your program is run than numeric variables do. This is important when you are dealing with arrays. If you use integer arrays instead of numeric arrays, you can use more or larger arrays in your program before you run out of space to run it.

Memory in your computer is measured in units called "bytes." A **byte** is the amount of memory needed to store one character of data. A numeric variable requires 7 bytes of storage; an integer variable requires only 4 bytes.

{1} - Experienced programmers, note: for the same reason, you cannot round down one value to an even multiple of another value by dividing and multiplying, like this:

```
Z%=5*(Z%/5)
```

To perform this sort of calculation, use the INT function:

```
Z%=5*int(Z%/5)
```

{2} - There are good technical reasons for this odd number, but they are beyond the scope of this book.

CHAPTER 11: SOME EXAMPLES

WHERE DO THE EXAMPLES COME FROM?

Looking at the programs in the preceding chapters, you may have gotten the feeling that they were created by magic. "How could I ever have done that?" you may have wondered.

You ***can*** develop programs like the ones you have seen, and more. You don't have to be a computer genius to do it. All you need is patience and practice. An intelligent, systematic plan of attack will make your work go much easier.

To give you a feel for how a program is developed, we're going to show the steps involved in writing the two example programs that we looked at in the chapter on arrays. In the process, we'll extract some general principles that will make any kind of Basic program easier for you to write.

THE DAY-OF-YEAR CALCULATOR

Before we try to write this program, we'll develop a general plan for it. It will be much easier to reach our goal once we know clearly what the goal is!

Here's our first try at a plan:

A. Prompt the user for the values of month-of-year and day-of-month. Store them in variables named MN and DM.

B. Convert the date (MN and DM) to a day of year. Store that in a variable named DR.

C. Print DR.

This may not look very impressive, but it is a start. We've reduced the size of the problem; the only part that needs further attention is step B. We've also identified some variables we will need, and given them names. As we work on the program we can refer back to this plan to help keep us on the right track; we won't have to distract ourselves trying to remember whether or how we defined a certain variable, what we named it, and so on.

Now let's refine step B further.

B. Convert the date (MN and DM) to a day of year. Store that in a variable named DR.

1. Convert MN to day-of-year for the first day of that month. (Call the result M1.)
2. DR = M1-1 + DM.

The only part of the plan that needs further refinement is step B(1). Here's where the array comes in.

There are several ways we could use an array in this step. We're going to choose one of them arbitrarily. (You may want to think of another one, and write a program for it, as an exercise.)

B (1). Convert MN to day-of-year for the first day of that month. (Call the result M1.)

We'll define an array with 12 elements, each containing the day-of-year for the first day of a month. Element 0 represents January, element 1 represents February, and so forth. We'll call this array MA.

a. Use MN as a subscript into the array, to assign to M1 the day-of-month for the first day of MN. In other words, M1 = MA(MN).

Now, if we assemble all these pieces into a single outline, our plan is complete:

A. Prompt the user for the numbers: month of year, day of month, and year. Store them in variables named MN and DM.

B. Convert the date (MN and DM) to a day of year. Store that in a variable named DR.

1. Convert MN to day-of-year for the first day of that month. (Call the result M1.)

   We'll define an array with 12 elements, each containing the day-of-year for the first day of a month. Element 0 represents January, element 1 represents February, and so forth. We'll call this array MA.

   a. Use MN as a subscript into the array, to assign to M1 the day-of-month for the first day of MN. In other words, M1 = MA(MN-1).

2. DR = M1-1 + DM.

C. Print DR.

Having done all this groundwork, we have a very detailed plan of the program we're going to write. By developing the program's logic first, and then writing the program itself, we can devote our attention to each aspect of the program without being distracted by the other. This allows us to develop the program with less likelihood of making an error, and usually with less overall effort.

When we first write the program from the outline above, we might come up with something like this:

```
110 REM Convert a date to day-of-year.
120 REM In: month, day.
130 REM Out: day of year.
140 REM Vars: MN=month of year.▶
    DM=day of month.
150 REM DR=day of year, the result.
160 REM M1=day-of-year for 1st day of▶
    MN.
170 REM
180 REM MA= array of days in year▶
    before each month.
190 DIM MA(12)
200 MA(0)=1:MA(1)=32:MA(2)=60:MA(3)=91
210 MA(4)=121:MA(5)=152:MA(6)=182:▶
    MA(7)=213
220 MA(8)=244:MA(9)=274:MA(10)=305:▶
    MA(11)=335
230 REM PromPt user for month, day.
240 INPUT "Month, day:";MN,DM
250 REM Calc M1, then DR.
260 M1=MA(MN-1)
270 DR=M1-1+DM
280 PRINT "Day of year=";DR
```

Looking over this program, we notice some things about it that could be improved.

1. In line 270, we could avoid subtracting 1 from M1 by reducing the value of each element in MA by 1. Then each element MN-1 of MA would contain "number of days before the first day of month MN" instead of "day-of-year for the first day in MN." This would shorten our program slightly.
2. We could combine the calculations of M1 and DR into a single step. In fact, we could combine them both with the PRINT statement and eliminate the need to have these variables at all.
3. We could make the program more useful by looping back from the end to statement 240, giving the user the option of doing two or more calculations.

These are things we might not have foreseen before writing the program. Now, having seen them, we might decide to revise the program in the light of what we have seen -- or we might decide to leave well enough alone. After all, the program works. "If it ain't broke, don't fix it!"

In this case, we decide to revise the program to incorporate changes #1 and #3. We forgo change #2; it would make the program a little shorter, but would also make it harder to read. The separate steps clarify what is happening when we calculate the day-of-year.

We end up with the version you saw in the chapter on arrays (changes from the first version are in boldface):

```
110 REM Convert a date to day-of-year.
```

```
120 REM In: month, day. A zero month▶
    means "end run."
130 REM Out: day of year.
140 REM Vars: MN=month of year. DM=day▶
    of month.
150 REM DR=day of year, the result.
160 REM M1=days Preceding 1st day of▶
    MN.
170 REM
180 REM MA= array of days in year▶
    before each month.
190 DIM MA(12)
200 MA(0)=0:MA(1)=31:MA(2)=59:MA(3)▶
    =90
210 MA(4)=120:MA(5)=151:MA(6)=181:▶
    MA(7)=212
220 MA(8)=243:MA(9)=273:MA(10)=304:▶
    MA(11)=334
230 REM Prompt user for month, day.
240 INPUT "Month, day:";MN,DM
250 IF MN=0 THEN END
260 REM Calc M1, then DR.
270 M1=MA(MN-1)
280 DR=M1+DM
290 PRINT "Day of year=";DR
300 GOTO 240
```

THE VALUE-ORDERING PROGRAM

We'll develop the value-ordering program the same way we developed the day-of-year program: by writing down an overall plan for program development, and progressively refining the plan until we have an outline so detailed that writing the program is easy.

Here's a first try at a plan for this program:

A. Get 10 values in order.

   1. Get a value.
   2. Store it in order.
   3. Repeat until 10 values are stored.

B. Print the values.

We're going to need an array with 10 elements to store the values. Let's call that array RA.

We'll also need a variable to read a value and hold it until we decide where in the array to store it. We'll call that variable N.

Finally, we'll need a variable to use as a counter to tell us when we've read 10 values. We'll call it NN.

Our next version of the plan will incorporate these variable

names, and will add some further refinement to step A(2):

A. Get 10 values in order.

   1. Get a value in N.
   2. Store N in order in RA.

   When we insert N as the NN'th value, RA already holds NN-1 values, in order, in RA(0) through RA(NN-2).

   We look for the element where we should insert N, working down from RA(NN-2) to RA(0). Let's call that element NP. When we find NP, we move the values in RA(NP) through RA(NN-1) up to RA(NP+1) through RA(NN), then store N into RA(NP).

   3. Repeat until 10 values have been inserted.

   a. NN=NN+1 (initially it is 0).

   b. If NN< 10, repeat from step 1.

B. Print the values in RA.

Now we have to refine the English description in step A(2) into a more program-like description that can easily be converted into a program.

As we contemplate the problem, it occurs to us that looking for the insertion point NP and moving the following values up are both element-by-element processes that proceed from RA(NN) downward. We might as well save ourselves a loop by doing both at once.

The next stage in refining step A(2) is this:

A (2). Store N in order in RA.

   a. NP=NN-1, subscript of the last element of RA used so far.

   b. Is RA(NP)<N? If so, N goes in RA(NP+1). Will it destroy another value already stored there? No; for RA(NP) is the last element of RA that has been used so far.

   c. If RA(NP)<N is ***not*** true, N will be inserted in the array somewhere before the value now in RA(NP). Copy the value in RA(NP) up to RA(NP+1); subtract 1 from NP and repeat step b. (This won't destroy another value already at RA(NP+1), for the same reason as in step b.)

   Note: on a second or later loop, will moving a value into RA(NP+1) destroy the value already there? No; for RA(NP+1) was copied to RA(NP+2) the previous time through the loop.

By working through a couple of examples with real values, we can confirm that this procedure does what we want. Before we go on, though, we must consider whether the

procedure will break down when it encounters its boundary conditions.

A **boundary condition** is a condition that is exceptional in some way when compared to all the conditions the program could possibly be in. For example, if a program were written to calculate the sum of all the integers from 1 to ***n,*** its boundary conditions would be ***n*** = 1 (for which the answer should be 1) and ***n*** < 1 (for which the answer should be something meaning "that doesn't make sense"). Boundary conditions very often cause otherwise problem-free programs to act incorrectly.

We can think of four boundary conditions for this program:

1. Inserting the first value in the array: NN will be 0, so NP will be -1. Right off we'll be comparing N to RA(-1), a non-existent element. This will cause an error, and we'd better allow for it.
2. Inserting the last value in the array: NN will be 9, so NP will be 8. We'll be moving some number of values up from RA(8) to RA(9), RA(7) to RA(8), ... This should work fine.
3. Inserting a value higher than any yet in the array: we'll simply find that RA(NP) < N on the first loop, so we'll store N into RA(NN). This creates no problem.
4. Inserting a value lower than any yet in the array: the loop will continue until NP is reduced to 0, then to -1, and we'll encounter a problem similar to that in boundary condition #1.

We can take care of condition #4 by inserting a new step between A(2)(a) and A(2)(b):

A(2). Store N in order in RA.

a. NP = NN-1, subscript of the last element of RA used so far.

b. If NP = -1, N is lower than any value yet in RA; store N in RA(0). (RA(0)...RA(NP) have already been moved out of the way.)

c. Is RA(NP) < N? If so, N goes in RA(NP + 1). Will it destroy another value already stored there? No; for RA(NP) is the last element of RA that has been used so far.

d. If RA(NP) < N is ***not*** true, N will be inserted in the array somewhere before the value now in RA(NP). Copy the value in RA(NP) up to RA(NP + 1); subtract 1 from NP and repeat from step b. (This won't destroy another value already at RA(NP + 1), for the same reason as in step b.)

   Note: on a second or later loop, will moving a value into RA(NP + 1) destroy the value already there? No; for RA(NP + 1) was copied to RA(NP + 2) on the preceding loop!

What about condition #1? Our first impulse might be to add a step before step A to get the first value and store it in RA(0). Then step A would get only the second through tenth values, and this boundary condition would never come up.

Before we do anything drastic, however, let's take a second look at condition #4. Condition #1 looks suspiciously like a special case of condition #4 -- could it be that the check we added for condition #4 will also take care of condition #1, or could easily be made to do so? We work through the plan for condition #4, and discover that our check will, indeed, handle condition #1 without change.

Here's a complete outline for our value-ordering program:

A. Get 10 values in order.

   RA is an array which accumulates the 10 values, in order.

   N holds a value between the time it is input and the time it is inserted in RA.

   NN counts the values as we insert them. RA(NN-1) is the last element to contain a value so far.

   NP is a pointer used to find the place in RA where N will be inserted; it decreases from NN-1 toward 0 on each loop.

1. Get a value in N.
2. Store N in order in RA.

When we insert N as the NN'th value, RA already holds NN-1 values, in order, in RA(0) through RA(NN-2).

We look for the point to insert N, working down from RA(NN-2) to RA(0). Let's call the subscript of that point NP. When we find NP, we move the values in RA(NP) through RA(NN-1) up to RA(NP + 1) through RA(NN), then store N into RA(NP).

a. NP = NN-1, subscript of the last element of RA used so far.

b. If NP = -1, N is lower than any value yet in RA; store N in RA(0). (RA(0)...RA(NP) have already been moved out of the way.) Note, this test also takes care of the case where NN = 0.

c. Is RA(NP) < N? If so, N goes in RA(NP + 1). Will it destroy another value already stored there? No; for RA(NP) is the last element of RA that has been used so far.

d. If RA(NP) < N is ***not*** true, N will be inserted in the array somewhere before the value now in RA(NP). Copy the value in RA(NP) up to RA(NP + 1); subtract 1 from NP and repeat from step b. (This won't destroy another value already at RA(NP + 1), for the same reason as in step b.)

Note: on a second or later loop, will moving a value into RA(NP+1) destroy the value already there? No; for RA(NP+1) was copied to RA(NP+2) on the preceding loop!

3. Repeat until 10 values have been inserted.

   a. NN=NN+1 (initially it is 0)

   b. If NN< 10, repeat from step 1.

B. Print the values in RA.

Now we're ready to write the program you saw in the chapter on arrays:

```
10 REM Read 10 values & Print them in▶
   order.
20 REM Values are stored in ascending▶
   order by value.
30 REM N=each value read; NN=number of▶
   this value (0-9);
40 REM RA is an array to hold values▶
   in order;
50 REM NP is a pointer used to insert▶
   a new value in RA.
60 DIM RA(10)
120 NN=0
130 REM
140 REM Loop for each number.
150 INPUT "Give a value";N
160 NP=NN-1
170 REM Insert N after highest NP▶
    where N>RA(NP), or at NP=0.
180 IF NP<0 THEN GOTO 210
190 IF RA(NP)<N GOTO 210
200 RA(NP+1)=RA(NP):NP=NP-1:GOTO 180
210 RA(NP+1)=N:NN=NN+1:IF NN<10 GOTO▶
    150
220 REM
230 REM Print the values.
240 NN=0:PRINT "The values are:";
250 PRINT RA(NN);
260 NN=NN+1:IF NN<10 GOTO 250
270 PRINT
```

SOME NOTES ON DEBUGGING

When you first test a program, don't be astonished if it doesn't work. Most new programs contain at least a few **bugs**. Plan to spend some time **debugging** each program you write before you can use it.

A ***thorough*** treatment of debugging is beyond the scope of this book. Experience will be your best teacher. We're just going to give you a few tips to get you started.

Avoiding Bugs

The best way to debug a program is to write a bug-free program in the first place. For most of us this is an unattainable goal, but we can program more efficiently if we work in a way that minimizes the number of bugs we produce.

The most important key to writing bug-free code is to design your program ***before*** you start to write it. This can't be emphasized too much!

If you were building a house, you wouldn't lay the foundation before deciding where to put the rooms; yet many programmers do the equivalent by writing code before developing a detailed plan for a program. For all but the tiniest programs, you need a plan to avoid a tremendous waste of effort, or even a complete failure. Don't try to work without one!

Avoid writing unnecessary or unwarranted assumptions into your code. For example, if you need a loop that will end when A> =B, don't write 'IF A=B..'. just because A ***ought*** to equal B when you want the loop to stop. Protect yourself from a runaway loop by assuming that something will throw A off a bit. If the program's logic permits it, write 'IF A>=B' even though you don't you don't think it's really necessary.

When you have a choice between two ways of writing a program, favor the simpler one. Simple designs are less likely to contain bugs in the first place, and when they do, they are easier to fix.

Keep your program's design and code "clean." Often there are ways to write a program with a little less code, or make it run in a little less time, that take advantage of some quirk in the way Basic works. Resist the temptation to write your program that way unless it will gain you some benefit that is really important. Such code is hard to debug, and once debugged, it is hard to modify. It usually costs you more than it is worth.

Eliminating Bugs

When you are faced with a bug, what should you do? First, don't panic! Remember that bugs are logical errors, and can be corrected by the use of logic. Your program is not the victim of black magic; it is simply doing what you wrote it to do, rather than what you wanted it to do.

Always debug with your thinking cap on. When your program behaves in totally outrageous, incomprehensible ways, in-

quire what the cause of its behavior could be. Watch for all evidence that might bear on that question; the most obvious symptom may not be the most significant clue.

It helps to work like a scientist. Form a theory about what your program is doing, then do an experiment that will prove or disprove your theory. If your program's behavior just doesn't make sense, don't guess in the dark; ask yourself what information you need then do an experiment that will get it for you.

Above all, don't make rash assumptions about what is wrong with your program. Suspect ***every*** statement of harboring a bug. There's nothing more frustrating than looking for a bug in one place when it's really someplace else.

Execution Tracing Aids

If your program goes into an endless loop, remember that the C1 key will stop it. Note what statement your program stops in; that is one clue to what went wrong.

Inspecting the values of your program's variables will also help you figure out what the program was doing when it went astray.

Basic's **trace facility** is a valuable debugging tool. The trace facility consists of two statements, TRON ("trace on") and TROFF ("trace off").

The **TRON** statement looks like this:

```
TRON
```

It makes Basic start displaying a trace of deferred-mode execution. The trace shows the line number of each line of your program when that line is executed. For example, a portion of a trace showing the execution of lines 60, 70, and 80 would look like this:

```
[60][70][80]
```

The **TROFF** statement looks like this:

```
TROFF
```

It makes Basic stop displaying the trace of started by TRON.

TRON and TROFF may be used in immediate or deferred mode. Once used, TRON is effective until the next BYE.

The PRINT statement is a useful debugging aid. By inserting PRINTs in your program at appropriate points, you can display a **trace** that tells you what parts of your program are

executing and how the values of important variables are changing.

If you have trouble keeping track of what your program is doing as you debug it, consider investing in a printer. A printer would enable you to create a permanent record of any number of lines of program output, and so get a better view of what your program is doing. (We will discuss the use of peripherals such as printers in a later chapter.)

The CONT Command

You can restart a program after halting it with the C1 key by entering the **CONT** (for "continue") command. CONT restarts a program at the next statement after the one where execution stopped, just as though it had never stopped at all. Unlike RUN, CONT does ***not*** reset the values of variables.

You can use CONT after modifying the values of program variables, but you cannot use CONT after modifying statements in your program.

You can use CONT after your program executes a STOP or END statement, as well as after you press C1. (**STOP** is like END, except that it prints line number of the statement your program STOPs in.) {**B**}

Finding All the Bugs

When you are debugging a program, remember to test whatever boundary conditions you can think of. These are the conditions that most frequently make a program fail, particularly after you think it's fully debugged.

Test your programs thoroughly. Don't assume a program is "working" as soon as you've made it run once. If you give it a different set of input it will probably fail, and you'll have a chance to eliminate another bug. Expect your program's behavior to improve gradually, until it reaches an apparently bug-free state.

Test your programs systematically. Make a list of every condition that might conceivably reveal a bug, and test each one. When you fix a bug, retest any condition that is processed by code that the fix might have disturbed.

Don't test a large program all at once. Use the "divide and conquer" approach to debugging; divide your program into logically separate units, and debug each unit. For example, if one part of your program reads input and checks it for validity, test that part of your program to make sure it works before trying to test other parts that depend on it.

In this sort of testing, it is useful to use the RUN command like this:

```
RUN 2040
```

where the number after RUN is a line number where you want execution to begin. (RUN with no line number starts execution at the first statement in your program.)

CONCLUSION

Do you still feel overwhelmed by these programs? You needn't be. Start by writing simple programs that you can grasp easily. As you gain experience with Basic, and as you think of more sophisticated programs that you would like to write, you will be able to tackle more and more ambitious projects.

Don't be upset if you don't understand the reasons for some of the choices we made while designing the programs in the examples above. There often are many ways a piece of code can be written. You can develop a sense of judgment about which way is best by trial and error, and by studying programs written by more experienced programmers.

Don't worry about finding the ***best*** way to write a given program. Be content to find a ***good*** way, and go forward with the work of designing and writing a program that works well. After you have used a program for a while, you will start to think of all sorts of better ways to make it work. Things that seem obscure one day will seem obvious the next -- and they'll send you off on a new round of improvements. That's part of the excitement of programming.

CHAPTER 12: THE FOR/NEXT STATEMENTS

Here is a kind of loop that appears quite often in Basic programs:

```
100 I=0
  .
  .
180 I=I+1
190 IF I<10 GOTO 110
```

Basic has a pair of statements, the FOR and NEXT statements, that make this kind of loop easier to write. Using FOR and NEXT, we could write the loop in the example above like this:

```
100 FOR I=0 TO 9
  .
  .
180 NEXT I
```

Basic assigns I the value 0, and executes the statements between FOR and NEXT. Then it assigns I the value 1, and executes the statements between FOR and NEXT again. then it assigns I the value 2, and so forth.

Basic executes the statements between FOR and NEXT for the last time after assigning I the value 9. Then it moves on to the statement after NEXT.

This sort of loop is called, logically enough, a **FOR/NEXT loop.**

A FOR/NEXT loop has several advantages over an equivalent loop written with IF's and GOTO's:

1. It is shorter an easier to write.
2. There is less chance that you will make an error writing it.
3. It makes the beginning and end of the loop more visible when you read the program.

SOME TERMINOLOGY AND RULES

In a FOR/NEXT loop like this,

```
FOR I=0 TO 10
  .
  .
NEXT I
```

'I' is called the **index.** It may be any numeric (non-integer) variable.

'0,' the first value assigned to the index, is called the **initial value.** It may be a constant, variable, or expression.

'10,' the value where looping stops, is called the **limit**. It also may be a constant, variable, or expression.

Every FOR/NEXT loop should be ended by one and only one NEXT.

To finish executing a loop before you reach the NEXT statement, GOTO the NEXT statement -- not back to the FOR.

You may safely use the value of a loop's index after you leave the loop.{1}

You may terminate a FOR/NEXT loop before it has finished its last time through the loop, simply by doing a GOTO to some statement that is outside the loop. But never try to start a FOR/NEXT loop by doing a GOTO from outside the loop to inside the loop! If you try this, you will get the message

```
NF error
```

meaning, "**N**EXT without **F**OR."

Don't try to assign a value to the index of a loop while inside the loop.{2}

A FOR/NEXT loop always loops at least once, even if the index is past the limit the first time through the loop. Therefore, don't use FOR/NEXT where you might want a loop to be executed zero times unless you include a test for the zero-times case before the FOR/NEXT. You can write a loop with IF's and GOTO's instead.

AN EXAMPLE

Our program for listing values in ascending order is a good example of how useful FOR/NEXT loops can be. There are two places in the program where they can be used:

```
10 REM Read 10 values & Print them in▶
   order.
20 REM Values are stored in ascending▶
   order by value.
30 REM N=each value read; NN=number of▶
   this value (0-9);
40 REM RA is an array to hold values▶
   in order;
50 REM NP is a Pointer used to insert▶
   a new value in RA.
60 DIM RA(10)
130 REM
140 REM Loop for each number.
145 FOR NN=0 TO 9
150 INPUT "Give a value";N
160 NP=NN-1
170 REM Insert N after highest NP▶
    where N>RA(NP), or at NP=0.
180 IF NP<0 GOTO 210
190 IF RA(NP)<N GOTO 210
200 RA(NP+1)=RA(NP):NP=NP-1:GOTO 180
210 RA(NP+1)=N
212 NEXT NN
220 REM
230 REM Print the values.
240 PRINT "The values are:";
250 FOR NN=0 TO 9
260 PRINT RA(NN);
265 NEXT NN
270 PRINT
```

{1} - The value of the index will be the value it had the last time through the loop, ***plus*** the step that is added to the index each time through.

{2} - This works in Microsoft Basic, but not in some other versions of Basic. It is bad practice because it invalidates the information that you get about the index by looking at the FOR statement.

ABOUT THE INITIAL VALUE AND THE LIMIT

You can start a FOR/NEXT loop at any initial value you wish, and end it at any limit you wish. For example, the FOR statement

```
FOR J3=-12 TO 17
```

loops 30 times, assigning J3 the values -12, -11, -10, . . . 15, 16, 17.

THE STEP WORD

You can make a FOR/NEXT loop increment the index by any value you wish.

In the FOR statements we have looked at so far, the index is always incremented by 1: from 0 to 1 to 2 . . . or from -17 to - 16 to -15 . . . . To increment the index by a different value, use the **STEP** word like this:

```
FOR Q=1 TO 9 STEP 2
```

This FOR statement increments the index by 2. It loops 5

times, assigning Q the values 1, 3, 5, 7, and 9.

The value after the word STEP is called the **step** or the **increment**. It may be a constant, variable, or expression.

A FOR/NEXT loop's step may be negative. That gives you the ability to count backwards:

```
FOR Q=9 TO 1 STEP -2
```

This FOR statement loops 5 times, assigning Q the values 9, 7, 5, 3, and 1.

The initial value, the limit, and the step need not be whole numbers. For example:

```
FOR AF=.1 TO 2.1 STEP .5
```

This FOR statement loops 5 times, assigning AF the values .1, .6, 1.1, 1.6, and 2.1.

The index need never take on a value exactly equal to the limit. A FOR/NEXT loop stops looping when the index goes ***past*** the limit. For example:

```
FOR I=1 TO 10 STEP 2
```

This FOR statement loops for I = 1, 3, 5, 7, and 9. The next value, I = 11, would be past the limit; therefore FOR stops looping after I = 9.

Similarly,

```
FOR I=10 TO 1 STEP -2
```

This FOR statement loops for I = 10, 8, 6, 4, and 2. The next value, I = 0, would be past the limit, so FOR stops looping after I = 2.

If you "mix" forward and backward instructions in a FOR/NEXT loop, like this,

```
FOR I=1 TO 10 STEP -1
```

or like this,

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

Basic will execute the loop once, and then terminate it. This is because the increment is past the limit before the loop even starts, and in such a case Basic executes the loop one time.

NESTED FOR/NEXT LOOPS

One FOR/NEXT loop can be **nested** inside another. A nested loop looks like this:

```
100 FOR I=0 TO 9
110 FOR J=0 TO 9
120 XY(I,J)=0
130 IF I=9-J THEN XY(I,J)=1
140 NEXT J
150 NEXT I
```

In this example, the outer loop in lines 100 through 150 is executed 10 times, for I = 0, 1, 2, . . . 9. Each time the outer loop is executed, the inner loop in lines 110 through 140 is executed 10 times, for J = 0, 1, 2, . . . 9. Thus the statements inside the inner loop, lines 120 and 130, are executed 100 times, for all possible combinations of I = 0 to 9 and J = 0 to 9.

When you nest FOR/NEXT loops, one loop must always be contained completely within the other. That is, the following loop is valid,

```
300 FOR I=0 TO 10
310 FOR J=0 TO 6
320 XY(I,J)=0
330 NEXT J
340 NEXT I
```

but the following loop is invalid,

```
300 FOR I=0 TO 10
310 FOR J=0 TO 6
320 XY(I,J)=0
330 NEXT I
340 NEXT J
```

because the inner loop is not contained completely within the outer loop.

If you tried to run this piece of code, Basic would execute the two nested loops properly, since 'NEXT I' ends both the 'FOR I' loop and the 'FOR J' loop. Then Basic would try to execute line 340, and would display the message

```
NF error in 340
```

because the NEXT statement in line 340 does not match any FOR/NEXT loop currently being executed. ('NF' stands for "NEXT without FOR.")

THE VALUE ORDERING PROGRAM, REVISITED

Now that we know we can nest loops, and have backward-running loops, we can change another IF/THEN...GOTO loop in the value ordering program to a FOR/NEXT loop. Study the result to see how all the loops work. Once you understand FOR/NEXT loops, is this version of the program clearer than the one we saw in the chapter on arrays?

```
10 REM Read 10 values & Print them▶
   in order.
20 REM Values are stored in ascending▶
   order by value.
30 REM N=each value read: NN=number of▶
   this value (0-9):
40 REM RA is an array to hold values▶
   in order:
50 REM NP is a Pointer used to insert▶
   a new value in RA.
60 DIM RA(10)
130 REM
140 REM Loop for each number.
145 FOR NN=0 TO 9
150 INPUT "Give a value":N
152 REM Handle special case of NN=0.
154 IF NN=0 GOTO 206
156 REM For NN>0 open up insertion▶
   Point.
160 FOR NP=NN-1 TO 0 STEP -1
190 IF RA(NP)<N GOTO 210
200 RA(NP+1)=RA(NP)
202 NEXT NP
204 REM Special case of lowest value▶
   so far.
206 RA(0)=N:GOTO 212
208 REM Put N at insertion Point.
210 RA(NP+1)=N
220 REM
230 REM Print the values.
240 PRINT "The values are:";
250 FOR NN=0 TO 9
260 PRINT RA(NN);
265 NEXT NN
270 PRINT
```


CHAPTER 13: MORE ABOUT I/O

THE READ AND DATA STATEMENTS

Let's return to the date conversion program that we used in the chapter on arrays:

```
110 REM Convert a date to day-of-year.
120 REM In: month, day. A zero month▶
   means "end run."
130 REM Out: day of year.
140 REM Vars: MN=month of year. DM=day▶
   of month.
150 REM DR=day of year, the result.
160 REM M1=days Preceding 1st day of▶
   MN.
170 REM
180 REM MA= array of days in Year▶
   before each month.
190 DIM MA(12)
200 MA(0)=0:MA(1)=31:MA(2)=59:MA(3)=90
210 MA(4)=120:MA(5)=151:MA(6)=181:▶
   MA(7)=212
220 MA(8)=243:MA(9)=273:MA(10)=304:▶
   MA(11)=334
230 REM Prompt user for month, day.
240 INPUT "Month, day:";MN,DM
250 IF MN=0 THEN END
260 REM Compute M1, then DR.
270 M1=MA(MN-1)
280 DR=M1+DM
290 PRINT "Day of Year=";DR
300 GOTO 240
```

The 12 statements in lines 200 through 220 initialize the elements of the array, MA. Suppose we wrote a program that needed to initialize an array with a hundred elements? This would be a very inconvenient way to initialize such a large array!

In situations like this, Basic's **READ** and **DATA** statements are useful. READ inputs data to a program, as INPUT does. While INPUT gets data from the HHC's keyboard, READ gets data from values that you put in DATA statements, which are stored with the program itself.

Here is how our day-of-year program would look if we wrote it with DATA statements:

```
110 REM Convert a date to day-of-year.
120 REM In: month, day. A zero month▶
    means "end run."
130 REM Out: day of year.
140 REM Vars: MN=month of year.▶
    DM=day of month.
150 REM DR=day of year, the result.
160 REM M1=days Preceding 1st day of▶
    MN.
170 REM
180 REM MA= array of days in year▶
    before each month.
190 DIM MA(12)
200 FOR I=0 TO 11
210 READ MA(I)
220 NEXT I
230 REM PromPt user for month, day.
240 INPUT "Month, day:";MN,DM
250 IF MN=0 THEN END
260 REM ComPute M1, then DR.
270 M1=MA(MN-1)
280 DR=M1+DM
290 PRINT "Day of year=";DR
300 GOTO 240
310 DATA 0,31,59,90,120,151,181,212,▶
    243
320 DATA 273,304,334
```

Now lines 200 through 220 contain a FOR/NEXT loop that executes 'READ MA(I)' 12 times, for I = 0, 1, 2, ... 11. Each time READ is executed, it reads one value from a DATA statement. Thus MA(0) is assigned the value 0, MA(1) is assigned the value 31, and so forth. When READ runs out of values in the first data statement, it starts on the second one.

READ and DATA have two advantages over assignment for initializing variables and arrays:

- For initializing arrays, they are faster and more compact.
- They collect all the data in one place, making the data easy to modify if necessary.

Some Rules For Using READ and DATA

You can use any number of READ statements in your program. Each READ statement reads data each time it is executed, just as an INPUT statement would.

A READ statement may read data into any number of variables and/or array elements. For example, this statement reads data into three variables:

```
READ I,J,K
```

A DATA statement may be used anywhere in your program. If it appears in the middle of the program, flow of control simply goes around it. (But for your own convenience, it is sensible to put all the DATA statements in one place, at the beginning of the program or the end.)

The data items in DATA statements are read left-to-right and then from the beginning of your program to the end, just as you would read the words on a page. If a READ statement runs out of data items in one DATA statement, it goes on to the next one. If a READ statement has data items left over, it leaves them for the next READ statement to read.

If you try to read past the last data item in the last DATA statement, you will get

```
OD error
```

for "**O**ut of **D**ata error." On the other hand, no error will occur if your program fails to read all the data items available to it.

The RESTORE Statement

The **RESTORE** statement "rewinds" your program's DATA statements, so that the next READ statement will read the first item in the first DATA statement. The RESTORE statement looks like this:

```
RESTORE
```

You can execute RESTORE anywhere in your program, and you can execute it any number of times.

USING PERIPHERALS {B}{H}

In the chapter on file management you learned to use one kind of peripheral device: the RAM expander, which can act as a substitute for your HHC's internal memory. Now we will learn about another kind of peripheral, which can serve as a substitute for your HHC's keyboard or LCD display.

The HHC's **micro printer** is a typical peripheral of this kind. If you own a micro printer, you can plug it into the HHC's bus socket (or into an I/O adaptor) and use it to make printed **hard copy** of the HHC's output.

We're going to discuss the use of peripherals using the micro

printer as an example. If you have a different kind of printer, such as an 80-column printer connected to the HHC through the serial-interface peripheral, some of the following instructions won't apply to you. See your peripheral device's user's manual for information, or seek assistance from the distributor of your HHC.

{H} Getting Ready

Before starting to use the printer, load a roll of paper into it. Instructions for loading the printer are in the user's manual that comes with it.

{H} Connecting the Micro Printer

When you connect the micro printer to your HHC, you should follow the same procedure as when you connect a Programmable Memory Peripheral. To repeat:

1. Save your program by entering BYE, and return to the HHC's primary menu by pressing CLEAR.
2. Turn the HHC off.
3. Connect the micro printer.
4. Turn the HHC on again. Press CLEAR, if necessary, to make the HHC resume displaying the primary menu.

Connect the micro printer to your HHC now. After you turn the HHC on, press the I/O key. The HHC shows you the I/O key's menu. One of the items on the menu looks like this:

```
1=PRINTER OUT,OFF,SLOT=0
```

'PRINTER' is the I/O menu's name for the micro printer.

'OUT' means that the printer is an output device, like the LCD. 'IN' in this position would mean that the peripheral was an input device, like the keyboard.

'OFF' means that the printer is now turned off, and cannot be used, even though it is plugged in. We've entered the I/O menu in order to turn it on.

'SLOT = 0' means that the printer is plugged into the HHC's bus socket. If the printer were plugged into a I/O adaptor, this part of the line would tell you which slot of the I/O adaptor the printer was in: SLOT = 1, SLOT = 2, etc.

To turn on the printer, enter the number representing its menu selection. The HHC displays a message like this:

```
PRINTER OUT,ON,SLOT=0
```

indicating that the printer is now turned on. Then it resumes displaying the I/O key menu.

To leave the I/O key menu and return to the primary menu, press the I/O key again.

Writing Information To the Printer {B}{H}

You write information to the printer from a Basic program by using the PRINT statement. PRINT needs a way of telling whether you want any given PRINT to write to the printer, the LCD, or some other output device. It does this with a "logical unit number."

A **logical unit number** (**LUN** for short) is a number that appears between the word PRINT and the first data item to be printed, like this:

```
PRINT #2;MAC(I)
```

In the PRINT statement above, the LUN is #2. It is preceded by a '#' sign, and is separated from the following data item by a comma or semicolon.

Before you can execute such a PRINT statement, you must **attach** the printer to LUN #2 with the **ATTACH** statement. The ATTACH statement looks like this:

```
ATTACH 68 TO #2
```

'68' is a value that means you want to attach the micro printer. (It doesn't matter where the micro printer is plugged in; ATTACH will find it, so long as it's plugged in somewhere, and is turned on.)

'2' refers to LUN #2. (You could assign the value 2 to the variable XY, and then say 'ATTACH' 68 TO #XY'. You could also say 'PRINT #XY;'.)

Attaching the Printer {B}{H}

Enter Basic now and create a new program file.

Attach the micro printer to LUN #2 by entering

```
ATTACH 68 TO #2.
```

Now let's try using PRINT to write some information to the micro printer. Enter

```
PRINT #2;17
```

Basic should print '17' on the micro printer.

Try executing some more complex PRINT statements that address LUN #2.

Write a little program that prints on LUN #2, and run it. Notice that the ATTACH you performed in immediate mode is

effective for a program run in deferred mode. Device attachments are not reset when you run a program, as variable values are.

Enter BYE; select the same program file and run it again. Observe that the ATTACH you did is still in effect.

Enter BYE, then press CLEAR to return to the primary menu. Re-enter Basic, re-select the program, and run it. Observe that your ATTACH is ***not*** effective now. You get the error message:

```
IO error
```

which means (among other possible things) that you tried to do I/O on a LUN that nothing is ATTACHed to. All the ATTACHes that you do are cancelled whenever you return to the primary menu.

{B}{H} Input On Peripherals

You can read data from peripherals with INPUT, just as you can write data to peripherals with PRINT. For example, you can read data from LUN #3 with a statement like:

```
INPUT #3;XY
```

or

```
INPUT #3,"Enter 2 values:";XY,YZ
```

The only difference between preparing to do input and output on a peripheral is that for input, the I/O key menu says 'IN' instead of 'OUT'.

Note that any LUN may be used for input ***or*** output, but not both at once. If you plug in a device that is capable of both input and output, such as the Serial Interface Adaptor, the I/O key menu represents it as ***two*** devices, one for input and one for output. If you want to use the device for both input and output, you must turn on both of these one-way "devices" with the I/O key menu, and ATTACH both in Basic.

{H} More About LUNs

Basic recognizes LUNs from #0 through #15. The keyboard is normally attached to #0, and the LCD is normally attached to #1. #2 through #15 have no "normal" attachments, and are reset to "unattached" status by CLEAR.

Once a device is attached to a LUN, it remains attached until you return to the primary menu, or until you attach another device. Only one device at a time can be attached to a LUN.

If you use PRINT with no LUN, Basic assumes you are referring to LUN #1. Thus, PRINT with no LUN normally writes to the LCD, as you would expect.

Similarly, if you use INPUT with no LUN, Basic assumes that you are referring to LUN #0. Thus, INPUT with no LUN normally reads data from the keyboard, as you would expect.

Device Independence {B}{H}

Although ATTACH is an extra step that you must perform before using a peripheral, it gives you a valuable kind of freedom: you can write a program that uses a peripheral, without saying what peripheral it is to use. You can defer that choice until you run the program. We say that ATTACH makes your program **device independent**, since the way you write the program is independent of the kind of device you run it with.

For example, you could make a program write output to a micro printer one time you run it; the next time, simply by attaching a different device, you can make the program write output to a serial-interface printer connected to the HHC through a Serial Interface Adaptor.

You can also put ATTACH statements in your program, and let the program decide what to attach.

LISTing On the Printer {B}{H}

One very useful application of a printer is printing listings of a Basic program.[1]

To print a listing, plug in the micro printer, turn it on, and ATTACH it as above. Then enter the statement

```
LIST #2
```

to send a listing of your program to the printer.

After entering 'LIST #2' and pressing ENTER, press ◆ once to print each line of your program. You may press ENTER at any point to print one more line and end the list. To list the

[1] - The micro printer is not well suited to this application because of its its 15-character line length. Many Basic statements are longer than 15 characters, and the micro printer must print such statements on two or more lines, reducing the readability of a program.

You can get more readable program listings by using a **serial printer** with a line length of 80 characters or more, connected to the HHC through a peripheral called a **serial interface adaptor**. See the literature that came with your HHC for more information about this device.

whole program in one operation, press LOCK, then ◆.

If you want to start printing a list at some point after the first line of your program, you may specify a line number with LIST:

```
LIST #2;50
```

or

```
LIST #2,50
```

NOTE ON SIDE EFFECTS OF BASIC I/O

When Basic is executing a program, some of the HHC's standard facilities can function only when an I/O operation occurs. Specifically:

1. A clock control alarm can go off only when an I/O operation occurs.
2. The HHC's auto-off feature can turn the HHC off only when an I/O operation occurs.
3. If the HHC's battery power gets low, the 'BATT LOW' message can be displayed only when an I/O operation occurs.

CHAPTER 14: FUNCTIONS

Consider the following problem: you are writing a program, and at one point, you need to truncate a numeric variable to the next lower integer value.

Here is a statement that does the desired truncation:

```
X=INT (X)
```

The expression

```
INT (X)
```

is called a **function**. It operates on the value inside the parentheses, which is called the function's **argument,** and **returns** a value.

The purpose of the function INT is to return the largest **inte**ger value that is equai to or less than the argument. For example,

if X is	INT (X) is
-1.9	-2
-1.1	-2
-1	-1
0	0
1	1
1.1	1
1.9	1
9876543.2	9876543

The argument of a function may be any constant, variable, or expression. Most functions may be used anywhere a numeric value may be used:

```
PRINT INT (X)
FOR I=1 TO 10000 STEP INT (X)
Y=INT (X+1)^3
```

AN EXAMPLE

Let's look at a program where INT is useful. We're going to return to our day-of-year program, and modify it to make allowances for leap years.

We're going to add another variable for the year; let's call it YR. We'll prompt the user for a year value.Then, if MN>2 (representing a month after February) and the year is a leap year, we'll add 1 to DR.

The rules for determining whether any given year is a leap year are:

1. If the year is evenly divisible by 400, it is a leap year.
2. Otherwise, if the year is evenly divisible by 100, it is ***not*** a leap year.
3. Otherwise, if the year is evenly divisible by 4, it is a leap year.
4. Otherwise, the year is ***not*** a leap year.

How can we test for "A is evenly divisible by B?" That's where INT comes in. If YR is evenly divisible by 4, for example, then the logical expression

```
YR/4=INT (YR/4)
```

is true. If not, YR/4 is not an integer value, and the assertion is false.

Now we can write a version of the program that allows for leap years:

```
110 REM Convert a date to day-of-year.
120 REM In: month, day, & year. A zero▶
    month means "end run."
130 REM Out: day of year.
140 REM Vars: MN=month of year. DM=day▶
    of month. YR=year.
150 REM DR=day of year, the result.
160 REM M1=days Preceding 1st day of▶
    MN.
170 REM
180 REM MA= array of days in year▶
    before each month.
190 DIM MA(12)
200 FOR I=0 TO 11
210 READ MA(I)
220 NEXT I
230 REM PromPt user for month, day.
240 INPUT "Month,day,year";MN,DM,YR
250 IF MN=0 THEN END
260 REM ComPute M1, then DR.
270 M1=MA(MN-1)
280 DR=M1+DM
290 REM Allow for leaP years.
300 IF MN<=2 GOTO 350
310 IF YR/400=INT (YR/400)GOTO 340
320 IF YR/100=INT (YR/100)GOTO 350
330 IF YR/4<>INT (YR/4)GOTO 350
340 DR=DR+1
350 PRINT "Day of year=";DR
```



```
360 GOTO 240
370 DATA 0,31,59,90,120,151,181,212
380 DATA 243,273,304,334
```

SOME OTHER USEFUL FUNCTIONS

Here are several useful functions that Basic supports. Each of them requires a numeric argument. If one of these functions is given an integer argument, it automatically converts the argument to numeric form.

- **ABS(X)** returns the absolute value of its argument: X if X>=0, -1*X if X< 0.
- **EXP(X)** returns the constant E (approximated in Basic by the value 2.71828183) raised to the power X. The maximum value of X that will not produce an overflow error is 88.02969.
- **FRE(X)** returns the number of free bytes of memory available for storing and running programs. This is the size of the current file space, minus the amount of space already occupied by programs and data. X is ignored. {**B**}
- **INT(X)** returns the largest integer less than or equal to X.
- **LOG(X)** returns the the natural (base E) log of X. To obtain the base Y log of X, use the formula LOG(X)/LOG(Y). ***Example:*** the base 10 (common) log of 7 is LOG(7)/LOG(10).
- **POS(X)** returns the current position of the LCD cursor.[1] If the cursor is at the left edge of the LCD, POS(X) returns 0; if the cursor is one position right of the edge, POS(X) returns 1; and so on.
- **RND(X)** returns a "random" number in the range 0<RND(X)<=1. This function is more fully described in the ***Reference Guide,*** Chapter 2. {**B**}
- **SGN(X)** returns the "sign" of X: 1 if X>0, 0 if X=0, -1 if X<0.
- **SQR(X)** returns the square root of X. Equivalent to $X^{.5}$. Causes an "FC error" ("function call error") if X<0.

  SQR(X) produces the same result as $X^{.5}$, but executes more quickly.

Trigonometric functions such as SIN(X) and COS(X) are not supported in Microsoft Basic on the HHC. You can simulate them with formulas given in the ***Reference Guide,*** Chapter 2. {**B**}

---

[1] - This description of POS(X) is accurate as long as you do not PRINT to peripherals. See the ***Reference Guide,*** Chapter 2, for a description of POS(X) that deals with peripherals too.

USER-DEFINED FUNCTIONS

Basic has facilities that let you define your own functions.

You define a function by including a DEF statement in your program. You must execute the DEF statement that defines your function before you execute any statement that calls the function.

Here is an example of a DEF statement:

```
DEF FN AB (X)=X*(X+1)
```

'FN' is a reserved word meaning "function." It must always be present in a DEF statement.

{**B**} 'AB' is the name of the function being defined. The name must obey the rules that govern a numeric variable name. (But you may use the same name for a function and a variable, if you are willing to risk confusing yourself!)

The 'X' in parentheses is called a **formal parameter**. It stands for the value you will use in a call to the function. It may have any name that is valid for a numeric variable. (It is ***not*** a numeric variable, however; it has no relation to any variable with the same name that may be used elsewhere in the program.)

The expression on the right side of the '=' is the **body** of the function definition. When you call the function, the value that it returns is the value of the body, calculated after the value of the argument in the call is substituted for the formal parameter wherever the formal parameter appears in the body.

The Formal Parameter: Some Examples

The formal parameter is an elusive concept. Many people find it confusing the first time they encounter it. A few examples may help to make it clearer.

First, let's consider the sample definition above. Suppose we executed the following piece of code in a program that contains this definition:

```
310 DEF FN AB(X)=X*(X+1)
320 M = 5
330 PRINT M;FN AB(M)
340 X = 9
350 PRINT X;FN AB(X-2)
```

Line 310 defines the function.

In line 320, variable M is assigned the value 5.

Line 330 calls FN AB with the argument M. The value of M is 5; therefore, Basic substitutes 5 for X, the formal parameter of

FN AB, wherever X appears in the body of FN AB. Then it evaluates FN AB:

```
FN AB(X)=X*(X+1)
```

evaluates to

```
FN AB(5)=5*(5+1)
```

which evaluates to

```
FN AB(5)=30
```

Thus the call to FN AB returns the value 30, and line 330 displays '5 30.'

Line 340 assigns X the value 9. Line 350 calls FN AB with the argument X-2. The value of X-2 is 7; therefore, Basic substitutes 7 for X, the formal parameter of FN AB, wherever X appears in the body of FN AB. Then it evaluates FN AB:

```
FN AB(X)=X*(X+1)
```

evaluates to

```
FN AB(7)=7*(7+1)
```

which evaluates to

```
FN AB(7)=56
```

Thus the call to FN AB returns the value 56, and line 330 displays '9 56.'

Notice that calling the function had no effect on the value of the variable X. We repeat: the formal parameter X in the definition of FN AB is not a variable, and has no relation to the variable named X which is used elsewhere in the program.

Some Benefits Of Using Defined Functions

When a program must perform a certain calculation many times, taking one value and producing one result, you can gain the several benefits by defining a function to perform the calculation:

- Your program becomes shorter and easier to write, since you only have to code the function definition once.
- Your risk of making an error is reduced, for the same reason.
- Your program becomes more readable. When you see a use of the function, it is instantly clear that you are performing exactly that calculation, and not another that is almost

the same but not quite, or is entirely different but happens to look the same.

{B} Some Limitations On User-Defined Functions

Microsoft Basic imposes the following restrictions on a user-defined function:

- A user-defined function must have exactly one formal parameter, and each call to it must have exactly one argument. (The formal parameter need not be used in the body; if it is not, the value of the argument in the call is ignored.)
- The formal parameter must be numeric.
- The function must return a numeric result.
- The body of the function must be an expression; multiple-statement function definitions are not allowed, as they are in some other versions of Basic.
- A DEF statement may be executed only in deferred mode, not in immediate mode. (Once a function has been defined by DEF, however, the function may be called in immediate mode.)

CHAPTER 15: STRINGS

Some of the most interesting programs you can write with Basic manipulate not numbers, but strings of characters. You've already encountered string constants in statements like this one:

```
PRINT "Fuel efficiency=";MP;"mPg."
```

In this statement,

```
"Fuel efficiency="
```

and

```
"mPg."
```

are string constants. Each has a value that consists not of a number, but of a sequence (that is, a string) of characters.

STRING VALUES {B}

A string value may be anything from 0 to 255 positions long.{1} Each of the positions may have any of 256 distinct characters. The 256 characters include all the characters you can enter through the HHC keyboard and see on the LCD.

The idea of a string value that is zero characters long deserves a bit of explanation. We call such a string a **null string**. You might think a null string would be useless; actually, it is an indispensible concept in string processing, just as the concept of the number zero is indispensible in arithmetic.

You can represent the null string by a pair of quotation marks with nothing between them:

```
""
```

Note that upper and lower case characters are distinct in strings. For example, 'S' and 's' are two different characters, just as 'S' and 'Q' are. This makes string values (and particularly string constants) an exception to the general rule of Basic that everything is stored in upper case.

{1} - Sometimes we loosely use "character" to mean "position," as when we say that a string is "three characters long." We will avoid this usage in places where it might cause confusion.

STRING VARIABLES

Basic recognizes **string variables,** just as it recognizes numeric variables. These two kinds of variables are distinct; a string variable may not have a numeric value, or ***vice versa.***

A string variable may be assigned any string value. There is no need for you to state somewhere in the program how long the string's value may be.

You form the name of a string variable just as you form the name of a numeric variable, except that you must add the symbol '$' to the end.

For example, the following are all valid names for string variables:

```
NA$ X1$ Q$ QU$ QUESTN$
```

Basic treats QU$ and QUESTN$ as the same variable, since their first two letters are the same.

QUESTION$ would be an ***invalid*** name for a string variable, because it contains a Basic reserved word, ON.

As with integer and numeric variables, you can give string and numeric variables similar names like NA and NA$ without confusing Basic; but you should generally avoid doing so, to keep from confusing yourself.

SOME SIMPLE STRING OPERATIONS

You can perform many of the same operations with string values that you can do with numeric values. You can PRINT them, INPUT them, and assign them. You can also use string arrays.

{**B**} Microsoft Basic does ***not*** allow a user-defined function to have a string value as the argument, or to return a string value.

Assignment

You assign a string value to a string variable the same way you assign a numeric value to a numeric variable:

```
QU$ = "What shape?"
QU$ = Q3$
```

The INPUT Statement

INPUT and READ work the same way with string values as they do with numeric values. For example, if Basic executes

the following statement:

```
INPUT "What shape";SH$
```

Basic displays

```
What shape?
```

on the LCD, and waits for your reply. Whatever string you type in (and end with ENTER) is made the value of the variable SH$.

There is one character that you can't INPUT into a string variable this way. That is the comma. Comma, remember, is the character that separates two values in a reply to an INPUT statement. For example, if we execute this statement: {**B**}

```
INPUT "Shape, color, inches";SH$,CO$,▶
  IN
```

and respond to its prompt like this:

```
Shape, color, inches? square,blue,5
```

INPUT assigns SH$ the value 'square', CO$ the value 'blue', and IN the value '5.'

Similarly, if we execute this statement:

```
INPUT "What shape";SH$
```

and respond to its prompt like this ('␣' stands for 'space'):

```
What shape? "square␣with␣round▶
  ed␣corners"
```

INPUT would assign SH$ the value 'square'. '␣with␣roundedcorners', the second value in our response, would simply be discarded, since there is no second variable to assign it to.

You can give INPUT a comma as part of a string value by enclosing the ***whole*** string value in quotation marks: {**B**}

```
What shape? "square,␣with␣roundedcor▶
  ners"
```

There is no way you can give INPUT a quotation mark in the first position of a string value. You can include a quotation mark in a string value anywhere after the first position, however -- provided the string is not enclosed in quotation marks -- and INPUT will treat it just like any other character. {**B**}

Note that the prompt that begins an INPUT statement ***must*** be a string ***constant:***

```
INPUT "What shape? ";SH$
```

If you try to use a string variable here, Basic will simply read a value into the variable, even though the variable is followed by a semicolon rather than a comma.

The READ Statement

Everything we just said about INPUT applies equally to READ. Only the source of the data -- a DATA statement rather than the keyboard -- is different.

Note that Basic treats letters in a DATA statement the same way that it treats letter in any other statement -- it shifts them to upper case as it stores them. You can make Basic store a string value in lower case by enclosing the value in quotation marks. For example,

```
500 data january,february,march
```

will be stored like this,

```
500 DATA JANUARY,FEBRUARY,MARCH
```

but

```
500 data "january","february","march"
```

will be stored like this:

```
500 DATA "january","february","march"
```

String Values *vs.* Numeric Values In INPUT and READ

What happens if you give INPUT or READ a string value for a numeric variable, or a numeric value for a string variable?

The "numeric value for a string variable" question is really no question at all. As far as INPUT and READ are concerned, any numeric input is also string input. When INPUT sees '523,' for example, it doesn't know or care whether you think you are typing in the numeric value 523, or the string value '523'. It just looks at the type of variable that is to receive the next value, and acts accordingly.

If you give INPUT a string value such as 'square', and the next variable that should receive a value is a numeric

variable, Basic gives you the prompt

```
Reenter
```

and repeats the INPUT prompt. It won't let you get past the INPUT statement until you enter valid numeric input.

If you give READ a string value, and the next variable is a numeric variable, Basic gives you the error message

```
SN error in line nnn
```

where "nnn" is the line number of the DATA statement READ was trying to read from.

CONCATENATION

Concatenation is the operation of fusing two string values to make one. For example, if we concatenate the string values 'abc' and 'DEF', we get the string value 'abcDEF'.

Basic lets you concatenate two string values with the '+' operator. For example, you could assign the value 'abcDEF' to the variable S$ like this:

```
X$="abc"+"DEF"
```

You could concatenate the string '***' to the beginning and end of another string like this:

```
XS$="***"+X$+"***"
```

or like this:

```
AK$="***"
X$=AK$+X$+AK$
```

In the example above, 'AK$ + X$ + AK$' is an expression with a string value, just as '5*(X + 1)' is a expression with a numeric value. It may be used any place a string value may be used, for example:

```
PRINT AK$+X$+AK$
```

Comparison

You can compare two string values, just as you can compare two numeric values. For example,

```
IF X$="F"THEN X$="Array is full!"
```

If the value of X$ is 'F', this statement assigns X$ the new value 'ArrayØisØfull!'.

Here is another example:

```
IF X$=""THEN X$="OK"
```

If the value of X$ is null, this statement assigns X$ the new value 'OK'.

AN EXAMPLE: THE FUEL EFFICIENCY CALCULATOR

As an example of simple string processing, let's return to the fuel efficiency calculator that we developed in an earlier chapter. Here's a version that asks the user whether he wants to do another calculation, and lets him answer 'y' (for 'yes') or 'n' (for 'no') instead of entering a number. As an additional refinement, it gives the user an error message and reprompts him if he responds with anything except 'y' or 'n'.

```
10 REM Fuel efficiency calculator.
20 REM In: start & end odometer▶
   readings, fuel used.
30 REM Out: miles/gallon.
40 REM Vars: SR=start odom., ER=end▶
   odom.,
50 REM GA=gallons, SU=sum of results,
52 REM CO=count of results,  YN$=▶
   "more?" response.
60 INPUT "Start reading";SR
62 INPUT "End reading this time";ER
70 INPUT "Gallons used";GA
80 PRINT "Fuel efficiency =";
90 PRINT (ER-SR)/GA
100 PRINT "mpg."
110 SU=SU+MP
120 CO=CO+1
122 SR=ER
130 PRINT "More input? ";
140 INPUT "More input? Answer y or▶
    n";YN$
142 IF YN$="y"GOTO 62
144 IF YN$="n"GOTO 150
146 PRINT "That isn't valid.":GOTO▶
    140
150 IF CO>1 THEN PRINT "Average mpg=▶
    ";SU/CO
```


EXAMPLE: THE DAY-OF-YEAR PROGRAM

Here's a slightly more ambitious example. We've modified the day-of-year to read a three-character abbreviation of a month's name ('jan', 'feb', . . . 'dec') instead of a month number (1,2, . . . 12).

```
110 REM Convert a date to day-of-year.
120 REM Input: month, day, & year.  A▶
    null month means "end run."
130 REM Output: day of year.
140 REM Variables: MN$=month of year▶
    (3 chars).
142 REM DM=day of month, YR=year.
150 REM DR=day of year, the result.
170 REM
180 REM MA= array of days in year▶
    before each month.
185 REM MO$(I)=name of month I+1.
190 DIM MO$(12),MA(12)
200 FOR I=0 TO 11
210 READ MO$(I),MA(I)
220 NEXT I
230 REM Prompt user for month, day,▶
    year
240 INPUT "Month,day,year";MN$,DM,YR
250 IF MN$=""THEN END
252 FOR I=0 TO 11
254 IF MN$=MO$(I)GOTO 280
256 NEXT I
258 PRINT "Month name invalid!":GOTO▶
    240
260 REM Calculate DR.
280 DR=MA(I)+DM
290 REM Allow for leap years.
300 IF I<=1 GOTO 350
310 IF YR/400=INT (YR/400)GOTO 340
320 IF YR/100=INT (YR/100)GOTO 350
330 IF YR/4<>INT (YR/4)GOTO 350
340 DR=DR+1
350 PRINT "Day of year=";DR
360 GOTO 240
370 DATA "jan",0,"feb",31,"mar",59
380 DATA "apr",90,"may",120,"jun",151
390 DATA "jul",181,"aug",212,"sep",▶
    243
400 DATA "oct",273,"nov",304,"dec",▶
    334
```

We have added a string array, MO$, which is initialized to contain three-character abbreviations for the names of the months. To do each date conversion, we read a month name abbreviation into a string variable, MN$ (line 240), and search MO$ for a matching element (lines 244 through 248). If we find one, the subscript of that element is 1 less than the month number; the corresponding element of MA contains the number of days in the year preceding that month. From there on, the logic of the program is the same as before.

Notice what happens at line 250. If we reach the end of the loop, our month name must be invalid, since it didn't match any element of MO$. We give the user a message telling him what went wrong, and go back to the INPUT statement to give him another try.

What would happen if line 250 were not included? The program would give the user no warning if he entered invalid data; it would simply give him an invalid answer. This program happens to be written so that it would assume the same month that was entered on the previous loop (January on the first loop). Some otherwise correct variations of the program could respond in ways even more absurd.

There's a lesson in this: unless you intend to use your program once and throw it away, try to make it do something reasonable with every conceivable kind of invalid input, as well as with valid input. In most cases you'll save more time using a program with good error checking than you'll spend writing it. Every time you make a mistake in entering data, you'll be glad you did this. (Or, at least, you'll be sorry if you didn't do it!)

GETTING THE LENGTH OF A STRING

You can get the length of a string value (***i.e.,*** the number of characters in the value) with the LEN function. For example:

```
N=LEN (X$)
```

assigns the length in characters of the string X$ to the numeric variable N.

COMBINING STRINGS AND NUMBERS

It is often useful to convert values back and forth between numeric and string form. For example, suppose you want to display a number in "dollars and cents" format. You can convert the number to a string and then use string operations to put it in the proper form.

Basic does not let you assign a numeric value directly to a string variable, or ***vice versa.*** If you try, you will get the error message:

```
TM error
```

meaning, "**T**ype **M**ismatch error."

Converting a Number To a String

To convert a number to a string, use the STR$ function. (Every Basic function that returns a string value has a name that ends with '$'.) For example:

```
NU$=STR$(X)
```

This statement converts the value of the numeric variable X to a string value, then assigns it to the variable NU$.

When STR$ converts a numeric value to a string, it uses the same conversion rules that the PRINT statement uses when it displays a numeric value.

Converting a String To a Number

To convert a string to a number, use the VAL function. For example:

```
X=VAL (NU$)
```

This statement converts the value of the string variable NU$ to a numeric value, which it assigns to the variable X.

When VAL converts a string value to a number, it uses the same conversion rules that the INPUT statement uses when it reads a numeric value. If the string value is something like '5X', which cannot be interpreted as a number, VAL simply ignores everything from the first invalid character to the end. Thus VAL("5X") returns the value 5 ; VAL("FGHRTY") returns the value 0. {**B**}

EXTRACTING PIECES OF STRINGS

One interesting characteristic of a string is that you can deal with pieces of it. You can talk about the first five characters, or the last five, or the five characters beginning with the 18th character. Such a piece of a string is called a **substring**. Basic has a complete set of functions for extracting substrings from strings.

The LEFT$ Function

LEFT$ extracts a substring from the leftmost (beginning) part of a string.

```
NU$=LEFT$(X$,N)
```

In this example, LEFT$ returns a value consisting of the N leftmost characters of X$. X$ must be a string value; N must be a numeric value.

For example, consider this statement:

```
PRINT LEFT$("ABCDEFG",4)
```

This statement displays 'ABCD', the leftmost 4 characters of 'ABCDEFG'.

N, the second parameter of LEFT$, may be zero. If it is, LEFT$ returns the null string as its value.

If N is not an integer value, LEFT$ truncates it to the next smaller integer. If N's value is larger than the length of the string, LEFT$ returns the whole string. If N<0 or N>=256, LEFT$ gives an 'FC error' ("Function call").

The RIGHT$ Function

The RIGHT$ function is used like this:

```
NU$=RIGHT$(X$,N)
```

RIGHT$ returns -- you guessed it -- the rightmost N characters of X$. X$ must be a string value; N must be a numeric value.

For example, this statement:

```
PRINT RIGHT$("ABCDEFG",4)
```

displays 'DEFG', the rightmost 4 characters of 'ABCDEFG'.

Again, N may be zero, making RIGHT$ return the null string as its value. RIGHT$ treats invalid (too-large, non-integer, and negative) values of N the same way that LEFT$ does.

The MID$ Function

MID$ returns a substring taken out of the ***middle*** of a string. It is used like this:

```
NU$=MID$(X$,P,N)
```

P is the position of the substring in X$. If P is 1, the substring begins at the first position in X$; if P is 2, the substring

begins at the second position in X$; and so forth.

N, again, is the length of the substring.

For example, this statement:

```
PRINT MID$("ABCDEFG",2,3)
```

displays 'BCD', a substring of 'ABCDEFG' that begins at the second character and is three characters long.

If P> =LEN(X$), MID$ returns the null string. If P is not an integer value, MID$ truncates it; if P< 0 or P>=256, MID$ gives an 'FC error'.

If N=0, MID$ returns the null string. If N is greater than number of characters remaining in the string from the P'th character to the end, MID$ returns the entire part of the string from the P'th character to the end.

MID$ treats an invalid value of N the same way LEFT$ and RIGHT$ do.

EXAMPLE: DAY-OF-YEAR CONVERSION USING SUBSTRINGS

The following program is a variation on our day-of-year program. This version takes a completely different approach to converting a month name to a number: it keeps all the possible month name abbreviations in a single string, and compares the input name to 3-character substrings that begin at character #1, #4, #7, . . . until it finds a match.

Notice also that we start by taking a 3-character substring of the input name with LEFT$, so that the user can enter a month's whole name if he wants to.

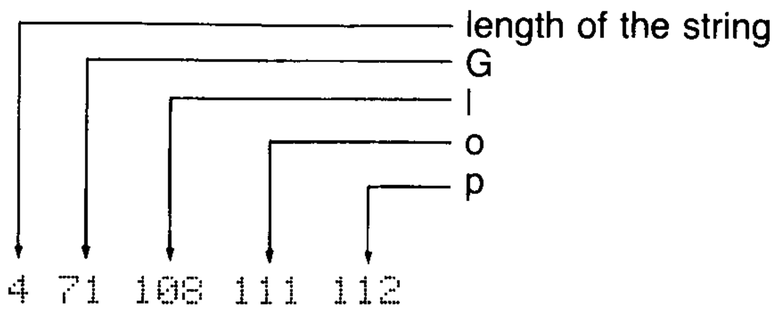
```
110 REM Convert a date to day-of-year.
120 REM InPut: month, day, & year. A▶
    null month means "end run."
130 REM OutPut: day of year.
140 REM Variables: MN$=month of year▶
    (3 chars).
142 REM DM=day of month. YR=year.
150 REM DR=day of year, the result.
170 REM
180 REM MA= array of days in year▶
    before each month.
185 REM MO$=array of month name▶
    abbr.s.
187 MO$="janfebmaraprmayjunjulaugsep▶
    octnovdec"
190 DIM MA(12)
```

```
200 FOR I=0 TO 11
210 READ MA(I)
220 NEXT I
230 REM Prompt user for month, day,▶
    year
240 INPUT "Month,day,year";MN$,DM,YR
250 IF MN$="" THEN END
251 MN$=LEFT$(MN$,3)
252 FOR I=0 TO 11
254 IF MN$=MID$(MO$,1+3*I,3)GOTO 280
256 NEXT I
260 PRINT "Month name invalid!":GOTO▶
    240
270 REM Compute M1, then DR.
280 DR=MA(I)+DM
290 REM Allow for leap years.
300 IF I<=1 GOTO 350
310 IF YR/400=INT (YR/400)GOTO 340
320 IF YR/100=INT (YR/100)GOTO 350
330 IF YR/4<>INT (YR/4)GOTO 350
340 DR=DR+1
350 PRINT "Day of Year=";DR
360 GOTO 240
370 DATA 0,31,59
380 DATA 90,120,151
390 DATA 181,212,243
400 DATA 273,304,334
```

MORE ABOUT CHARACTERS

Inside your HHC, each character is actually stored as a number between 0 and 255. Basic string values are represented by sequences of such numbers.

For example, the character 'G' is represented by the number 71. The string value 'Glop' is represented by the sequence of numbers:



Comparing Characters

Because of this property of strings, we can say not only whether one character is "equal to" another, but also whether one character is "greater than" or "less than" another. One character is greater than or less than another if the value that the HHC uses to represent it is less than or greater than the other.

For example,

'o' < 'p' (because 111 < 112).

'o' > 'l' (because 111 > 108).

Every lower case letter	>	Every upper case letter
(values 97 through 122)		(values 65 through 90)

Comparing Strings

By extension, we can say whether any string is greater than, equal to, or less than any other string. We do this by applying the following rules:

1. If two strings are the same length, and every character of one string is equal to the corresponding character of the other string, the strings are equal.
2. If two strings are not the same length, but are equal up to the point where the shorter one ends, the shorter string is "less than" the longer string.
3. In any other case, let position X be the first position in which the two strings differ; then the string with the lesser character in position X is "less than" the other string.

For example:

'Glop' = 'Glop'	(rule 1)
'Glop' < 'Gloph'	(rule 2)
'Glop' > 'GLOP'	(rule 3, second character)
'Glop' < 'glop'	(rule 3, first character)
the null string < every other string	(rule 2)

ASCII Code

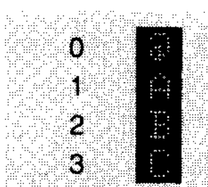
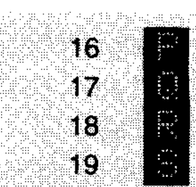
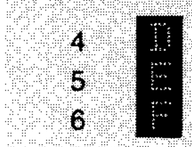
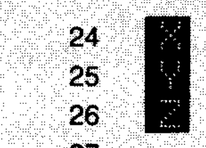
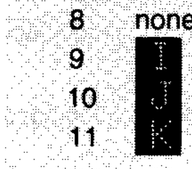
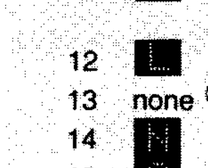
The HHC's system of representing characters by numbers is based on the **American Standard Code for Information Interchange** (**ASCII**), a system used on the vast majority of computers that are manufactured today. ASCII is an official standard adopted by the American National Standards Institute (ANSI), a division of the United States Department of Commerce.

The HHC's version of ASCII is shown in the table below. Standard ASCII characters are unshaded; characters unique to the HHC are shaded.

This table shows the characters that the HHC can display. Except for characters #0 through #31, and character #127 and up, these are all standard ASCII characters. You can enter each standard character on the keyboard by pressing the key that is labelled with the character that is displayed.

A following table shows how to enter some of the non-standard characters through the keyboard. (Not all of the HHC's displayable characters can be entered in this way.)

A more detailed version of this table is shown in the ***Reference Guide,*** Chapter 9.

numeric value	LCD display	numeric value	LCD display	numeric value	LCD display
0		16		32	SPace
1		17		33	!
2		18		34	"
3		19		35	#
4		20		36	$
5		21		37	%
6		22		38	&
7	none[2]	23		39	'
8	none[2]	24		40	(
9		25		41	)
10		26		42	*
11		27	none[2]	43	+
12		28		44	,
13	none [2]	29		45	-
14		30		46	.
15		31		47	/

[2] - This character has a special function when written to the LCD. See the following table.

numeric value	LCD display	numeric value	LCD display	numeric value	LCD display
48	0	80	P	112	p
49	1	81	Q	113	q
50	2	82	R	114	r
51	3	83	S	115	s
52	4	84	T	116	t
53	5	85	U	117	u
54	6	86	V	118	v
55	7	87	W	119	w
56	8	88	X	120	x
57	9	89	Y	121	y
58	:	90	Z	122	z
59	;	91	[	123	{
60	<	92	\	124	|
61	=	93	]	125	}
62	>	94	^	126	~
63	?	95	←	127	▒
64	@	96	`	128	↑
65	A	97	a	129	←
66	B	98	b	130	→
67	C	99	c	131	↓
68	D	100	d	132	Ä
69	E	101	e	133	É
70	F	102	f	134	Ñ
71	G	103	g	135	÷
72	H	104	h	136	×
73	I	105	i	137	■
74	J	106	j	138	□
75	K	107	k	139	ä
76	L	108	l	140	ö
77	M	109	m	141	ü
78	N	110	n	142	ñ
79	O	111	o		

The following characters have special functions when written to the LCD:

numeric value	LCD function
7	Makes HHC to "beep."
8	Backspaces cursor.
13	Clears LCD (as when ENTER is pressed).
27	Begins an escape control sequence (see the chapter on "Advanced I/O Techniques")

Here are keys that can be used to enter some of the HHC's non-standard characters:

numeric value	LCD display	key
7	"beep"	ROTATE
11	K	I/O[3]
20	T	HELP[3]
128	†	[3]
129	non-destructive backspace	[3]
130	non-destructive space	[3]
131	↓	[3]
132	Ä	INSERT[3]
133	Ë	DELETE[3]
137	■	SEARCH[3]
139	ä	C1[3]
140	ö	C2
141	ü	C3
142	ñ	C4

Converting Characters To Numbers...

The ASC function converts a character (more precisely, a one-character string) to the number that represents it in ASCII notation. Here is an example of ASC:

```
N=ASC ("g")-ASC ("G")
```

[3] - These keys may be read only by the GET statement, not the INPUT statement.

This statement converts two one-character strings, 'g' and 'G', to numeric form, subtracts one from the other, and assigns the difference to N.[4]

If the argument of ASC is more than one character long, ASC converts the first character. If the argument of ASC is the null string, ASC gives an "FC error."

...And Back

The CHR$ function converts a number to the character that the number presents in ASCII notation. Here is an example of CHR$:

```
X$="g"
X$=CHR$(ASC (X$)-32)
```

The first statement assigns X$ a value which consists of a lower-case letter. The second statement converts the character in X$ to numeric form, subtracts 32 from it, converts it back, and assigns it back to X$. The total effect is to convert the lower-case letter in X$, whatever it may be, to upper case.

If CHR$'s argument is not in the range 0 to 255, CHR$ will give an "FC error."

Example: Forcing Characters To Lower Case

A sophisticated program often asks its user for string input, and then searches a string array for the string he entered. Such a program generally should translate the input to all-upper-case or all-lower-case, so that the string array doesn't have to include every possible combination of cases, like 'square', 'Square', 'SQUARE', etc.

Here is a piece of code that forces all the characters in a string to lower case:

```
500 REM Input: string in IN$.
510 REM Output: OU$ = IN$ forced to▶
    all lower case.
520 REM Works with I,L,C$.
530 L=LEN (IN$)
540 OU$=""
550 FOR I=1 TO L
560 C$=MID$(IN$,I,1)
570 IF C$>="A" AND C$<=" "THEN▶
    C$=CHR$(ASC (C$)+32)
580 OU$=OU$+C$
590 NEXT I
```

[4] - The number that represents any lower case letter is 32 greater than the number that represents the corresponding upper-case letter. We'll make use of this fact in a moment.

VAL and ASC

VAL and ASC are both conversion functions that expect a string argument and return a numeric value; yet they have quite different uses. One of the most common errors made by beginning Basic programmers is to confuse these two functions.

VAL converts a string representing a number to the number. ASC converts a character to the number that represents the character in ASCII notation.

For example, consider the following two statements:

```
M=VAL ("235")
N=ASC ("235")
```

In the first statement, VAL converts the ASCII string '235' to the numeric value 235, and so the statement assigns M the value 235.

In the second statement, ASC pays attention only to the first character of its value, which is '2'. The numeric representation of the ASCII character '2' is 50, so the statement assigns N the value 50.

STR$ and CHR$

Similarly, STR$ and CHR$ are both conversion functions that take a numeric argument and return a string value; yet they have quite different uses.

STR$ converts a number to a string that represents the number. CHR$ converts a number to the character that the number represents in ASCII notation.

For example, consider the following two statements:

```
M$=STR$(50)
N$=CHR$(50)
```

In the first statement, STR$ converts the number 50 to the string value '50', and so the statement assigns M$ the value '50'.

In the second statement, CHR$ converts the number 50 to the ASCII character represented by the number 50. That is '2'; so the statement assigns N$ the value '2'.

{B} THE GET STATEMENT

Basic has one more statement for reading data that can be extremely useful when you are processing strings. This is the GET statement. It looks like this:


```
GET X$
```

or

```
GET #N,X$
```

where X$ is a string variable, and N is a numeric value specifying a logical unit number (LUN).

GET reads data from the keyboard or from a peripheral, as INPUT does. It differs from INPUT in two important ways.

GET Reads One Character

First, GET reads a single character rather than a string of characters ended by ENTER. It assigns its string variable a one-character value containing the character it read.

GET considers ENTER to be a character like any other. For example, if you give GET the character 'p', it assigns a 'p' (numeric value 112) to the string variable. If you give GET the character ENTER, it assigns an ENTER (numeric value 13) to the string variable.

There are several special keys that Basic considers equivalent to ENTER in most contexts, such as the SEARCH and HELP keys. GET reads these keys as distinct characters. That can be very useful if you need a key that you can assign some meaning of your own without risking a conflict with other meanings that Basic might have for it.

GET Does Not Echo

Unlike INPUT, GET does not automatically **echo** characters on the LCD as you type them. If you want the characters you type to appear on the LCD, you must display them yourself. If you want ***something else*** to appear on the LCD as you type -- the characters that you type, forced into upper case, for example -- you are free to do so instead.

GET's lack of echoing gives you complete control over the way your program treats its input. For example, you can ask the user to type a command code and then you can display the command's name, rather than the code, if the code is valid. You can display an error message if the code is not valid. Or, you can ask the user for a password before giving him certain information, and avoid echoing the password so that another person watching what the user does cannot read it and steal it.

One useful application of GET is in reading the four keys labelled C1 through C4, which are located in the lower left corner of the keyboard. You can use these keys to command

your program to perform commonly used operations, much as the HHC itself uses keys like CLEAR, I/O, and SHIFT.[5]

Caution Against Typing Ahead

The HHC generally lets you "type ahead" of the program you are running. If you want to enter keystrokes before the program is ready to input them, you may do so; when the program inputs the keystrokes, they will be waiting.

Basic is an exception to this rule. For reasons too technical to go into here, you cannot type ahead when your keystrokes are being read by Basic programs. If you try, the program will miss many of the keystrokes you enter.

SOME IDEAS FOR PRACTICE

Here are some small projects you can use for practice if you want to become more familiar with string processing:

- Write a program that tells you how many times a given substring appears in a given string.
- Write a program that displays the mirror image of a string (placing the first character last, the second character next-to-last, etc.).
- Write a program that converts a numeric value to a string in dollars-and-cents format, and displays the string. For example, this program should convert 200 to '$2.00'; 20 to '$.20'; 2 to '$.02'; .0097 to '$.01'.
- Write a program that reads a number into a string variable and converts it to a numeric value, ***checking for every possible error*** so that the user can get a nice message like 'Invalid; please try again' instead of INPUT's brusque 'Reenter'.
- Write a program like the preceding one, with this addition: as soon as the user enters a character that makes the input invalid, the program refuses to echo the character, and waits for a valid character. Let ENTER or 'space' end the number.

You'll have to use GET to write this program. Can you write it by modifying your previous one, or do you have to start over? If you have to start over, do you see a way you could have written the previous program that would have made it adaptable to this purpose . . . if you'd only known? . . .

---

[5] - But be careful; the C1 key is your tool for halting the execution of a malfunctioning program, and GET pre-empts that use by reading C1 as an ordinary character. You can ***usually*** get around this difficulty by pressing C1 several times in rapid succession. If your program grabs the C1's as fast as you can enter them, you will have to press CLEAR once.

CHAPTER 16: SUBROUTINES

As you write Basic programs, you will develop many pieces of code that you want to use over and over. Often you will need to incorporate the same piece of code in several places in a program.

It would be nice if you could include such a piece of code in your program just once, and GOTO it wherever you need the function it performs. This would create a problem: how could the code return control to the part of your program that did the GOTO, when that could be in any of several places?

With Basic's **GOSUB** and **RETURN** statements, your program can do this.

Whenever you need to execute a frequently-used piece of code, execute a GOSUB to the code:

```
GOSUB 1020
```

GOSUB is like GOTO, except that when Basic executes a GOSUB, it saves a pointer to the next statement after the GOSUB. The code that you GOSUBed to can pass control back to the statement after the GOSUB, wherever that statement may be, by executing a RETURN statement:

```
RETURN
```

Thus, you can GOSUB to the same piece of code from any number of places in your program, and when the piece of code is done executing, it can RETURN to the proper place, without knowing what statement called it.

The code that you GOSUB to is called a **subroutine**. We say that a **calling routine** uses GOSUB to **call** a subroutine.

A SIMPLE EXAMPLE

Here is a simple program that illustrates how GOSUB and RETURN are used:

```
10 AA=8:AB=5:GOSUB 1000
20 PRINT AC
30 AA=15:AB=-9:GOSUB 1000
40 PRINT AC
50 A(1)=8:A(2)=9:A(3)=14:A(4)=25:A(0)▶
   =4:GOSUB 2000
60 PRINT AC
70 END
1000 REM Add 2 numbers: AC=AA+AB.
1010 AC=AA+AB
1020 RETURN
```

```
2000 REM Add N numbers: AC=A(1)▶
     +...+A(A(A0))
2010 AC=0
2020 FOR I=1 TO A(0)
2030 AB=A(I):AA=AC
2040 GOSUB 1000
2050 NEXT I
2060 RETURN
```

The program consists of a **main routine** from lines 10 through 70, and two subroutines, beginning at lines 1000 and 2000.

The subroutine at line 1000 expects input in the variables AA and AB. In lines 10 and 20, we set up input to the subroutine in these variables, call the subroutine, and display the result that it returns in AC. In lines 30 and 40 we do the same thing a second time, and in lines 50 and 60 we do it a third time.

Of course, this subroutine is so simple that it is hardly worth using except to illustrate how to do a GOSUB. But it could just as well perform a task that required dozens of statements. If it did, then having the subroutine would shorten our program by about 2/3.

Now look at line 50, where we set up and execute a call to the other subroutine, beginning at line 2000. That subroutine is a more complex one. It adds a ***variable*** number of values, which it gets from array elements A(1), A(2), A(3), . . . The number of elements to add is found in A(0).

Look at the way this subroutine performs the addition. ***It calls the other subroutine.***

Basic lets you write a subroutine that calls another subroutine. We call this kind of call a **nested** call, and say that the call to the subroutine at line 1000 is nested within the call to the subroutine at line 2000. Basic lets you execute nested subroutine calls to a depth of about 22 calls.

ANOTHER EXAMPLE: THE DIFFERENCE BETWEEN TWO DATES

To illustrate the usefulness of subroutines, we're going to make another extension to our day-of-year calculator. We're going to make it perform a completely new task: calculating the ***difference,*** in days, between two dates in the same year. For example, we will make it tell us that the difference between March 8 and March 15 is 7 days, and the difference between April 8 and March 15 is -24 days.

When we develop a plan for this program, we immediately see that the day-of-year calculator will do most of the work for us:

1. Get the first date and convert it to a day of year.
2. Get the second date and convert it to a day of year.
3. Subtract the first date from the second date, giving the difference in days.
4. Print the result.

We can easily turn the day-of-year calculator into a subroutine and write a very short main routine to call it:

```
10 REM Date difference calculator.
20 REM In: user is PromPted for Year▶
   & 2 dates.
30 REM Out: number of days from 1st▶
   date to 2nd date
40 REM
100 REM Set-uP for day-of-Year▶
    subroutine.
190 DIM MO$(12),MA(12)
200 FOR I=0 TO 11
210 READ MO$(I),MA(I)
220 NEXT I
370 DATA "jan",0,"feb",31,"mar",59
380 DATA "aPr",90,"maY",120,"jun",151
390 DATA "jul",181,"aug",212,"seP",▶
    243
400 DATA "oct",273,"nov",304,"dec",▶
    334
410 REM
1000 REM Main routine.
1010 REM YR=Year; MN$=month; DM=day▶
     of month;
1020 REM D1=1st day-of-Year; YN$=▶
     Yes/no resPonse.
1030 INPUT "What Year";YR
1040 INPUT "1st month,day";MN$,DM
1050 GOSUB 2000
1060 D1=DR
1070 INPUT "2nd month,day";MN$,DM
1080 GOSUB 2000
1090 PRINT "Difference is";DR-D1;▶
     "days."
1100 INPUT "Again";YN$
1110 YN$=LEFT$(YN$,1)
1120 IF YN$="y"GOTO 1030
1130 IF YN$="n"THEN END
1140 PRINT "Answer 'y' or'n".":GOTO▶
     1100
1150 REM
```

```
2000 REM Subroutine to calculate day▶
     of year.
2010 REM In: YR=year, MN$=month,▶
     DM=day-of-month.
2020 REM Out: DR=day-of-year.
2030 REM Used: I=loop index, M1=days▶
     -before-month.
2040 FOR I=0 TO 11
2050 IF MN$=MO$(I)GOTO 2090
2060 NEXT I
2070 PRINT "Month name invalid!":▶
     RETURN
2080 REM Compute M1, then DR.
2090 DR=MA(I)+DM
2100 REM Allow for leap years.
2110 IF I<=1 THEN RETURN
2120 IF YR/400=INT (YR/400)GOTO 2150
2130 IF YR/100=INT (YR/100)THEN▶
     RETURN
2140 IF YR/4<>INT (YR/4) THEN RETURN
2150 DR=DR+1
2160 RETURN
```

Note On Errors In Subroutines

If the user enters an invalid month name, the date difference calculator shown here gives an error message, then goes ahead and calculates a meaningless result. To avoid calculating a meaningless result, we would have to modify the subroutine to return an 'invalid input' indicator. The indicator might be an otherwise impossible value of DR, or a value in a new variable added to the program for that purpose. We would also have to modify the main routine to check the "invalid input" indicator, and avoid calculating any result when that indicator was found.

When you write a subroutine, you will usually have to give some additional thought to such error conditions. To make a program handle an error when a subroutine is involved, you must

1. Make the subroutine recognize the error and note it by setting some variable to an appropriate value, and
2. Make the calling routine act appropriately when the subroutine indicates that an error has occurred.

Planning Ahead

There are ways we could have planned this program that would have made the day-of-year calculator useless. By seeing the day-of-year calculator's potential usefulness,

though, we were able to plan our program to take maximum advantage of work we had already done.

This is an important principal in program design. ***When you write programs, build on your past work.*** Design your programs so that you can re-use code you have already written. Write your programs so that you can re-use their parts in the future.

GENERALITY *VS.* EFFICIENCY

Notice that the subroutine in the date-difference calculator does the "leap year?" calculation each time it is called, because we chose not to modify the heart of the day-of-year calculator at all. There were some good reasons for making that choice:

- We saved ourselves some work.
- We saved ourselves the risk of introducing a bug in the subroutine, which we would then have had to fix.
- We preserved the subroutine's usefulness for its original purpose.

Now, if we want to go back and make the day-of-year calculator work by calling the subroutine, we can easily do so. And if we should later find a bug in the subroutine, we can fix it in both programs without having to deal with two different versions of the code.

Thus, there are a number of advantages to keeping the day-of-year subroutine's code as it is. But there are some disadvantages, too. When we run the date difference calculator, it will do the "leap year?" calculation twice, even though it is logically necessary to do that calculation only once. Thus, the program is dong more work than it has to, and is running more slowly than it has to.

In this case, our decision not to change the day-of-year code has gained us **generality** at the expense of **efficiency.** The program runs less efficiently than it could, but the subroutine is more generally useful than it would otherwise be.

This sort of **trade-off** is common. When we design a program, we often must choose among conflicting virtues like efficiency, generality, compactness, reliability, and accuracy.

Should a program be small or should it be accurate? Should it be efficient, or should it be generalized? Those are questions we have to face each time we design a program. We answer them in the light of the program's purpose.

In the case of the date-difference calculator, the additional processing time that the "inefficient" program takes is so

slight that the program still appears to run instantaneously. Thus the real cost of our design choice is nil. Generality wins.

About Line Numbers and Subroutines

Notice the line numbers in our first sample program. They take a big jump at the start of each subroutine. This is intentional. First, it emphasizes the fact that each subroutine is a unit distinct from the code that precedes and follows it. Second, it makes each subroutine easy to expand without disturbing the line numbers around it. For example, we could change the function of the subroutine at line 1000, and expand it to several dozen statements, without having to re-number the lines that follow it. The freedom to do this can be invaluable when we want to make extensive changes to a program.

If we have to make extensive additions to a long piece of code, we can (and usually should) write the additions as a new subroutine. Then we can add the subroutine at the end of the program, and add little more than GOSUBs to the existing code.

What Should Go Into a Subroutine?

As you design programs, you often must to ask yourself two related questions: first, "When should I write a subroutine?" and second, "when I write a subroutine, exactly what part of my program's function should I include in it?"

In deciding when to write a subroutine, consider:

- Is there a sensible way to design your program so that a subroutine is useful in two or more places?
- If a certain part of the program can be written as a subroutine, is the subroutine likely to be useful in future programs? A subroutine is easier to "transplant" to a new program than a piece of ordinary code, since its relationship to the code that surrounds it is easier to understand.
- Are you adding a lot of code to an existing program? If so, the addition will be easier to make if you make it in the form of a subroutine.

In deciding just what to include in a subroutine, consider:

- What will maximize the usefulness of the subroutine? If you include too little of your program's function, the subroutine's usefulness is reduced because it does less than it could. If you include too much, the subroutine's usefulness is reduced because it is harder to utilize in many different contexts.
- What will simplify the subroutine's **interface** -- the things you have to know about the subroutine in order to call it? A simple interface makes a subroutine easy to use, and tends to increase its generality. If your subroutine's interface is complex, look for a different way to define the subroutine that would simplify it.

For example, consider our date-difference calculator. We could have shortened the program by making the subroutine display a prompt, as well as inputting and analyzing data. We could pass a word like '1st' or 'next' to the subroutine in a string variable named W$, and start off the subroutine like this:

```
2040 PRINT "Enter the ";W$;" date: ";
2050 INPUT MN$,DM
       :
       :
```

If we did this, we would add another variable, W$, to the subroutine's linkage. We would have to note the meaning of W$ in the comments; and we would restrict the prompt to the formats that line 2040 (above) could display. Instead, we chose to keep the linkage simple and generalized by letting the calling routine display the prompt and get the input values.

ERRORS THAT GOSUBS CAN CAUSE

Remember that every GOSUB to a subroutine must be matched by a RETURN from the subroutine to the calling routine. Failure to observe this rule is one of the most common errors associated with the use of GOSUBs.

If you try to do a RETURN without having done a matching GOSUB, you will get the message:

```
RG error
```

meaning "**R**eturn witho**U**t **G**osub error."

If you try to nest GOSUBs more deeply than Basic allows, you will get the message

```
OM error
```

meaning, "**O**ut of **M**emory error."

THE ON/GOSUB STATEMENT

The **ON/GOSUB** statement is like the ON/GOTO statement, except that it does a GOSUB instead of a GOTO. It is useful when you want to call one of several subroutines, and can choose one depending on whether the value of a variable is 1, 2, 3, etc.

Here is an example of an ON/GOSUB statement:

```
ON XX GOSUB 1000,6000,5000
```

If XX (truncated to the next lower integer value, if necessary) is 1, this ON/GOSUB calls a subroutine at line 1000. If XX is 2, the ON/GOSUB calls a subroutine at line 6000; if XX is 3, it calls a subroutine at line 5000. If XX is less than 1 or greater than 3, the ON/GOSUB does nothing; that is, it calls no subroutine.

CHAPTER 17: PEEKS AND POKES

INTRODUCING POKE

The HHC has a number of features that Basic cannot enable you to use directly. You can use many of these features by manipulating parts of the HHC's memory where data about the status of the HHC and the Basic interpreter are stored.

Basic has a very powerful statement, **POKE**, that lets you manipulate the HHC's memory in this way.

For example, suppose you want a program to attach a peripheral device, such as a printer, to LUN #1. When the program is done, you want it to re-attach the LCD to LUN #1. But looking in the ***Reference Guide***, Chapter 7, you find that since the LCD is not a peripheral, it has no device code. How can you re-attach it?

You can re-attach the LCD to LUN #1 by POKEing a number into a table that tells the HHC's I/O routines what device is attached to each LUN.

How POKE Works

The POKE statement looks like this:

```
POKE AD,CN
```

AD is the **address** of a byte in the HHC's memory. That is, AD, is a value between 0 and 65535 that identifies a particular byte.

CN is an integer value between 0 and 255. (This is the range of values a byte can hold.)

POKE stores the value of CN at the address given by AD.

An Example: Reattaching the LCD to LUN #1 {B}

The HHC keeps track of LUN attachments through the **System Device Table** (**SDT**). You can attach the LCD to a LUN by POKEing the value 6 into one of the entries in the SDT.

The SDT is kept at memory locations 705 through 712. The byte at 705 represents LUN #0; the byte at 706 represents LUN #1; and so forth.

To re-attach the LCD to LUN #1, execute this POKE:

```
POKE 706,6
```

CAUTION!!!

POKE works great as long as you POKE the right thing into the right place at the right time. If you make a mistake with POKE there is no danger that you will do physical harm to your HHC, but you may do serious damage to your program or to the files in your HHC's storage.

{**B**} Here are some of the nasty things you can do to your HHC with incorrect POKE statements.

1. You can change the contents of storage occupied by a file in the HHC's memory. This could change the the contents of a file, or it could turn the file into nonsense, so that the HHC can't even list it.
2. You can destroy the integrity of the file system, forcing the HHC to erase all the files that are stored in its memory.
3. You can change data that the HHC needs to perform basic functions like detecting data entered on the keyboard or forming characters on the LCD. Then your HHC will not do anything until you return to the primary menu with the CLEAR key.
4. You can change certain critical data that the HHC needs to respond properly to the CLEAR key. If this happens, you must restart your HHC by turning it off and back on with the ALL OFF switch on the back of the case.
5. You can change the contents of an area where the HHC maintains a record of the current time and date, so that you must reset the time and date when you are done POKEing.

For safety's sake, we recommend taking the following precautions when you use POKE:

1. Pause and ask yourself if you really need to do a POKE.
2. Very carefully define the operation of the code that does the POKE. Keep it short and simple; put it all in one part of your program, and use remarks to document it thoroughly.
3. Before testing the program, copy all your files (including the program) to a Programmable Memory Peripheral or other storage medium. Ensure that if the file system is erased by an error, you won't lose anything that is hard to replace!
4. Pause and ask yourself again if you really need to do a POKE. If the answer is still "yes". . . . go ahead.

INTRODUCING THE PEEK FUNCTION

Just as you can use the POKE statement to store data directly into the HHC's memory, you can use the PEEK function to fetch data directly from the HHC's memory.

The PEEK function looks like this:

```
X=PEEK (AD)
```

AD represents the address of a byte in the HHC's memory.

PEEK returns an integer value between 0 and 255, giving the contents of the memory location at address AD.

Note On PEEKing Two-Byte Fields

The HHC stores an integer value or an address in a two-byte field. The ***first*** byte is the ***less*** significant, and the ***second*** byte is the ***more*** significant.

You may think of such a value as a two-digit base 256 number in which the 1's place is on the left, and the 256's place is on the right.

For example, if the integer value 300 were stored in locations 1000 and 1001, it would look like this:

location	1000	1001
contents	44	256

You could PEEK and reconstruct the value like this:

```
VL=PEEK(1000)+256*PEEK (1001)
```

Example: Is a Device Attached To a LUN? {B}

You can use PEEK to determine whether a device is attached to a particular LUN. This could be useful to determine whether your program should or should not try to do I/O on that LUN.

If no device is attached to a particular LUN, the SDT entry representing that LUN has the value 255. If a device is attached to the LUN, the SDT entry representing that LUN has some other (lower) value.

The following code shows how you could PEEK at the SDT entry for LUN #4 and write information to that LUN if anything is attached to the LUN:

```
500 REM Print debug info if LUN #4▶
    attached.
510 IF PEEK (709)=255 THEN RETURN
520 PRINT #4; . . .
      :
      :
580 RETURN
```

OTHER PEEKS AND POKES

For a list of useful PEEKs and POKEs on the HHC, see the ***Reference Guide***, Chapter 8.

CHAPTER 18: USING THE FUNCTION KEYS

The HHC has three special keys, called **function keys**, that are labelled **f1, f2,**and **f3**. {B} {H}

You can define each of the function keys to represent a string of up to 15 keystrokes. Then, whenever you press one of the function keys, the HHC will respond just as if you had entered the string of keystrokes that the function key represents.

Function key definitions are not erased by the CLEAR key. Barring accidents (***e.g.***, with POKE), they stay around until you change them, or until you turn the HHC off with the ALL OFF switch.

DEFINING A FUNCTION KEY {B}

You can only define a function key when you are in Basic's menu or the primary menu, or when you are running a program other than Basic.

To define a function key,

1. Press the HELP key (above the ◖ key). The HHC displays the message 'PRESS KEY FOR DEFINITION'.
2. Press the function key you want to define. The HHC displays 'DEFINE FUNCTION', and then displays the current definition (if any) of the function key you pressed. It leaves non-blinking underscore cursor to the right of the definition.{1}
3. Enter the string of keystrokes you want this function key to represent. This string may include any key except ON, OFF, another function key, and CLEAR. For example, it may include the ENTER key.

   The HHC erases the function key's previous definition and displays the keystrokes you enter, beginning at the left edge of the LCD.
4. When you are done, press the CLEAR key. This completes the function key definition and returns the HHC to whatever it was doing when you pressed HELP.

Note: since a function key definition may include ◗ and ◖, you can't use those keys to edit a definition! If you make a mistake, you must finish the definition and start over.

{1} - If the previous definition is 15 characters long, the HHC displays only the first 14 characters of it.

EXAMPLE: A FUNCTION KEY FOR LIST

Let's define the f1 key to represent the LIST command.

Press the HELP key, then the f1 key. Now enter the five keystrokes **l i s t** and ENTER. Press CLEAR.

Now enter Basic and select one of your programs from Basic's menu. Press the f1 key and watch Basic begin listing your program.

Suppose you had included L, I, S, and T, but not ENTER, in the definition of f1? Then pressing f1 would make the HHC respond as if you had pressed L, I, S, and T, but not ENTER. You would have to press ENTER after f1 to make the HHC begin listing your program.

{B} DISPLAYING A FUNCTION KEY'S DEFINITION

To display the definition of a function key, press HELP and then the function key. The LCD displays DEFINE FUNCTION again, and then the definition of the function key.

When you are done looking at the definition, press CLEAR.

For example, to display the definition of a f1, press HELP, then f1. When you are done reviewing the definition, press CLEAR.

{B} SPECIAL KEYS IN A FUNCTION KEY DEFINITION

Did you notice the odd symbol that the LCD displayed for the ENTER key in the definition of F1? That symbol is an **inverse-image** 'M' -- an 'M' formed from clear dots on background of black. The HHC displays it because the ASCII code for the ENTER key (it is 13) places an inverse-image 'M' on the LCD when it is displayed.

You can put most of the HHC's special keys in a function key definition. Each of these keys will have its usual effect when you call up the function by pressing the function key, not when you place the keystroke in the function definition by pressing the key.

Every one of the special keys you can put in a function definition displays its own unique symbol on the LCD:

key	*symbol*	*key*	*symbol*	*key*	*symbol*	*key*	*symbol*
▲	↑	HELP	T	ROTATE	G	C1	á
▼	↓	I/O	K	INSERT	A	C2	é
◀	←	STP/SPD	H	DELETE	E	C3	ó
▶	→	SEARCH	■	ENTER	M	C4	ñ

You cannot put the following special keys into a function key definition: another function key, ON, OFF, CLEAR, SHIFT, and 2nd SFT. (But SHIFT and 2nd SFT have their usual effects on other keys.)

HOW TO CORRECT A FUNCTION DEFINITION {B}

Notice that pressing ◀ stores a '◀' into a function definition. If you make a mistake while entering a function definition, then, how can you correct it?

To correct a mistake in a function definition you must re-enter the entire definition. To do this,

1. End the definition by pressing CLEAR.
2. Press HELP, then the appropriate function key, as though you were reviewing the definition.
3. Start entering the correct definition. When you enter the first keystroke, the old function key definition disappears from the LCD, to be replaced by the new one you are entering.
4. To end the new definition, press CLEAR.

HOW TO ERASE A FUNCTION DEFINITION {B}

You can erase a function definition by pressing HELP, then the appropriate function key, then the ***same*** function key ***again***.

LIKELY USES FOR FUNCTION KEYS {B}

Here are some function key definitions that you may find valuable:

- The LIST command, possibly followed by ENTER, as in our example above.
- The RUN command, possibly followed by ENTER.
- The BYE command, possibly followed by ENTER.
- Frequently used Basic reserved words such as GOTO, INPUT, etc.; or frequently used phrases, such as 'IF A(0) = -1'.
- A sequence of keystrokes that you must enter over and over at a particular point in your work; for example, a sequence of keystrokes to perform an editing operation that you must apply to many of the lines in your program.

CAUTION AGAINST TYPING AHEAD

Just as Basic does not work properly if you try to type ahead of your program, it does not work properly if you type a function

key to simulate multiple inputs. It misses some or all of the characters in the function key's definition. Therefore, do not try to use a function key to input with several INPUT or GET statements at one time.

CHAPTER 19: ADVANCED I/O TECHNIQUES

In this chapter we will cover some ways of doing sophisticated I/O operations with HHC Basic.

CONTROL CHARACTERS

The ASCII codes that represent normal, "displayable" characters begin with code 32, which represents "space." (See the ***Reference Guide***, Chapter 9.)

The ASCII "characters" with codes 0 through 31 have no standard character representations. They are **control characters** that are used to control I/O devices. Each control character has a standard name and meaning, although its exact function varies from one device to another.

For example, code 13 is "carriage return." It customarily makes a device begin writing a new line of output. This code is input by the HHC's ENTER key. PRINTing it on the LCD erases the LCD and moves the cursor to the left edge. PRINTing it on a micro printer advances the printer's paper one line and moves the print head to the paper's left margin.{1}

You can send a control character to the LCD like this:

```
PRINT CHR$(N);
```

where N is the number of the control character you want to send. You will find it convenient to create string variables for the control characters you use frequently, for example:

```
CR$=CHR$(13)
PRINT "Each line displayed";CR$;"▶
   is one test result.";
```

For another example, ASCII code 7, "bell," traditionally is used to make a peripheral device emit an audible alarm. On the LCD, it creates the beep that you hear when you make an error. You can make the HHC beep by sending a 7 to the LCD like this:

```
BP$=CHR$(7)
PRINT BP$;
```

{1} - The micro printer actually accumulates enough characters to print ***two*** lines, and prints both lines at once. If the first line is ended by a carriage return, however (as opposed to being ended by overflow onto the second line), the micro printer prints the line immediately.

Notice the semi-colon after BP$. This prevents Basic from writing a carriage return to the LCD after executing the PRINT statement. A carriage return at that point would be superfluous, and might be unwelcome, because the PRINT statement was only intended to make the HHC beep; it didn't display anything!

{H} Displaying Control Characters On the LCD

Although control characters normally perform control operations on the LCD, they also have displayable representations. For example, code 13, "carriage return," which clears the LCD, is represented by an inverse-image 'M'. (Recall that the ENTER key displayed an inverse-image 'M' when you pressed it while defining a function key.)

If a particular control character has no control function on the LCD, it displays its inverse-image representation. If it has a control function on the LCD, it will perform that function.

{H} You can display any control character, instead of performing its control function, by sending the following sequence of characters to the LCD:

1. ASCII character 27, "escape."
2. ASCII character 71,'G'.
3. The control character you want to display.

For example, you can display a "carriage return" on the LCD (as an inverse-image 'M') by executing the following statement:

```
PRINT CHR$(27);"G";CHR$(13);
```

Note that this technique works on ***some*** peripheral devices, but not all! See the ***Reference Guide***, Chapter 7, for details on each peripheral.

{H} ESCAPE CONTROL SEQUENCES

The sequence "escape I enter" to put the displayable representation of "enter" on the LCD is called an **escape control sequence**. Many of the HHC's peripherals recognize escape control sequences as commands to perform various kinds of operations.

Every escape control sequence consists of three characters:

1. ASCII character 27, "escape."
2. A character called the **operation code**, or **opcode** for short, which specifies the operation this escape control sequence is to perform.
3. A character called the **data byte** which gives additional information about the operation. The meaning of the data byte depends on the value of the opcode. With some opcodes the data byte is ignored, but it always must be present.

Like control characters, escape control sequences have standardized meanings, but their functions differ on different devices.

Unlike control characters, which are used by almost all computers that use ASCII, escape control sequences have meaning only on the HHC.

Here are some further examples of escape control sequences that work with the LCD, and on many of the HHC's peripherals:

- **Set inverse mode**: subsequent characters are displayed in inverse-image form. Sequence is 'escape C x' where 'x', the data byte, is ignored.
- **Set uninverse mode**: subsequent characters are displayed in ordinary form (not inverse-image). Sequence is 'escape D x', where 'x', the data byte, is ignored.
- **Set flash mode**: subsequent characters are displayed flashing on and off. Sequence is 'escape E x' where 'x', the data byte, is ignored.
- **Set unflash mode**: subsequent characters are displayed without flashing. Sequence is 'escape F x' where 'x', the data byte, is ignored.

The tables in the ***Reference Guide***, Chapter 7, list all of the escape control sequences recognized by the HHC. Chapter 7 also describes the effect that the sequences have on each device.

If a given sequence is not recognized by a given device, the device will ignore it; that is, the device will behave as though no part of the sequence had been sent.

INDEX

NOTES

NOTES

NOTES

NOTES

FRIENDS AMIS, INC.

The program described in this document is furnished under a license and may be used, copied, and disclosed only in accordance with the terms of such license.

Friends Amis, Inc. ("FA") EXPRESSLY DISCLAIMS THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR USE FOR A PARTICULAR PURPOSE RESPECTING THE HHC SOFTWARE PROGRAM AND MANUAL. THE PROGRAM AND MANUAL ARE SOLD "AS IS". THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR USE FOR A PARTICULAR PURPOSE AS TO THE MEDIUM ON WHICH THE SOFTWARE IS RECORDED ARE LIMITED TO SIXTY (60) DAYS FROM THE DATE OF LICENSING BY THE INITIAL USER OF THE PRODUCT AND ARE NOT EXTENDED TO ANY OTHER PARTY.

USER AGREES THAT ANY LIABILITY OF FA HEREUNDER, REGARDLESS OF THE FORM OF ACTION, SHALL NOT EXCEED THE LICENSE FEE PAID BY USER TO FA. FA SHALL NOT BE LIABLE FOR INDIRECT, SPECIAL OR CONSEQUENTIAL DAMAGES, SUCH AS, BUT NOT LIMITED TO, LOSS OR INJURY TO BUSINESS, PROFITS, GOODWILL, OR FOR EXEMPLARY DAMAGES, EVEN IF FA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

FA will not honor any warranty when the product has been subjected to physical abuse or used in defective or non-compatible equipment.

The user shall be solely responsible for determining the appropriate use to be made of the program and establishing the limitations of the program in the user's own operation.

An Important note: Good data processing procedure dictates that the user test the program, run and test sample sets of data, and run the system in parallel with the system previously in use for a period of time adequate to insure that results of operation of the computer or programs are satisfactory.