USA
Panasonic Company
Division of Matsushita Electric Corporation of America
One Panasonic Way,
Secaucus, New Jersey 07094

Panasonic Hawaii Inc.
91-238 Kauhi St. Ewa Beach
P.O. Box 774
Honolulu, Hawaii 96808-0774

Panasonic Sales Company
Division of Matsushita Electric of Puerto Rico, Inc.
Ave. 65 De Infanteria, KM 9.7
Victoria Industrial Park
Carolina, Puerto Rico 00630

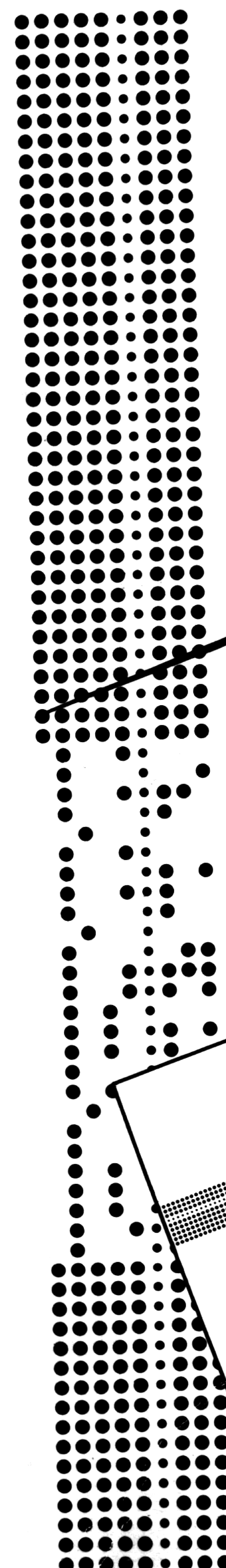
CANADA
Panasonic Canada
Division of Matsushita Electric of Canada Limited
5770 Ambler Drive, Mississaugo,
Ontario L4W2T3

OTHERS
Matsushita Electric Trading Co., Ltd.
32nd floor, World Trade Center Bldg.,
No. 4-1, Hamamatsu-Cho 2-Chome,
Minato-Ku, Tokyo 105, Japan
Tokyo Branch P.O. Box 18 Trade Center

www.pocketmuseum.com

Panasonic

HHC T.M.



Not for sale

SnapBASIC

an advanced programming language for the HHC™

VOLUME I: TUTORIAL

www.pocketmuseum.com

Not for sale

Copyright © 1982 by Matsushita Electric Industrial Co., Ltd.
All Rights Reserved. HHC™ is a Trademark of Matsushita
Electric Industrial Co., Ltd.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

WHAT IS BASIC? 1-1
WHAT CAN SnapBASIC DO? 1-1
HOW TO USE THIS BOOK 1-3

CHAPTER 2: GETTING STARTED

PREPARING THE HHC TO RUN SnapBASIC 2-1
GETTING ACQUAINTED 2-2
STARTING SnapBASIC 2-3
HOW SnapBASIC HANDLES ERRORS 2-5
THE PRINT STATEMENT 2-6
Fingers Tired? 2-7
INTERLUDE: USING THE SHIFT KEYS, AND OTHER MATTERS 2-7
The SHIFT Key 2-8
The 2nd SFT Key 2-8
NEGATIVE NUMBERS 2-9
DOING SOME CALCULATIONS 2-9
EXPRESSIONS 2-10
The Order of Arithmetic Operations 2-10
Overriding the Normal Operator Precedence 2-11
TURNING THE HHC OFF 2-12
THE AUTO-SHUTOFF FEATURE 2-13

CHAPTER 3: VARIABLES

INTRODUCTION TO VARIABLES 3-1
NAMING VARIABLES 3-2
THE INITIAL VALUE OF A VARIABLE 3-2
VARIABLES AND EXPRESSIONS 3-3
X=X+1. . .? 3-3
PERFORMING A CALCULATION IN STEPS 3-4
WHAT IS PROGRAMMING ABOUT? 3-5

CHAPTER 4: WRITING A STORED PROGRAM

WRITING A PROGRAM 4-1
Memory, Storage, and Some Other Terms 4-1
Getting Started 4-2
What Happened 4-2
Adding Statements To a Program 4-3
Limit On the Length Of a Line 4-3

www.pocketmuseum.com
Not for sale

LIST: REVIEWING THE CONTENTS
OF A PROGRAM 4-4
Listing Part Of a Program 4-5
Replacing a Line 4-5
Entering the Editor 4-5
Changing the Contents Of a Line 4-6
Moving the Cursor Right 4-6
Deleting a Line 4-7
Copying a Line 4-7
The Auto-Repeat Feature 4-8
TRICKS FOR LAZY PROGRAMMERS 4-8
More About Editing 4-9
SAVING A PROGRAM IN A FILE 4-9
On the Relation Between SnapBASIC and
the File System 4-10
VARIABLE VALUES AND RUN 4-10
THE INPUT STATEMENT 4-11
INPUT With Multiple Variables 4-11
INPUT With a Prompt 4-12
CONCLUSION 4-13

CHAPTER 5: MORE ABOUT BASIC

ERROR MESSAGES IN A BASIC PROGRAM 5-1
THE RANGE OF A NUMERIC VALUE 5-2
HOW BASIC PRINTS VERY LARGE AND
SMALL NUMBERS 5-3
MORE ABOUT THE PRINT STATEMENT 5-4
Printing Several Values On a Line 5-4
How SnapBASIC Spaces Values 5-4
Printing String Constants 5-5
How To Avoid PRINT's Zones 5-5
Combining Strings and Numbers
In a PRINT Statement 5-6
SEVERAL PRINT STATEMENTS,
ONE PRINTED LINE 5-6
The SPC$ Function 5-7
THE POS FUNCTION 5-7
THE STP/SPD KEY 5-8

CHAPTER 6: MORE ABOUT EDITING PROGRAMS

INSERTING A CHARACTER IN A LINE 6-1
Inserting Several Characters In a Line 6-2
The ← and → Keys In Insert Mode 6-2
DELETING A CHARACTER IN A LINE 6-3
EDITING A LINE LONGER THAN THE LCD 6-4
REVIEWING A LINE 6-5
THE ↑ AND ↓ KEYS 6-5
SOME ADDITIONAL EDITING OPERATIONS 6-6
SHIFTING CASE WITH THE LOCK KEY 6-6
NOTE ON EDITING IN IMMEDIATE MODE 6-6

CHAPTER 7: MANAGING PROGRAM FILES

KEEPING AN EYE ON YOUR FILES 7-1
HOW TO DELETE A FILE 7-1
WARNING: AVOID THE CLEAR KEY 7-2
HOW TO RECOVER FROM CLEAR WHILE
EDITING A BASIC PROGRAM 7-2
HOW TO RENAME A FILE 7-3
HOW TO COPY A FILE 7-3
INTRODUCING THE PROGRAMMABLE
MEMORY PERIPHERAL 7-4
Copying a File To a Programmable
Memory Peripheral 7-4
When You Run Out Of Space 7-5
Managing Files In a Programmable
Memory Peripheral 7-6
Recovering a File From a Programmable
Memory Peripheral 7-7
Programmable Memory Peripheral
Anomalies 7-7
Note On Multiple Peripherals 7-7

CHAPTER 8: REMARKS IN SnapBASIC PROGRAMS

THE REM STATEMENT 8-1
ADVANTAGES IN USING REM STATEMENTS 8-2
DISADVANTAGES IN USING REM STATEMENTS 8-2
WHAT TO WRITE IN REMARKS 8-3

CHAPTER 9: FLOW OF CONTROL

INTRODUCTION TO FLOW OF CONTROL 9-1
THE GOTO STATEMENT 9-1
ENDING EXECUTION OF THE PROGRAM 9-2
SOME GENERAL NOTES ABOUT GOTO 9-2
INTRODUCING THE IF STATEMENT 9-3
RELATIONAL OPERATORS 9-4
More Relational Operators 9-5
A Little Quiz 9-5
IF. . .THEN 9-6
Some Variations On the Program 9-7
PLANNING PROGRAMS FOR CHANGE 9-8

www.pocketmuseum.com
Not for sale

MULTIPLE TESTS IN ONE "IF" 9-9
Examples 9-10
SEVERAL STATEMENTS ON A LINE 9-10
THE ON/GOTO STATEMENT: MULTI-WAY DECISIONS 9-11

CHAPTER 10: MORE ABOUT VARIABLES

PROPERTIES OF REAL VARIABLES 10-1
INTEGER VARIABLES 10-3
BOOLEAN VARIABLES 10-5

CHAPTER 11: ARRAYS

WHAT IS AN ARRAY? 11-1
THE DIMENSION OF AN ARRAY 11-2
USES OF ARRAYS 11-2
AN EXAMPLE: CALCULATING THE NUMBER OF A DAY IN A YEAR 11-3
ANOTHER EXAMPLE: RECORDING VALUES IN ORDER 11-4
MULTI-DIMENSIONAL ARRAYS 11-5

CHAPTER 12: SOME EXAMPLES

WHERE DO THE EXAMPLES COME FROM? 12-1
THE DAY-OF-YEAR CALCULATOR 12-1
THE VALUE-ORDERING PROGRAM 12-4
SOME NOTES ON DEBUGGING 12-8
Avoiding Bugs 12-8
Eliminating Bugs 12-9
Execution Tracing Aids 12-10
The CONT Command 12-12
Finding All the Bugs 12-12
CONCLUSION 12-13

CHAPTER 13: THE FOR/NEXT STATEMENT

SOME TERMINOLOGY AND RULES 13-1
AN EXAMPLE 13-2
ABOUT THE INITIAL VALUE AND THE LIMIT 13-3
THE STEP WORD 13-3
NESTED FOR/NEXT LOOPS 13-4
THE VALUE ORDERING PROGRAM, REVISITED 13-5
SPEEDING UP LOOPS 13-6
SOME DETAILS 13-7

CHAPTER 14: MORE ABOUT I/O

THE READ AND DATA STATEMENTS 14-1
Some Rules For Using READ and DATA 14-2
The RESTORE Statement 14-3
USING PERIPHERALS 14-3
Getting Ready 14-4
Connecting the Micro Printer 14-4
Writing Information To the Printer 14-4
Attaching the Printer 14-5
Detaching Devices 14-5
Input From Peripherals 14-6
GETting from Peripherals 14-6
More About LUNs 14-6
Device Independence 14-7
LISTing On the Printer 14-7
I/O ANOMALIES 14-8
ATTACH CODES FOR VARIOUS DEVICES 14-9

CHAPTER 15: FUNCTIONS

AN EXAMPLE 15-1
SOME OTHER USEFUL FUNCTIONS 15-3
INTEGER FUNCTIONS 15-4
CONVERSION BETWEEN INTEGERS AND FLOATING POINT NUMBERS 15-4
AN IMPORTANT NOTE ABOUT FUNCTION CALLS 15-6
USER DEFINED FUNCTIONS 15-6
The Formal Parameter: Some Examples 15-6
Some Benefits Of Using Defined Functions 15-8
Some Limitations On User Defined Functions 15-8

CHAPTER 16: STRINGS

STRING VALUES 16-1
STRING VARIABLES 16-2
SOME SIMPLE STRING OPERATIONS 16-2
Assignment 16-3
The INPUT Statement 16-3
The READ Statement 16-4
String Values ***vs***. Numeric Values In INPUT and READ 16-5
CONCATENATION 16-5
STRING SUBTRACTION 16-6
THE INSERT$ FUNCTION 16-6
THE ERASE$ FUNCTION 16-7
Comparison 16-7
AN EXAMPLE: THE FUEL EFFICIENCY

www.pocketmuseum.com
Not for sale

CALCULATOR 16-7
EXAMPLE: THE DAY-OF-YEAR PROGRAM 16-8
GETTING THE LENGTH OF A STRING 16-10
COMBINING STRINGS AND NUMBERS 16-10
Converting a Number To a String 16-10
Converting a String To a Number 16-11
EXTRACTING PIECES OF STRINGS 16-11
The LEFT$ Function 16-11
The RIGHT$ Function 16-12
The MID$ Function 16-12
EXAMPLE: DAY-OF-YEAR CONVERSION
USING SUBSTRINGS 16-13
MORE EXACT FORMATTING: THE STRF$
FUNCTION 16-14
MORE ABOUT CHARACTERS 16-18
Comparing Characters 16-18
Comparing Strings 16-19
ASCII Code 16-19
Converting Characters To Numbers... 16-22
...And Back 16-22
Example: Forcing Characters To Lower Case 16-22
VAL and ASC 16-23
STR$ and CHR$ 16-23
THE GET STATEMENT 16-24
GET Reads One Character 16-24
GET Does Not Echo 16-25
SOME IDEAS FOR PRACTICE 16-25

CHAPTER 17: SUBROUTINES

A SIMPLE EXAMPLE 17-1
ANOTHER EXAMPLE: THE DIFFERENCE
BETWEEN TWO DATES 17-2
Note On Errors In Subroutines 17-4
Planning Ahead 17-4
GENERALITY *VS.* EFFICIENCY 17-5
About Line Numbers and Subroutines 17-5
What Should Go Into a Subroutine? 17-6
ERRORS THAT GOSUBS CAN CAUSE 17-7
THE ON/GOSUB STATEMENT 17-7

CHAPTER 18: PEEKS AND POKES

INTRODUCING THE PEEK FUNCTION 18-1
Note On PEEKing Two-Byte Fields 18-1
Example: Is a Device Attached To a LUN? 18-1
INTRODUCING POKE 18-2
How POKE Works 18-2
An Example: Disabling the Auto-Off Timer 18-3
CAUTION!!! 18-3
OTHER PEEKS AND POKES 18-4

CHAPTER 19: USING THE FUNCTION KEYS

DEFINING A FUNCTION KEY 19-1
EXAMPLE: A FUNCTION KEY FOR LIST 19-1
DISPLAYING A FUNCTION KEY'S DEFINITION 19-2
SPECIAL KEYS IN A FUNCTION KEY DEFINITION .. 19-2
HOW TO CORRECT A FUNCTION DEFINITION 19-3
HOW TO ERASE A FUNCTION DEFINITION 19-3
LIKELY USES FOR FUNCTION KEYS 19-3

CHAPTER 20: ADVANCED I/O TECHNIQUES

CONTROL CHARACTERS 20-1
Displaying Control Characters On the LCD 20-2
ESCAPE CONTROL SEQUENCES 20-2
SQUEAK COMMAND 20-3

CHAPTER 21: ADVANCED FILE TECHNIQUES

VARIOUS FILE TYPES 21-1
BINARY FILES 21-1
TEXT FILES 21-3
EXAMPLES 21-5
FILE ERRORS 21-5
SPECIAL FILE TECHNIQUES 21-5
SUGGESTIONS FOR USE OF FILES
IN SnapBASIC 21-6
EXAMPLE: A TEXT FILE SORTER 21-7

CHAPTER 22: THE FINAL STAGE—PREPARING A CAPSULE

HOW TO PREPARE A SnapBASIC PROGRAM
FOR BURNING 22-1
HOW TO BURN 22-3
A SAMPLE BURN 22-5

CHAPTER 23: EXCEPTION HANDLING—THE ONERR STATEMENT

ANOTHER USEFUL TRICK 23-2

www.pocketmuseum.com
Not for sale

CHAPTER 24: LOADING PROGRAMS FROM FILES AND PERIPHERALS

THE LOAD COMMAND 24-1
DOWNLOADING PROGRAMS FROM
OTHER COMPUTERS 24-2
OTHER WAYS TO DOWNLOAD 24-4

CHAPTER 1: INTRODUCTION

This book is your introduction to computer programming with SnapBASIC for your HHC[tm].

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 SnapBASIC 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]

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

WHAT CAN SnapBASIC 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 write accounting programs and similar programs for business or personal use.
- It is available on many different computers. Once you are familiar with SnapBASIC, you will find that you can learn to

www.pocketmuseum.com
Not for sale

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.

The SnapBASIC compiler/interpreter has some special features:

- While the ease of access to BASIC through the interpreter has been retained, all the code of the program is compiled, i.e. it runs much faster than if it were interpreted.
- Well tested programs can be 'BURN'ed into EPROM and ROM capsules. This allows you to duplicate your own programs for your own use or for distribution.[1][2]
- SnapBASIC provides a powerful interface between itself and the file system, allowing processing by SnapBASIC of information in text files generated with the HHC editor, as well as data base capabilities using text or binary files.

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

- Features that make large programs easy to write and change.
- Greater efficiency. Some programs written in SnapBASIC may run less efficiently than equivalent programs written in another language.

If you are interested in writing programs in another language, consider learning to program in the following language:

- SnapFORTH, a very powerful language designed for professional programmers who want to develop packaged programs. SnapFORTH was used to write much of the HHC's internal software, and most commercially available capsules.

---

[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 your HHC dealer.

[2] - Another way is to use the new HHC EPROM burner adaptor. Information on this and other HHC peripherals is available from your dealer.

Both SnapFORTH and SnapBASIC are good languages for developing new HHC capsule programs.

Your HHC dealer 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 SnapBASIC.

This book's companion volume, the ***SnapBASIC 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 both SnapBASIC 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 SnapBASIC 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 statements, functions, operators and keywords SnapBASIC recognizes.

If you are familiar with the HHC, but not with SnapBASIC, 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 SnapBASIC 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**}.

CHAPTER 2: GETTING STARTED

PREPARING THE HHC TO RUN SnapBASIC {H}

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 pulling 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 SnapBASIC 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 SnapBASIC 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. Don't worry if it already ON.

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

www.pocketmuseum.com

Not for sale

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 'BAT LOW', and the HHC turns itself off. This is a signal that you had better plug in the AC Adaptor to run the HHC and recharge the batteries.

{H} 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 to reset the HHC when you have finished a task. Never use the CLEAR key to exit SnapBASIC, except for in the most dire emergencies. If you do, you will most likely discover that your program is completely unusable when you try select it again. Use the BYE command instead. Repeat: NEVER USE CLEAR.
- 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).

  There is one more unmarked blip to the left of the SHIFT blip. This is used to indicate that SnapBASIC is busy with some or another internal task, such as 'housekeeping'. When this blip goes off, SnapBASIC continues normal operation.
- 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.

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), or connecting to the I/O Adaptor when multiple peripherals are used.

STARTING SnapBASIC {H}

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=SnapBASIC
```


www.pocketmuseum.com Not for sale

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 SnapBASIC, 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

```
SnapBASIC
```

to confirm your choice. Then it begins running the SnapBASIC interpreter.[1]

SnapBASIC 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 SnapBASIC program. The message says "New file" because when SnapBASIC 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 SnapBASIC 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

```
New 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

[1] - **Note to old HHC hands**: Now is the time to select your RAM bank, if you need to. The I/O key is disabled in SnapBASIC. Its I/O peripheral capabilities are replaced with the ATTACH and DETACH statements; the only function that is not replaced is RAM bank selection.

www.pocketmuseum.com

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 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. This will make it easier for you to remember in the future. Later we will find out how to use SnapBASIC to manipulate files, and you will see the advantages about being consistent in your use of lower case filenames.

The cursor is always over the next position where a typed character will be displayed, like the type element of a typewriter. Note that if you enter a name longer than 10 characters, the "New progam name" prompt will scroll to the left, and the cursor will remain in the rightmost position in the LCD. The name may be whatever you like; whatever characters you desire may be included. It may be up to 80 characters long, if you really want to type in that much. (After 80 characters, the HHC will beep at you.)

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

```
>
```

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

HOW SnapBASIC HANDLES ERRORS {**B**}

Before you begin learning to write correct SnapBASIC 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. SnapBASIC will list the line and display the **error message**

```
**** QWERTY <--SY-ERROR
```

This means "**Sy**ntax error", an error in the way you wrote something. SnapBASIC is telling you what you did wrong and when.

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 SnapBASIC program that will have any

Not for sale

effect more serious than this one did. (The few "dangerous" errors can happen only in rarely used parts of SnapBASIC 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 SnapBASIC, each step in a program is called a **statement**.

We're going to start programming in SnapBASIC 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, SnapBASIC clears the LCD (that always happens when you press ENTER) 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 SnapBASIC. A list of such words is contained in the ***Reference Guide***.

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
```

SnapBASIC obediently displays '15.3'. SnapBASIC 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
```

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

1. SnapBASIC processes the the number '15.300000'. SnapBASIC 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. SnapBASIC 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 SnapBASIC displays a number it always omits trailing zeros after the decimal point.

Try the following statement:

```
Print 1.000123456789
```

SnapBASIC doesn't display exactly what you entered; it rounds the number to '1.00012345679'. SnapBASIC stores numbers in a form that limits their precision to twelve decimal places. If a number can't be represented exactly in twelve decimal places, SnapBASIC rounds it to the nearest twelve-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 SnapBASIC will display each of the numbers you enter.

Fingers Tired? {**B**}

Here's a useful hint: SnapBASIC accepts '?' as an abbreviation for the word 'print'. For example, in place of 'print 229.5', you can type '? 229.5'. You will find the '?' just to the left of the SPACE bar.

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").

Not for sale

The SHIFT Key

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.[2]

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.

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

```
>PrI
```

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'. **SnapBASIC 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 chapters where we learn about situations in which SHIFT is useful.

The 2nd SFT Key

The 2nd SFT ("second shift") key produces all the characters inscribed on the HHC's keyboard in red.

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
```

[2] - 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?

www.pocketmuseum.com

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 '×' symbol (on the 'I' key[3]). 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[4]). Again, this use of '/' is almost universal.

```
Print 15/5
```

[3],[4] - The '×' and '÷' symbols are used for multiplication and division in the HHC's calculator program. In most other prgrams, including SnapBASIC, they duplicate the function of the SPACE key.

Not for sale

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*5.135
```

You can write an expression that subtracts or divides several numbers, or does any combination of operations in one 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 2^2, or 2*2*2*2.

Non-integer exponents are allowed; for example:

```
Print 2^4.3
```

displays 19.6983106135, 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.

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

If you are familiar with mathematics, you will find that SnapBASIC'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. Relational operators '>' '<' and ' = ' have the next precedence, followed by relational 'AND' and 'XOR', followed by 'OR'.
6. If two or more consecutive operations hae equal precedence, they are performed left-to-right. ***Examples***: in '5-2-1', 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 SnapBASIC's operator precedence rules, put parentheses around the part of the expression SnapBASIC is to evaluate first.

For example, consider the following statement:

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

Here is how we make SnapBASIC 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, SnapBASIC would evaluate '1/2' first, then add 12 to the quotient. ('/' has higher precedence than ' + '.) Then SnapBASIC 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 SnapBASIC will do the addition, the subtraction, and multiplication by ***pi*** (3.1415926) first, then do the other multiplication, then do the division? Try it on your HHC; does

www.pocketmuseum.com

Not for sale

your solution work? The result it should produce is 13.7509873177.

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!). What's worse, they might be too complex for SnapBASIC to handle. 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.[5] 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, jut as it was when you pressed OFF! You canb turn the HHC off at ***any*** time, and whatever operation the HHC is performing will continue undisturbed when you turn it back on.

[5] - Note the ALL OFF switch on the back of the HHC. Remember, this switch stays ON at all times.

THE AUTO-SHUTOFF FEATURE

The HHC will automatically turn itself off ten minutes after the last program input, or ten minutes after the last key is pressed. If a SnapBASIC program runs continuously with no I/O or key press for longer than ten minutes, the HHC will automatically shut off. Each character received by SnapBASIC, either in immediate mode or by the execution of an INPUT or GET statement, starts the 10 minute period all over again.

The auto-shutoff feature is designed to preserve the HHC's batteries in case you absentmindedly leave the HHC on when you are done using it. (You can disable this feature from SnapBASIC if you need to. See the chapter in this manual on PEEKS AND POKES.)

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

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 SnapBASIC. Turn your HHC on; type in the following SnapBASIC statement and press ENTER:

```
x=5
```

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

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

Now enter the following statement:

```
Print x
```

and SnapBASIC 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, SnapBASIC 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 SnapBASIC 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.)

{B} ## NAMING VARIABLES

Whenever you write a program, you must choose names for the variables you plan to use. In naming SnapBASIC 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.
3. A name may be any reasonable length; SnapBASIC will use all characters to distinguish names. This is in contrast with most BASICs that ignore all characters after the first two. Note however that longer names take more room in memory; also, you can't put more than eighty characters on a line, in any case.
4. {B} A variable name may not be or ***start*** with a reserved word, such as 'print'. This is in contrast to most Basics, that will not allow a variable to ***contain*** a reserved word anywhere within the name.

   An attempt to use an illegal variable name will cause an SY (syntax) error.

   Here are some reserved words that could appear as the beginning of your variable names if you don't guard against them:

AND	END	NEW	PI
CONT	FOR	NOT	REM
DATA	IF	ON	RUN
DEF	LET	OR	TO

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

Note that all the function words will be seen as variables unless the typical word is followed by a left paranthesis "(" or is followed by a $ sign for string functions.

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 MONTHS, or at least MN, rather than just M or (horrors) X.

{B} ## THE INITIAL VALUE OF A VARIABLE

Try PRINTing a variable you have never assigned a value to, such as DAYS. You will get 0.

Every variable that you use in a SnapBASIC 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.

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 SnapBASIC***.

In mathematics, '=' represents a ***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 SnapBASIC, '=' 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 SnapBASIC. 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 SnapBASIC, 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 SnapBASIC 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 outwards 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:

$$1+\frac{1}{X}$$

Think about what we have here. There are two SnapBASIC 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 SnapBASIC 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, and therefore to program.

After you master the "ABC's" of SnapBASIC, most of your programming time will be spent figuring out how to describe the solutions to problems so that a SnapBASIC 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.

Not for sale

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 SnapBASIC 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 SnapBASIC will ***hold in storage*** rather than ***execute***. When we have typed in several statements, we will tell SnapBASIC 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. ROM can either be permanent, or programmable, in which case it is usually called PROM (Programmable ROM) or EPROM (Erasable PROM).

We will use the term **storage** to describe RAM in which you can store, edit and run SnapBASIC 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 3,000 to approximately 7,000 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 built-in ROMs that hold the HHC's fundamental operating programs; the ROM in HHC capsules; and any memory in a peripheral device other than a Programmable Memory Peripheral.

www.pocketmuseum.com

Not for sale

Getting Started

Let's write a simple computer program on paper. This is actually something we've already done. Any two SnapBASIC 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. SnapBASIC hasn't executed either statement yet.

Next, type the command

```
run
```

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

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

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

What Happened

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

When SnapBASIC 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, SnapBASIC executes it in **immediate mode**; that is, SnapBASIC 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. SnapBASIC displays 63, then 67.

When you entered statements preceded by the line numbers 8 and 6, SnapBASIC compiled the statements of the program into memory 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 {B}

You cannot create a line in a SnapBASIC program that is more than about 80 characters long. If you try, SnapBASIC will type the 81st and subsequent characters on top of the 80th character; what's worse, they will not be accepted by SnapBASIC.

Also, because SnapBASIC does not necessarily LIST lines in the exact same way in which you entered them, there is the possibility that entering a line shorter than 80 characters may still make SnapBASIC try to LIST the line as longer than 80 characters. When this happens, the line will be listed in inverse video, and part of the line may be lost, necessitating splitting the line and restoring the lost information. This is most likely to happen if you don't enter spaces and if you say '?' for PRINT at the beginning of a line.

When a line is displayed, there may be a funny symbol at the right end of the LCD that looks like four little hyphens stacked on top of each other. This indicates that the line continues past the right end of the display. Pressing the right arrow key will bring that part into view.

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, SnapBASIC has the **LIST** command, which reconstructs a listing of the current program and lists it 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.

After a small waiting time, SnapBASIC displays the next line of your program, '8 PRINT X'. (The display speed is controlled by the STP/SPD key.) After that, SnapBASIC displays the next line, and so on, until the last line of your program has been displayed.

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

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

```
20 ? x , y , z
```

or

```
20Printx,y,z
```

SnapBASIC will display the statement as

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

This feature of SnapBASIC 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. SnapBASIC will put blanks in all the right places when it lists the statement. If you don't enter blanks, you better stay way short of the 80 character limit, since trouble will occur when SnapBASIC attempts to list the line if it is too long.

Listing Part Of a Program

You need not start listing a program at the very beginning. You can start at any line. Neither do you need to list a program until the very end. You can stop at any line.

To list a program starting at some line after the beginning, and stopping at some line before the end, enter LIST followed by the numbers of those lines, seperated by a comma. For example, to start listing your program at line 6, and stop at line 10, enter

```
list 6,10
```

Try this. After line 6 appears, line 8 will appear, followed by line 10, and then the SnapBASIC prompt }.

If you enter LIST with a line number that does not exist, SnapBASIC will start listing your program at the next larger line number, and stop at the previous smaller line number that does exist.

Once you start a listing, it will continue at a speed depending on your current speed setting. If you want to stop the listing momentarily, press the STP/SPD key again. To slow down or speed up the listing, press the STP/SPD key and then a number from 0 to 9. 1 is the slowest speed and 0 is the fastest. Finally, to abort the listing prematurely, press the BREAK key.

Replacing a Line

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

Entering the Editor

{**B**}

SnapBASIC allows you to change the contents of any line in a program. There are two ways of doing this. The first way is to type a whole new line. The other way is to use the Line Editor feature of SnapBASIC.

It is easy to enter the Editor. Just press ♠ or ♥. This will take you into the last accessed line of your program. For instance, if you had just LISTed a line, the ♠r key will take you into this line in 'edit mode'. That is to say that you can use all the usual HHC commands to delete, insert and change characters. Also you can use the ♠ or ♥ key to go to a previous or to a following line (any changes made in the line are made permanent by doing this). You can also enter the Editor by LISTing a single line.

It is also easy to exit the Editor. Just press ENTER, or the BREAK key. (The BREAK key is the one labelled C1 on the HHC keyboard. We will refer to it often.) When you press enter, the changes in the last accessed line are made permanent, but when you press BREAK the changes are not accepted and the original line, if existing , is restored. In both cases you will have exited from the Editor. BREAK is also used to interrupt a running SnapBASIC program.

{B} Changing the Contents Of a Line

To change the contents of a line, LIST the line, move the cursor right to the part you want to change, and type new characters over the characters that are already there.

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

```
6 X=63
↑
└──── cursor is here
```

Use the ➡key to move cursor right seven 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 BREAK, 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 'BREAK', that means, "I've changed my mind; I don't want to modify this line after all." Only ENTER, ⬇ , and ⬆ store 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. You can, of course, extend the line to the right by adding more characters.)

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 (i.e. line# ENTER). Alternatively, you can use the command DEL line# or DEL line#1, line#2 (first through last) (see the Reference Manual). Deleting a long range of lines, like DEL 100,2520 can take a long time. Be patient!

Copying a Line {B}

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'. SnapBASIC will display line 10 like this:

```
10 X=67
↑
└── cursor is here
```

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

```
10 X=67
↑
cursor is here
```

```
6 X=67
 ↑
 cursor is here
```


www.pocketmuseum.com

Not for sale

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 SnapBASIC accept the change you have made by pressing ENTER. LIST your program again. Is it back in its original form?

{H} **The Auto-Repeat Feature**

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.

{B} **TRICKS FOR LAZY PROGRAMMERS**

Some people get real annoyed at having to type in the line numbers before every line. What can you do about this?

SnapBASIC has a facility, called **AUTO**, which will automatically generate line numbers for you. Invoke AUTO like this:

```
AUTO n1,n2
```

where ***n1*** is the first line number you want to enter, and ***n2*** is the number you want to increment the line number by. For example,

```
AUTO 100,10
```

will generate line numbers 100,110, and so on.

Once you invoke AUTO, SnapBASIC will prompt you with the first line number you specify. Enter and edit your line in the usual way, typing ENTER when you are done. After you type ENTER, SnapBASIC will prompt you with the next line number. When you have entered all the lines you want, press BREAK, and SnapBASIC will return to immediate mode.

A couple of details: If you already have a line in the program with the same number that AUTO generates for you, it will be deleted as soon as you press ENTER ***unless*** you press ENTER immediately after the line number appears.

You can also enter a negative line increment, in which case you can enter your program backwards. (Why anyone would want to do this is a mystery, though.)

Yet another nice facility is available, called RESEQUENCE. This command looks like this:

```
RESEQ n
```

This will resequence (renumber) all the lines in your program to multiples of ***n***. For example,

```
RESEQ 100
```

will change all of your lines to 100,200,300 and so on. While the resequencing is going on, the program will be displayed. Note that GOTOs and GOSUBS get taken care of properly, with the new line numbers inserted in place of the old ones.

A note about line numbers: You are sure to make mistakes when entering your program, and will almost certainly want to insert lines in between old ones. It makes working a lot easier if you use line numbers like 100,200,300—you can then insert a whole lot of lines. If, instead, you entered 1,2,3 you would be stuck if you needed to insert a line between 1 and 2. Resequencing might take care of this, but resequencing only really works when the program is somewhat complete (because errors are generated if GOTOs have nowhere to go).

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}

SnapBASIC 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, SnapBASIC saves your program in its file system, under the name that you chose when you first

Not for sale

entered SnapBASIC. Then it returns you to the primary menu. Select SnapBASIC again. 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 SnapBASIC's menu and return to the HHC's primary menu, so that you can use some program other than the SnapBASIC interpreter.

Try pressing "2" to resume using "program 1." Do a LIST to verify that you do, indeed have your program back again. (Sometimes it takes a long time to load a program from the menu. Don't panic! Be patient.) Enter BYE again to return to the primary menu.

Pick selection 1 , if you wish, and create another program. Get some practice in writing and editing SnapBASIC programs, and try switching back and forth between your two programs. You can press CLEAR to leave the SnapBASIC menu, but it is never a good idea—indeed, it is downright dangerous to leave a selected file with CLEAR. ***Always use BYE!***

{B} On the Relation Between SnapBASIC 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 SnapBASIC program file is another type.

The SnapBASIC menu shows only SnapBASIC 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 SnapBASIC program only with the editor built into SnapBASIC. You can, however, use the file system to create a file that can then be compiled with the LOAD command. See Chapter 24.

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, SnapBASIC sets all the variables to their initial value of 0 before executing the first statement.

This habit of SnapBASIC 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, and leaves the message:

```
Rest Ignored
```

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

```
Error, Retype line
```

and prompts you again with a '?' after the error statement and starts reading the first variable in again.

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 now INPUT does not add a '?' to the end of the prompt. Nor does it append a space.

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 SnapBASIC 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,

www.pocketmuseum.com

above. Note the ';' that separates the string constant from the list of variables; you must use a ';' here, not a ',', or SnapBASIC 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 SnapBASIC 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 SnapBASIC 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 their 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.

Not for sale

CHAPTER 5: MORE ABOUT BASIC

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

ERROR MESSAGES IN A BASIC PROGRAM {B}

You have already encountered error messages. When you entered an invalid statement in Chapter 2, you got an error message that said "SY 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 SnapBASIC finds an error while executing a program in deferred mode. Turn on your HHC and enter SnapBASIC, if necessary; select "program 1" from the SnapBASIC 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 SnapBASIC to accept this "statement," SnapBASIC objects and gives you the error message

```
10 X=& <--CH-ERROR
```

The error code, CH, tells you what happened.

The line number, '10', reminds you which line you were trying to compile in your program when the error happened. (This happens in exactly the same way as when you were entering statements in immediate mode, because the error was always in the statement you had just entered; however 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:

SY **_Syntax Error_**. A SnapBASIC statement is incorrectly written: parentheses do not match, an operator is used in an improper context, a reserved word is misspelled, etc.

IQ **_Illegal Quantity._** The result of a calculation is too large to be represented in SnapBASIC'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.)

Division by zero also gives this error.

CO **_Command Error_**. You have tried to use a statement in immediate mode that is only legal in deferred mode; or vice-versa.

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.)

AE **_Arithmetic Error._** A standard function call is incorrect, or an overflow in in a floating point or integer number occurs.

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

For a complete list of SnapBASIC's error codes, see the **_Reference Guide_**, Chapter 10.

THE RANGE OF A NUMERIC VALUE

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

The greatest positive number that SnapBASIC can represent is about $9.99999999999*10^{1023}$ (that is, 999999999999 followed by 1012 zeroes). The greatest negative number that SnapBASIC can represent is the negative of this $-9.99999999999*10^{1023}$.

There is also a limit to the tiniest ***absolute*** value that SnapBASIC can represent: about $1.00000*10^{-1024}$ (1 preceded by a decimal point and 1023 zeroes). The tiniest negative value that SnapBASIC can represent is the negative of that: $-1.00000*10^{-1023}$. If a calculation results in a number smaller than this, it will become zero.

HOW BASIC PRINTS VERY LARGE AND SMALL NUMBERS

SnapBASIC 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?

SnapBASIC 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 SnapBASIC 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 SnapBASIC program. You can also use it to enter any numeric value in response to an INPUT statement;

you may use both an upper case 'E' or a lower case 'e' before the exponent.

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
```

SnapBASIC 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
```

{B} How SnapBASIC Spaces Values

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

1. It divides the line into **zones**, each 22 characters wide.
2. It displays the first value beginning at the left edge of the first zone: that is, in LCD position 0.
3. It skips to the beginning of the next zone.
4. It displays the second value beginning at the left edge of the second zone, and so forth until all the values have been displayed.

Note that SnapBASIC differs from other BASIC systems in that it does not automatically print spaces before and after numbers. This is because of the limited size of the LCD display.

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.
```

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 "Your fuel efficiency is "
   ,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, scrolling across the LCD screen, something like this:

```
Your fuel efficiency is     27.5031
  128123          mpg.
```

Why did this happen? Remember how SnapBASIC divides a line of output into 22-character zones! If we printed this program's output and marked the divisions between zones, the reason for the odd spacing would become clear:

```
Your fuel efficiency is        27.5031128123
```

1st zone   2nd zone   3rd zone

The first value displayed by line 30, 'Your fuel efficiency is', 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. For the same reason, PRINT puts the third value, 'mpg.', in the fourth zone, off the end of this page.

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 'Your fuel efficiency is 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 "Your fuel efficiency is ";
    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:

```
Your fuel efficiency is 27.5031128123 mpg.
```

first value — second value — third 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.

SnapBASIC 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, SnapBASIC 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 "Your fuel efficiency is ";
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 "Your fuel efficiency is ";
40 PRINT MP;
50 PRINT " mpg.";
60 PRINT
```

The SPC$ Function

SPC$ makes PRINT display a specified number of spaces in a line of output. It may be used anywhere in a string expression. For example, consider the following statement:

```
PRINT "Your fuel efficiency is";
    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 '#' for 'space' so that you can see how many spaces there are):

```
PRINT "Your fuel efficiency is
    ########";MP
```

THE POS FUNCTION {B}

The POS function is a way of determining the POSition of the current character in the LCD. It is useful for placing your output precisely where you want it to be. It looks like this:

```
POS(0)
```

and returns a number (the same way a variable does). Don't worry about the (0); just use it. It is a dummy parameter, and you will learn more about such things in the chapter on Functions. POS is particularly useful when you are outputting data to a printer, or to a multi-line device like the TV Adaptor, instead of the LCD. POS makes it easy for you to align a column of numbers exactly where you want them. For example, the following sequence of statements:

```
40 PRINT "Start reading =";
42 PRINT SPC$(15-POS(0)); SR;
    SPC$(27-POS(0));"miles"
```

```
50 PRINT "End reading =";
52 PRINT SPC$(15-POS(0)); ER;
   SPC$(27-POS(0));"miles"
60 PRINT "Gallons used =";
62 PRINT SPC$(15-POS(0)); GA
70 PRINT "Your fuel efficiency is ";
72 PRINT SPC$(15-POS(0)); MP;" mpg"
```

will display output looking like this:

```
Start reading =  35505    miles
End reading =    37707    miles
Gallons used =      22
Your fuel efficiency is
    100.090909091 mpg
```

The item SPC$(X-POS(0)) puts output at position X by figuring out the current position (with POS), subtracting it from X to determine how many spaces are needed, and then generating the spaces with SPC$. Note that the fourth line of the output is not aligned; SPC$ with a negative argument does not generate anything.

{B} 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.

It is possible to get a speed faster than that of the 0 key, but you must issue a special command (POKE) to get it. See the section in this manual on PEEKS AND POKES.

CHAPTER 6: MORE ABOUT EDITING PROGRAMS

So far you've learned how to insert or delete a line in a SnapBASIC 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 SnapBASIC, 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

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.

SnapBASIC 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 SnapBASIC, with a line number before it, so that SnapBASIC 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 only to one character at a time; the next character you type will replace the '*' unless you press INSERT again.

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 SnapBASIC, "start inserting characters, and keep on inserting them until I tell you to stop."

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

Let's try this. Enter the line above into SnapBASIC (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 SnapBASIC 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.

DELETING A CHARACTER IN A LINE

{B}

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 SnapBASIC 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 SnapBASIC 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 BREAK (C1) as you learned to do when we first introduced LIST, and

Not for sale

SnapBASIC will leave this statement unchanged. (List it again to assure yourself of that.) When you are done entering a line, you must press ENTER or the ▼ and ▲ keys to make SnapBASIC accept the changes you have made. Pressing BREAK will always make SnapBASIC 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. SnapBASIC'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 SnapBASIC 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, SnapBASIC 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.)

The time may come when you LIST a line, and it comes up in inverse video (that is, white characters on a black background). This means that the line you entered, when LISTed, is longer than 80 characters. This probably happened because you used '?' instead of PRINT, and the rest of the line came close to the 80 character limit. This means that the end of your line has been lost! You need to recover from this by splitting your line in two parts. You should avoid this unpleasant occurrance by keeping your lines shorter, especially when you are using '?'.

REVIEWING A LINE {B}

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

SnapBASIC 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 SnapBASIC stop rotating, press any key. The right arrow is the best key to use here, since it will not change your line at all.

THE ▲ AND ▼ KEYS {H}

When SnapBASIC is in immediate mode , the ▲ and ▼ keys initiate EDIT mode, and show the last accessed line. You can now edit the line using all the edit functions described here.

When SnapBASIC is in edit mode, the ▲ and ▼ keys allow you to go to a previous line or to a next line, just as you are accustomed to in using the HHC. Before the commands are executed, SnapBASIC will recompile the whole line you have just been editing, and will give an error message in case of an error. If an error is found, the original line is not changed, and any original line is again available for editting. But remember, once a line has been accepted, its changes are permanent.

SnapBASIC will remember the last line you were editing; if you press one of these arrow keys from the } prompt, you will be shown the last line you were editing.

If you are viewing the last line in your program, pressing the down-arrow key will return you to the SnapBASIC prompt, the down-arrow key will return you to the SnapBASIC prompt, indicating that there is no more program. Pressing the down-arrow again will get you to the first line in the program. Similarly, an up-arrow from the first line will get you to the SnapBASIC prompt; hitting it again will get you to the last line in your program.

To exit from edit mode, press the RETURN key . Any changes in the line you have been editting will have been made permanent when you typed the ▲, ▼ or RETURN keys.

When you typed the BREAK (C1) key, SnapBASIC will exit the ***edit*** mode and will not update the current line.

SOME ADDITIONAL EDITING OPERATIONS

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

- ***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.

A complete chart of SnapBASIC editing functions is contained in the ***Reference Guide***.

{H}
SHIFTING CASE WITH THE LOCK KEY

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 select a program, or you will get the message 'OM error' when you try to extend or 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
{B}

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 SnapBASIC menu.

You can tell roughly how much file space you have left by going to the SnapBASIC 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, and to leave SnapBASIC).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.

---

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

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

```
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 SnapBASIC 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.

WARNING: AVOID THE CLEAR KEY

Pressing the CLEAR key quickly twice in succession can be disasterous. DON'T DO IT! Always use the BYE command to leave BASIC. If for some reason you must use the CLEAR key to exit BASIC, press it once, wait a second, and then press it again. You should then re-enter the same BASIC file and exit with BYE.

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

If you should happen to press CLEAR while running or editing a SnapBASIC program, the HHC will act as if its file storage space were almost full.

The amount of free memory for the HHC depends also on FREE : the amount of free memory in SnapBASIC.

FREE (HHC) = Total (HHC) - BASIC (program)
- BASIC (variables)
- BASIC (free)

To recover from this condition, do the following:

1. Return to the SnapBASIC menu.
2. 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 accidentally 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.

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. Remember to press the ENTER key when you are done.

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 SnapBASIC 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. (This will only occur if you have a Programmable Memory Peripheral.)

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.

{H} INTRODUCING THE PROGRAMMABLE MEMORY PERIPHERAL

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 to 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 SnapBASIC programs. The Programmable Memory Peripheral, like the HHC's intrinsic RAM, can hold SnapBASIC 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 SnapBASIC programming.

Save your SnapBASIC 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.[2]

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
```

(The exact numbers you see in this menu will depend on the amount of intrinsic RAM in your HHC, the amount of RAM in your PMP, and any RAM currently being used for existing files.)

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.)

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 SnapBASIC) 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.

---

[2] - **Be sure to save your SnapBASIC program** before plugging in the PMP! Plugging in a peripheral has the same effect on the HHC as pressing CLEAR. Recall that pressing CLEAR while you are editing a program will have you lose the last unfinished updates of the line you are working on. Plugging in a peripheral has the same effects.

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.

{H} Managing Files In a Programmable Memory Peripheral

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-video characters.

To select a new current memory area, press the I/O key. 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 SnapBASIC, you can edit them directly with SnapBASIC's program editor.

Recovering a File From a Programmable Memory Peripheral {H}

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".

Programmable Memory Peripheral Anomalies

When the PMP is selected and the BASIC interpreter is entered, it is still necessary for BASIC to use a certain amount of intrinsic RAM. It is possible that not enough intrinsic RAM is available; if this happens, BASIC will automatically select intrinsic RAM as the current RAM and enter the NO ROOM, DELETE FILE menu. At this point you must free some intrinsic RAM if you wish to continue using SnapBASIC.

Note On Multiple Peripherals {H}

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.

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 SnapBASIC, 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 SnapBASIC PROGRAMS

THE REM STATEMENT

As you begin to write more complex programs in SnapBASIC, 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 SnapBASIC 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
```

Note: SnapBASIC does not insert a space between REM and the remark, so that if you enter

```
REM remark
```

LIST will show you

```
REM remark
```

(in small letters, as you typed them)

If you enter

```
REMremark
```

LIST will show you

```
REMREMARK
```

(in capital letters)

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.
```


www.pocketmuseum.com

Not for sale

```
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.

Important note: If you are going to BURN your program, you must be more careful where you put your REM statements. BURN removes all REM statements, so that optimum use is made of memory space. But there is a catch: if a GOTO or a GOSUB refers to a REM line, there will be no place to go come burn time! Hence, if a program is to be BURNED, do not ever GOSUB or GOTO a REM statement.

SnapBASIC does not squeeze out blanks from REM statements, nor does it translate letters to upper case (except in the word REM itself, and in any word immediately following and connected to REM). 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 SnapBASIC, this kind of information should be kept in a seperate 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 "Your fuel efficiency is ";
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 "Your fuel efficiency is ";
90 PRINT (ER-SR)/GA;
100 PRINT " mpg."
110 GOTO 60
```

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

www.pocketmuseum.com

Not for sale

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). SnapBASIC responds with the message

```
Break in line #60
```

where "60" is the line number of the line SnapBASIC 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.

We always refer to the C1 key as the BREAK key in this manual.

You can also interrupt your program's execution by pressing the CLEAR key ***once*** (this takes you back to the Main Menu). The BREAK key is preferable, however. Form the habit of never using the CLEAR key in SnapBASIC unless the BREAK key does not work for some reason. If you do accidently use CLEAR, immediately return to the same SnapBASIC program, and use BYE to exit it properly.

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:

```
GO ERROR in line #110
```

which is short for "**GO**to Statement error". The program has no statement with the line number 11.

www.pocketmuseum.com

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 "Your fuel efficiency is ";
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 "Your fuel efficiency is ";
```


Not for sale

```
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 SnapBASIC executes line 140, SnapBASIC goes to line 60. If CT is not 1, SnapBASIC 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.

RELATIONAL OPERATORS

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

{B} 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 SnapBASIC 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

SnapBASIC 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 SnapBASIC 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*** the 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

SnapBASIC 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*** SnapBASIC 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]

```
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 "Your fuel efficiency is ";
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
```

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

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 "Your fuel efficiency is ";
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...?"

SnapBASIC 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***.

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

true XOR ***true***	results in ***false***
true XOR ***false***	results in ***true***
false XOR ***true***	results in ***true***
false XOR ***false***	results in ***false***

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

The fourth logical operator in SnapBASIC 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 operators,

---

[2] - Logical operators actually operate on Boolean variables (see Chapter 10, "More about Variables".) Any non-zero value is considered to be TRUE, while only zero itself is FALSE.

NOT has the highest priority,
the relational operators have the next priority,
AND has the next priority, and
XOR and OR have the lowest priority.

Note: NOT has however a higher priority than the 6 relational operators, necessitating the use of parentheses in situations like NOT (A>B) to first evaluate A>B, and then NOT.

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

Flow of control 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

Flow of control 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

Flow of control 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

Flow of control does not go to line 45.

{B} SEVERAL STATEMENTS ON A LINE

Up to now you have seen every SnapBASIC 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, than 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 a little space in the program file (to be exact, 4 bytes.)
- 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. What's more, when LIST expands the statements, they might well exceed the 80 character limit.

Also, there is a limit on how complex a line is allowed to be. If there are too many nested parentheses, or some other such situation, a "CX error" (complexity error) will result. This must be fixed by making the line simpler.

In any case, do not try to make a program line more than 80 characters long. SnapBASIC will not accept a longer line as input.

THE ON/GOTO STATEMENT: MULTI-WAY DECISIONS

SnapBASIC has another decision making statement, the ON/GOTO statement, that is useful when you want to execute

Not for sale

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: MORE ABOUT VARIABLES

Up until this point, we have not concerned ourselves with the properties of the numbers that we have been using. All of the numbers we have used have been ***real*** numbers. There are, however, other types of variables in SnapBASIC, and there are advantages to using each type where it is appropriate.

PROPERTIES OF REAL VARIABLES

In Chapter 5, we talked about the range of a variable. We said that a variable must be smaller (in magnitude) than 1.0E1024 (a quite large number—thats a 1 with 1024 zeroes after it, bigger than a googol) and that the tiniest number that could be represented is 1.0E-1024. These limits are true for what we call **real** variables. A real number in computer talk is not the same as a real number in mathematics, though. You might recall from high school mathematics that a real number is just about any number from minus to plus infinity, and that there are an infinite number of them (and that between any two real numbers there is another real number).

A real number in computer talk is quite a different thing. Every number is represented by a bit pattern; to indicate an infinite amount of real numbers would require an infinite amount of bits (something that neither the HHC nor any other computer is capable of). Rather, the term **real number** refers to the set of numbers that can be represented as a thirteen-digit mantissa multiplied by by a 10-bit exponent. (A real number consumes 8 bytes total. Each byte of mantissa has two BCD digits. The two extra bits are sign bits for the mantissa and the exponent). As such, a total of about 2.048E16 values are expressable—a plenty large number, but nowhere near infinity.

What are the ramifications of this? For one, most of the numbers are not stored exactly. Try this:

```
?9.999E100 + 1
```

The result is 9.999E100. What happened to the + 1? Vanished. As far as SnapBASIC is concerned, the smallest number that can be added to 9.999E100 to have any effect at all is .00000000001E100.

The only place this is likely to cause problems is if you are attempting to do exact calculations where there is an intermediate result with a very large magnitude. It is to your advantage (if you are interested in precision) to keep your numbers as small as possible. As an example, the formula for

Not for sale

determining the number of ways n items can be selected from a collection of m items is

$$C = (m! / (n! * (m-n)!))$$

where n! (read "n factorial") is equal to n * (n-1) * ... * 1. N! grows very quickly; for 1-10 it is

N	1	2	3	4	5	6	7	8	9	10
N!	1	2	6	24	120	720	5040	40320	362880	3628800

So try this program.

```
10 INPUT N,M
20 IF M < N THEN PRINT "No good.
   Try again.";:GOTO 10
30 K = M : GOSUB 1000 : MF = KF
40 K = N : GOSUB 1000 : NF = KF
50 K = M-N : GOSUB 1000 : MNF = KF
60 PRINT "The result is ";MF /
   (NF * MNF)
70 GOTO 10

1000 REM A subroutine to calculate
     the factorial of K.
1010 REM Input is K; output is KF.
1020 KF = 1
1030 IF K < 2 THEN RETURN
1040 FOR I = 2 TO K
1050 KF = KF * I-
1060 NEXT I
1070 RETURN
```

(Don't worry about what the subroutine means right now—or look in Chapter 17 to get an idea.) Now RUN the program. Note that if you try the values m = 458 and n = 229, you get the result 2.77337295706E136; but if you try m = 459, n = 229, you get the message

```
IQ ERROR in line 1050
```

What's going on? That E136 number is well below E1023; the next value should not be much bigger. But in line 1050, where we are calculating the factorial, we are trying to calculate 459!, which is greater than 1.0E1024—out of range.

So what can we do? Well, a little bit of examination reveals that the expression C = M!/(N! * (M-N)!) can be modified to

$$C = \frac{m * (m-1) * ... *(n+1)}{(m-n)!}$$

This will eliminate the very large number m!. Here's another program that does the same thing, then:

```
10 INPUT "M and N, please:";M,N
20 IF M<N THEN 10
30 MF = 1
40 FOR I = M TO N+1 STEP -1
50 MF = MF * I
60 NEXT I
70 MNF = 1
80 FOR I = 2 TO M-N
90 MNF = MNF * I
100 NEXT I
110 PRINT "The result is ";MF / MNF;
120 GOTO 10
```

Now you can try the larger values: for m = 459, n = 229, you get the result 5.53468777084E136. In fact, you can use numbers clear up to and including 747 and 374 before the IQ error shows up.

So lesson number 1 about real numbers is: try to design your formulas to keep numbers as small as possible. It works the other way, also: numbers smaller than E-1023 can't be expressed, so be careful when using tiny numbers as well.

INTEGER VARIABLES

{**B**}

You might very well have no need for floating point numbers. If all of your calculations are going to be integers—that is, numbers with no fractional part—and your numbers are never going to be larger in magnitude than 32767, you have an alternative. SnapBASIC provides **integer variables**. An integer variable can hold only integers.

You create an integer variable by putting a '%' sign after a variable name. For example, HE% is the name of an integer variable. The naming conventions that apply to real variables apply to integer variables also.

You can operate on integer variables in the same way that you operate upon 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 mix integer and real variables freely within a statement:

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

When SnapBASIC evaluates an expression containing integers, it checks whether any conversions to real form are required. If so, SnapBASIC converts all of the integer values to real values; or, it might convert the real values to integers. At any rate, some conversion will happen. This conversion is necessary any time any real variable shows up in an expression with integer variables.

Integer arithmetic is considerably faster than real arithmetic. The statement

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

will operate considerably faster than

```
X = X + Y
```

This advantage, however, only exists when integers and reals are not mixed in expressions: as soon as a single real value comes into the expression, the entire expression is evaluated as real, and the speed advantage vanishes.

Integer variables have another advantage: they eat up less of your storage space. Memory in your computer is measured in units called "bytes"; a byte is the amount of memory required to store one character of text data. A real variable requires 8 bytes, while an integer variable requires only 2 bytes.

SnapBASIC provides a number of functions that are specifically designed for integers. Among these functions are:

- **MOD%(A%,B%)**—returns the remainder of A% / B%
- **MIN%(A%,B%)**—returns the lesser of A% and B%
- **MAX%(A%,B%)**—returns the greater of A% and B%

Three other integer functions are available that are designed for operating upon specific bits within integers. These are:

- **BAND(A%,B%)** returns the bitwise AND of A% and B%.
- **BOR(A%,B%)** returns the bitwise OR of A% and B%.
- **BXOR(A%,B%)** returns the bitwise XOR of A% and B%.

For more information on how these functions operate, see the ***Reference Guide***.

www.pocketmuseum.com

Integer arrays are also possible. To create one, just say

```
DIM XX%(20)
```

(for example). An array of integers results in a substantial space savings.

BOOLEAN VARIABLES

SnapBASIC provides yet another type of variable: the **Boolean variable**. The name "Boolean" comes from George Boole, a famous logician. Boolean variables may have only two values: **TRUE** and **FALSE**. TRUE has the value of 1; FALSE has the value of 0. They take up only one byte of storage.

You create a Boolean variable the same way as you create any other variable, but you add a '?' to the end. For example, VALID?, X?, and LEGAL? are all Boolean variables.

Why would you want to use a Boolean variable? Assume you have a complicated expression like

```
IF (A=B) AND ((C=D) OR (X=Y)) THEN ...
```

that you need to test for several times within your program. It would be a waste of both time and space to have to include the same line over and over again. Instead, you could have one line in your program that said

```
WELL? = (A=B) AND ((C=D) OR (X=Y))
```

and then, when you need to test the expression, you could say

```
IF WELL? THEN ...
```

Boolean variables have their own special operations. These are the same as the "relations" that you can include in an IF statement. For example,

```
B? = A? AND C?
Q? = NOT C?
A? = B? XOR C?
```

are all pure Boolean operations.

As with integers, Booleans can be interspersed with other data types in expressions. If this is done, the value FALSE will be converted to the number 0, while TRUE will be converted to 1. Also, reals and integers can be forced to Booleans (as in the case of

```
B? = A AND C?
```


Not for sale

In this case, A will be converted to Boolean (since AND requires a Boolean operand). Any non-zero value is converted to TRUE; only zero is converted to FALSE.

When entering Boolean data with the INPUT statement, SnapBASIC uses any string starting with 0, F (for false), or N (for NO) as FALSE, and any other value is determined to be TRUE.

Boolean arrays are also possible; define a Boolean array as

```
DIM BOOL?(10)
```

for example. Boolean arrays are the most space-efficient array.

A good use of Boolean variables is when space is at a premium, and there are a lot of flags in your program that can either be on or off. For example, you might do this:

```
10 REM Definition of FLAG?
20 REM Bit 0: Is it time for lunch?
30 REM Bit 1: Is there food?
40 REM Bit 2: Is the stove on?
50 REM Bit 3: Are the dishes washed?
60 REM and other useful flags...
70 IF BAND(FLAG?,1) GOTO 1000
80 REM ...not time for lunch
        :
1000 IF NOT BAND(FLAG?,8) THEN PRINT
     "Wash dishes!"
1010 IF BAND(FLAG?,2) THEN PRINT
     "Have some food!"
```

Of course, you would probably do something more useful. But this should give you the general idea.

Note that though there is a substantial space savings this way, there is also a substantial time penalty. This is the usual trade-off in optimizing: time vs. space.

www.pocketmuseum.com

CHAPTER 11: ARRAYS

WHAT IS AN ARRAY?

Up to now we have dealt with variables that have a single value. SnapBASIC 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**. You may remember from algebra that members of the array A are referred to as A_1, A_2, and so on. There is no way to really write a subscript in SnapBASIC, so instead we write an expression like this:

```
A(5)
```

where the (5) is the subscript.

An array reference (that is, an array name followed by a subscript) may occur almost anywhere a normal variable may be used.

```
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.

SnapBASIC 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. SnapBASIC 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.)

Not for sale

THE DIMENSION OF AN ARRAY

The **dimension** of an array defines the number of elements in the array. If an array has a dimension n, the array contains n+1 elements. The first element of an array is always element 0. For example, an array with the dimension of 53 contains 54 elements numbered from 0 through 54.

To create an array we use the **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 array contains 54 elements subscripted from #0 to #53.

You ***must*** create an array with a DIM statement before you try to reference it. If you do not, you will get a UD (undefined dimension) error. What's more, you may ***not*** redimension an array by using the DIM statement a second time; if you do, you will get an RD (redefining dimension) error.

Note that some BASICs assume a default dimension of 10 for arrays. This is ***not*** true for SnapBASIC: all arrays must be explicitly dimensioned.

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(11)
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

www.pocketmuseum.com

Not for sale

(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(9)
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 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.

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-DIMENSIONAL ARRAYS

{**B**}

SnapBASIC 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(4,5)
```

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(4,5): 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)
R	1	RA(1,0)	RA(1,1)	RA(1,2)	RA(1,3)	RA(1,4)	RA(1,5)
O	2	RA(2,0)	RA(2,1)	RA(2,2)	RA(2,3)	RA(2,4)	RA(2,5)
W							
S	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 ten elements per dimension, the array would contain 100,000 elements, and this is considerably more than the HHC can manage, even with the largest-capacity Programmable Memory Peripheral. The HHC has a limit of 13 dimensions for arrays; any more dimensions than this will cause a CX (complexity) error.

CHAPTER 12: 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 SnapBASIC 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 at least 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. (Note that DIM MA(12) actually reserves 13 places. Don't worry about it.)

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 at least 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 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 SnapBASIC 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, inquire 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.

{{B}

Execution Tracing Aids

If your program goes into an endless loop, remember that the BREAK 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.

To help you find where and why a program may have gone wrong, SnapBASIC has two debugging facilities : the HISTORY and the Trace facilities.

SnapBASIC'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 SnapBASIC 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, interspersed with the normal program output. 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 SnapBASIC 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 TROFF or BYE.

As an example, you might wish to know when a certain branch is being executed in your program. So you could do this:

```
100 TRON
110 IF A < B THEN 300
120 TROFF
```

When SnapBASIC reaches line 100, tracing will be enabled. You would know if the branch was taken by the following display:

```
#110 #300 ...
```

but the display would be different if the branch was not taken:

```
#110 #120
```


at which point tracing would be disabled again.

SnapBASIC's **History facility** is also a valuable debugging tool. The trace facility consists of two statements, HISTORY ON and HISTORY OFF. {B}

The **HISTORY ON** Statement looks like this:

```
HISTORY ON
```

When HISTORY ON is in effect, any error will turn on SnapBASIC's history reporting facility. When the error occurs, first SnapBASIC will print out the usual error message. It will then print out some more numbers, possibly. Try the following program:

```
10 HISTORY ON
20 FOR I=1 TO 10
30 FOR J=1 TO 10
40 PRINT I/0
50 NEXT J
60 NEXT I
```

Of course, there will be an IQ error at line 40, since we are dividing by 0. But with the history on, the display will show

```
****IQ ERROR in line #40 #30 #20
```

The two other line numbers are the pending FOR/NEXT loops. Similarly, if there are any pending GOSUBS, the line number that the GOSUB was called from will be displayed.

The **HISTORY OFF** statement looks like this:

```
HISTORY [OFF]
```

It disables the error-time display enabled by HISTORY ON. (Note that the OFF is optional; HISTORY by itself will turn the history OFF.)

HISTORY ON and HISTORY OFF may be used in immediate or deferred mode. Once used, HISTORY ON is effective until the next BYE or HISTORY OFF.

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.)

Not for sale

The CONT Command

You can restart a program after halting it with the BREAK 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, nor after a fatal error has occurred.

{**B**} You can use CONT after your program executes a STOP statement, as well as after you press BREAK (**STOP** is like END, except that it prints the line number of the statement your program STOPs in.) STOP is a useful debugging aid, since you can insert a STOP in your program, and then examine the contents of your variables in immediate mode, and the CONT if you want to (after possibly changing some of the variables' values.)

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 again, 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.)

Note that RUN clears variables. If you don't want to reset your variables, you can type

```
GOTO 2040
```

in immediate mode, and execution will continue.

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 SnapBASIC, and as you think of more sophisticated programs that you would like to write, you willl 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 13: THE FOR/NEXT STATEMENT

Here is a kind of loop that appears quite often in SnapBASIC programs:

```
100 I=0
.
.
180 I=I+1
190 IF I<10 GOTO 110
```

SnapBASIC 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
```

SnapBASIC 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.

SnapBASIC 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 and 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 {B}

In a FOR/NEXT loop like this,

```
FOR I=0 TO 10
.
.
NEXT I
```

"I" is called the **index**. It may be any numeric variable.

"0", the first value assigned to the index, is called the **initial value**. It may be a constant, variable, or expression.

www.pocketmuseum.com Not for sale

"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

```
NX ERROR in line#..
```

meaning, "**NE**X**T** without FOR".

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 such a loop with IFs and GOTOs 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;
```

[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 SnapBASIC 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. Also, note that the above procedure will not change the number of times the loop is executed.


```
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
```

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
```

SnapBASIC 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 SnapBASIC executes the loop one time.

NESTED FOR/NEXT LOOPS

One FOR/NEXT loop can be **nested** inside another. A nested loop looks like this:

```
90 DIM XY(9,9)
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,

```
290 DIM XY(10,6)
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,

```
290 DIM XY(10,6)
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, SnapBASIC would execute the two nested loops properly, since "NEXT I" ends both the "FOR I" loop and the "FOR J" loop. (However, the value of J would stay J = 0). Then SnapBASIC would try to execute line 340, and would display the message

```
NX ERROR in line #340
```

because the NEXT statement in line 340 does not match any FOR/NEXT loop currently being executed. ('NX' 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
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
```

{B} ## SPEEDING UP LOOPS

If you need to speed up a loop, you can in some cases use the ***integer loop*** facility that is available in SnapBASIC. By using integer variables and integer loop indices, it is possible to gain speed because the integer arithmetic is much faster than the general (i.e. floating point) numeric arithmetic. An example of the earlier mentioned loop program is given here to show what you can do to speed up a loop. However, integer values can only be used when their range (0-32767) is sufficient for the program you want to use. Example:

```
290 DIM XY%(10,6)
300 FOR I%=0 TO 10
310 FOR J%=0 TO 6
320 XY%(I,J)=0
330 NEXT J%
340 NEXT I%
```

SOME DETAILS {B}

The variable name following the NEXT may be omitted. If there is no variable name, then the NEXT applies to the most recent FOR. But be careful. Look at the following example:

```
10 FOR I=1 TO 10
20 FOR J=1 TO 10
30 IF J=5 GOTO 50
40 NEXT
50 PRINT J
60 NEXT
```

This will work just fine, until the branch in line 30 is taken. At this point, the most recent FOR is in line 20, so the NEXT in line 60 is actually NEXT J, not NEXT I as seems logical.

Another oddity is the relationship between FOR/NEXT loops and GOSUBS (which are introduced in Chapter 17). NEXT and RETURN are processed about the same way. If SnapBASIC is working on a GOSUB and sees a NEXT, it has the smarts to execute a RETURN and treat it as if it were a NEXT. It is singularly bad programming to write a program that works this way; but if your program is acting strangely, this could be the reason.

You may use FOR/NEXT in immediate mode, but only if you type in one line containing the FOR and the NEXT (with colons in between). For example, this line is quite valid in immediate mode:

```
FOR I=1 TO 1000:PRINT I:NEXT I
```

Yet another oddity is what happens when you BREAK from a loop and then execute a CLEAR, followed by CONTINUE. Try this:

```
10 FOR I=1 TO 10
20 PRINT I
30 NEXT I
```

Enter this program, then RUN it. Press BREAK after 4 numbers have been printed. Now execute the CLEAR command (which sets all variables to 0). Then execute the CONT command. See what happens? The numbers 0 through 5 are printed! Why? Well, the number of times the

loop is going to execute is determined when the loop is started, and cannot be altered. So even though you have changed the value of I with the CLEAR, the loop is still going to executed a total of 10 times. Thus, the result.

CHAPTER 14: 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, SnapBASIC'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 document easier and are 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

```
DA ERROR
```

for "Out of **DA**ta 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

{**H**}{**B**}

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.

Not for sale

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.

{H}{B} Writing Information To the Printer

You write information to the printer from a SnapBASIC 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,MA(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.

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. Each peripheral device has its own attach code; see Chapter 7 in the ***Reference Guide***. (It doesn't matter where the micro printer is plugged in; ATTACH will find it, so long as it's plugged in somewhere. ATTACH will take care of turning the printer on for you. It does not matter whether the I/O key showed the device to be ON or OFF before you entered SnapBASIC. When you exit SnapBASIC, the I/O key will reflect the most recent setting from within SnapBASIC.)

www.pocketmuseum.com

"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 {H}{B}

Enter SnapBASIC 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
```

SnapBASIC 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 no longer in effect.

Enter BYE to return to the primary menu. Re-enter SnapBASIC, re-select the program, and run it. Observe that your PRINT #2,17 is ***not*** effective now. You get the error message:

```
IO ERROR
```

This is an **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.

Detaching Devices

What goes up must come down. We have learned how to ATTACH a device to a logical unit number, and how to do I/O to that LUN. Suppose we are through with that device and we would like to use that LUN for something else now. Well, all we have to do to break the association of the peripheral from the LUN is to DETACH it. Suppose we had ATTACHed the printer to LUN #2. To DETACH it we would type:

```
DETACH #2
```


Not for sale

This will leave the device turned on, but remove the association of the printer to LUN #2. If we now PRINT to LUN #2 we will get an I/O error, since no device is currently attached to LUN #2. However we are free at this point to ATTACH another device to LUN #2 and continue.

{H}{B} Input From 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 ATTACH both in SnapBASIC.

{H}{B} GETting from Peripherals

Just as you can INPUT from peripherals, you can use the GET command to fetch single character input from peripherals. The command format is similar:

```
GET #lun,sv
```

All of the considerations for GETting from the keyboard also apply to GETting from an arbitrary LUN; for example, the character is not echoed, and some special keys are not decoded (such as BREAK).

{B} More About LUNs

SnapBASIC 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 BYE.

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, SnapBASIC 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 or GET with no LUN, SnapBASIC assumes that you are referring to LUN #0. Thus, INPUT or GET with no LUN normally reads data from the keyboard, as you would expect.

Device Independence {H}{B}

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 {H}{B}

One very useful application of a printer is printing listings of a SnapBASIC 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.

[1] - The micro printer is not well suited to this application because of its 15-character line length. Many SnapBASIC 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.

Not for sale

After entering 'LIST #2' and pressing ENTER, the printer will list your program, exactly as you would have expected.

If you want to start printing a list at some point after the first line of your program, you may specify a beginning and an ending line number with LIST:

```
LIST #2,50,100
```

In the same way you can list only one line of your program as:

```
LIST #2,50
```

But remember: this also enters the edit mode. To get out: press 'RETURN' key.

This, however, will put you in edit mode, so press ENTER to get back to the SnapBASIC prompt.

I/O ANOMALIES

The whole area of Peripheral I/O and logical unit numbers has several subtleties associated with it. For example, we told you that the HHC keyboard is attached to logical unit number 0. Suppose you used the RS-232 peripheral and attached the RS-232 input to LUN #0. Now all of the input for BASIC would have to come from the RS-232, even the BREAK character. The HHC keyboard is more or less out of commission.

Even worse than that, suppose we mistakenly attached the printer to LUN #0. If we then start printing (i.e. writing to the keyboard), we will really mess things up, and the only way we can recover is to press the CLEAR key and re-enter the file. This is always dangerous.

This phenomenon is not unique to the printer. In fact, attaching any output device to LUN #0, and then performing output to LUN #0 will cause this to happen. Fortunately there isn't really any legitimate reason for doing output to the keyboard, is there?

ATTACH CODES FOR VARIOUS DEVICES {H}{B}

Here is a list of some of the devices you can attach to the HHC, and their corresponding ATTACH codes. (Note that the ATTACH codes may actually correspond to the capsules, not to the hardware; hence, if the Telecomputing 2 capsule is installed in the RS-232 adaptor, the ATTACH codes are 130 and 66.)

Device	***Input code***	***Output code***
Keyboard	129	N/A
LCD	N/A	65
Telecomputing series	130	66
TV Adaptor	N/A	67
Micro Printer	N/A	68
RS-232 Capsule	134	70
Plotter	N/A	76

(N/A means this device is not available for input or output.)

CHAPTER 15: 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 **int**eger value that is equal 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.

www.pocketmuseum.com

Not for sale

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 {B}

Here are several useful functions that SnapBASIC supports. Each of them requires a numeric argument. If one of these functions is given an integer argument, it automatically converts the argument to real form.

- **ABS(X)** returns the absolute value of its argument: X if X>=0, -1*X if X < 0. Example: To determine the magnitude of the difference between two numbers X and Y, use ABS(X-Y).
- **EXP(X)** returns the constant E (approximated in SnapBASIC by the value 2.71828182846) raised to the power X. The maximum value of X that will not produce an overflow error is approximately 2357.7.
- **FREE(0)** 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. FREE(X) opens up memory space; for details see the Reference manual. {**B**}{**H**}
- **INT(X)** returns the largest integer less than or equal to X.
- **LN(X)** returns the the natural (base E) log of X. To obtain the base Y log of X, use the formula LN(X)/LN(Y). ***Example***: the base 2 log of 7 is LN(7)/LN(2).
- **LOG(X)** returns the the common (base 10) log of X.
- **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**}
- **SQRT(X)** returns the square root of X. Equivalent to X^.5. Causes an "AE ERROR" ("arithmetic error") if X<0.

  SQRT(X) produces the same result as X^.5, but executes more quickly.
- **SQR(X)** returns the square of X. Equivalent to X^2. SQR(X) produces the same result as X^2, but executes more quickly.

The following trigonometric functions are also supported in SnapBASIC. For details, see the Reference Manual.

- **SIN(X)**
- **COS(X)**
- **TAN(X)**

Also, the constant **PI** (3.14159265359) is defined. PI can be inserted into your program as if you had defined a variable named PI with the value of pi assigned to it. This is the only such value defined in SnapBASIC.

INTEGER FUNCTIONS

SnapBASIC supports a number of functions that operate upon integer arguments and return integer values as results. Thus, their operation is considerably faster than their corresponding real functions.

If an integer function is given a real argument, that value will be truncated to an integer before the function is evaluated. Note that this causes a loss in speed.

The following integer functions are identical to their real counterparts:

- **ABS%(n%)**
- **MAX%(n%,m%)**
- **MIN%(n%,m%)**

The function **MOD%(n%,m%)** is identical to MOD(n,m) for positive values of ***m*** and ***n***. However, while MOD(n,m) will return a negative result if and only if ***m*** is negative, MOD%(n%,m%) will return a negative result if and only if ***n%*** is negative.

SnapBASIC also supports four functions that perform bitwise logical operations upon integers. These functions are useful for creating specific bit patterns (for example, to toggle certain bit flags within HHC memory, or to optimize memory usage by using bit flags within SnapBASIC). These operations are:

- **BAND(m%,n%)** returns the bitwise "AND" of ***m%*** and ***n%***. For example, BAND(5,3) is 1, since the binary representation of 5 is 0101, while that of 3 is 0011.
- **BOR(m%,n%)** returns the bitwise "OR" of ***m%*** and ***n%***. Example: BOR(5,3) is 7.
- **BXOR(m%,n%)** returns the bitwise exclusive "OR" of ***m%*** and ***n%***. Example: BXOR(5,3) is 6.

{B} CONVERSION BETWEEN INTEGERS AND FLOATING POINT NUMBERS

There are several functions in SnapBASIC that are similar, but different. Understanding the relationship between these functions can help you to write effective and efficient programs.

First, look at the definition of INT. You will see that INT(X) always returns the largest integer that is less than or equal to X. (This is called the FLOOR function.) For positive numbers, this does about what you would expect. But INT does something not as nice for negative numbers—note that INT(-1.1) is -2. Makes sense, though: -2 is indeed the largest

integer less than or equal to -1.1. This may well not be what you are looking for. What if you really want the integer part of the number, before the decimal point? There are a few ways to do this. You could always put statements in your program like

```
100 REM Calculate integer
    closest to zero of X
110 A = INT(X)
120 IF (A=X) OR (X>0) THEN 140
130 A = INT(X+1)
140 REM and so on
```

But this is messy. Inelegant. Inefficient. There is a way to do it more efficiently:

```
100 A = FIX(X)
```

Whoa! What is FIX? Simple: FIX(X) deletes the fractional part of X. Some BASICs do not automatically convert between real and integers.

Another function that does the same thing, as it turns out, is FLOAT(X%). FLOAT, by definition, converts the integer X% to a floating point number. But! Remember that if a function requires an integer argument, a real argument will be converted first to an integer: implicitly, FLOAT(X) = FLOAT(INT(X)) if X is real.

Now, there are some interesting tricks you can do with FLOAT and its friends. It is sometimes annoying that SnapBASIC prints numbers to so many decimal places. If you are dealing with currency, you really don't want more than two places after the decimal place. Look at this statement:

```
PRINT FIX(X*100)/100
```

Neat! Multiply X by 100 to get the cents in front of the decimal point. Truncate by FIX, and divide by 100 to get the cents after the decimal point. But: this is not quite precise. You will lose fractions of cents. Doing this to 100.009 will leave 100.00—and a penny is gone. What do we do? Another trick!

```
PRINT ROUND(X*100)/100
```

ROUND is like FIX, but is rounds to the nearest integer rather than truncating the fractional part.

These methods can be use to shorten the precision of a number to whatever you want. Another way to do the same thing, but much more flexibly, is with the STRF$ function, which will be described in the chapter in this volume on strings.

AN IMPORTANT NOTE ABOUT FUNCTION CALLS

There is a strict limit as to the depth of function calls. Specifically, you may not nest function calls more than three deep, or a CX (complexity) error will occur. For example, this is illegal:

```
PRINT FIX(ROUND(SQRT(ABS(C))))
```

because the nesting is too many levels deep.

{B} USER DEFINED FUNCTIONS

SnapBASIC 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 FNAB(X)=X*(X+1)
```

"FN" is a reserved word meaning "function." It must always be present in a DEF statement. 'DEF' is a reserved word meaning DEFine, and only occurs in this context.

"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.

www.pocketmuseum.com

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 FNAB(X)=X*(X+1)
320 M = 5
330 PRINT M;"  ";FNAB(M)
340 X = 9
350 PRINT X;"  ";FNAB(X-2)
```

Line 310 defines the function.

In line 320, variable M is assigned the value 5.

Line 330 calls FNAB with the argument M. The value of M is 5; therefore, SnapBASIC substitutes 5 for X, the formal parameter of FNAB, wherever X appears in the body of FNAB. Then it evaluates FNAB:

```
FNAB(X)=X*(X+1)
```

evaluates to

```
FNAB(5)=5*(5+1)
```

which evaluates to

```
FNAB(5)=30
```

Thus the call to FNAB returns the value 30, and line 330 displays "5   30."

Line 340 assigns X the value 9. Line 350 calls FNAB with the argument X-2. The value of X-2 is 7; therefore, SnapBASIC substitutes 7 for X, the formal parameter of FNAB, wherever X appears in the body of FNAB. Then it evaluates FNAB:

```
FNAB(X)=X*(X+1)
```

evaluates to

```
FNAB(7)=7*(7+1)
```

which evaluates to

```
FNAB(7)=56
```

Thus the call to FNAB 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 FNAB is not a variable, and has no relation to the variable named X which is used elsewhere in the program.

Not for sale

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, nor is entirely different but happens to look the same.

{B} Some Limitations On User Defined Functions

SnapBASIC imposes the following restrictions on a user-defined function:

- A user-defined function can have more than one formal parameter, and each call to it must have exactly the right number of arguments. (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.)

Warning:

- Redefining a function is allowed in the program. You can start off with FNA(y) = Y * Y, and later on in the same program rename it as FNA(y) = 3 * Y. Though this is legal, the old definition will no longer be accessible. It is always best to use new names (unless space is at an absolute premium).

CHAPTER 16: STRINGS

Some of the most interesting programs you can write with SnapBASIC 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 128 distinct characters. The 128 characters include all the characters you can enter through the HHC keyboard and see on the LCD in SnapBASIC. (Note that the ***Reference Guide*** refers to character codes greater than 127. Though these exist on the HHC, they cannot be included in strings, printed, or input. However, GET can indeed sometimes fetch characters with values greater than 127, but the "garbage collector" (a SnapBASIC subsystem that conserves memory) will lose the most significant bits. Hence, do not expect to be able to input characters greater than 127 with any reliability.

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,

[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.

Not for sale

just as 'S' and 'Q' are. This makes string values (and particularly string constants) an exception to the general rule of SnapBASIC that everything is stored in upper case.

Note that strings are of variable length; operations you do on strings can increase or decrease the size of the strings. This is called "dynamic string allocation", and differs from some other versions of BASIC (which require strings to be of fixed length).

STRING VARIABLES

SnapBASIC 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$
```

LETTER$ would be an ***invalid*** name for a string variable, because it starts with a SnapBASIC reserved word, LET.

As with integer and numeric variables, you can give string and numeric variables similar names like NA and NA$ without confusing SnapBASIC; 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.

SnapBASIC 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$
QT$ = SPC$(10)
```

The INPUT Statement

INPUT and READ work the same way with string values as they do with numeric values. For example, if SnapBASIC executes the following statement:

```
INPUT "What shape? ";SH$
```

SnapBASIC 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$.

Strings may be INPUTted with or without surrounding quotes. There are certain considerations you must understand when deciding how to enter a string. If the string to be inputted does not contain any commas, and does not have leading spaces, the string may be entered without quotes.

If the string is to be entered with leading spaces, it must be contained in quotes; otherwise, SnapBASIC will strip the leading blanks. Embedded blanks may be entered with or without quotes, however.

If a string is entered enclosed in quotes, and it is to contain quotes, ***two*** quotes must be entered where each quote is wanted. For example:

```
"This is one quote:"" and this is
   two:"""""
```

There is one other character that you have to be careful with when using 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:

```
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#rounded#
    corners"
```

INPUT would assign SH$ the value:
'square,#with#rounded#corners',

In the above example we see that you can give INPUT a comma as part of a string value by enclosing the ***whole*** string value in quotation marks.

However, if we execute this statement:

```
INPUT "What shape? ";SH$
```

and respond to its prompt like this ('#' stands for "space"):

```
What shape square,#with#rounded#
    corners
```

INPUT would assign SH$ the value 'square'.

'#with#rounded#corners', the second value in our response, would simply be discarded with message 'Rest Ignored', since there is no second variable to assign it to. SnapBASIC would display the message 'Rest Ignored' to indicate that the extra data has been ignored.

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, SnapBASIC 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 SnapBASIC treats letters in a DATA statement the same way that it treats letters in an INPUT statement. For example,

```
500 data january,february,march
```

will be stored like this,


```
500 DATA january,february,march
```

but

```
500 datajanuary,february,march
```

will be stored like this:

```
500 DATAJANUARY,february,march
```

because the first string is not seperated from the DATA statement, just as with the REM statement.

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, SnapBASIC gives you the prompt

```
Error, Retype line
```

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, SnapBASIC gives you the error message

```
DE ERROR in line #nnn
```

where "nnn" is the line number of the READ statement that was trying to read from a data statement.

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'.

SnapBASIC 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 an expression with a numeric value. It may be used any place a string value may be used, for example:

```
PRINT AK$+X$+AK$
```

{B} STRING SUBTRACTION

String subtraction is the operation of deleting one string from another string. For example, if we string subtract 'ABC' from 'ABCDEF', we get the string value 'DEF'.

SnapBASIC lets you subtract one string from another with the '-' operator. Just like concatenation, string subtraction may be used anywhere a string value may appear. Note that only the first instance of the second string is deleted. For example,

```
X$ = "ABCDEFABC" - "ABC"
```

results in

```
X$ = "DEFABC"
```

If the first string does not include the string to be deleted, then no deletion happens, and the first string is unchanged. For example,

```
"ABC" - "DEF"
```

leaves the result 'ABC'.

{B} THE INSERT$ FUNCTION

Concatenation is fine if you want to join two strings end to end. But what if you want to stick one string in the middle of another? The function **INSERT** does this for you. It looks like this:

```
A$ = INSERT$(B$,C$,N)
```

where B$ is the original string, C$ is the string to be inserted, and N is the position where the inserted string is to go.

For example,

```
PRINT INSERT("ABC","D",1)
```

results in 'ADBC'.

If N is less than zero, it is forced to 0 (that is, the characters are inserted at the beginning of the string). If N is greater than the length of the original string, the characters are inserted at the end of the string.

THE ERASE$ FUNCTION {B}

Another thing you can do with a string is to delete characters from it. Use the ERASE$ function for this. It looks like this:

```
A$ = ERASE$(B$,N,M)
```

where B$ is the original string, N is the position from which to delete characters, and M is the number of characters to delete. For example,

```
PRINT ERASE$("ABCDE",3,2)
```

results in

```
ABE
```

If N is greater than the length of the string, no characters are deleted. Obviously, if you try to delete past the end of the string, you only delete to the end of the string.

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 'Array#is#full!'.

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$="more?" response.
60 INPUT "Start reading: ";SR
62 INPUT "End reading this time: ";ER
70 INPUT "Gallons used: ";GA
80 PRINT "Fuel efficiency =";
85 MP = (ER-SR)/GA
90 PRINT MP;
100 PRINT " mpg."
110 SU = SU+MP
120 CO = CO+1
122 SR = ER
140 INPUT "More input (y/n)?";YN$
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). Also, we've changed our variables to integers, for time and space; similarly, we are using MOD% for the leap year test.

```
110 REM Convert a date to day-of-year.
120 REM Input: month, day, & year.
    'end' means it.
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$(12),MA%(12)
200 FOR I%=0 TO 12
210 READ MO$(I%),MA%(I%)
220 NEXT I%
230 REM Prompt user for month, day,
    year.
240 INPUT "Month,day,year: ";
    MN$,DM%,YR%
252 FOR I%=0 TO 12
254 IF MN$=MO$(I%)GOTO 280
256 NEXT I%
258 PRINT "Month name invalid!":
    GOTO 240
260 REM Calculate DR%.
280 IF I%=12 THEN END
285 DR%=MA%(I%)+DM%
290 REM Allow for leap years.
300 IF I%<=2 GOTO 350
310 IF MOD%(YR%,400)=0 GOTO 340
320 IF MOD%(YR%,100)=0 GOTO 350
330 IF MOD%(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,end,0
```

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 252 through 256). 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 258. 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 258 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.

SnapBASIC 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:

```
AS ERROR
```

meaning, "**AS**ignment error."

Converting a Number To a String

To convert a number to a string, use the STR$ function. (Every SnapBASIC 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.

For example, to convert a number of cents into a string, and then convert it into dollars and cents, with a dollar sign in front:

```
10 A$ = STR$(CENTS)-"."
20 IF CENTS < 10 THEN A$ = "0"+A$
30 IF CENTS < 100 THEN A$ = "0"+A$
40 A$ = INSERT$(A$,".",LEN(A$)-2)
50 A$ = "$"+A$
```

Note that line 10 removes the trailing decimal point; line 20 inserts a 0 in front if we have less than 10 cents; line 30 does it again if we have less than a dollar; line 40 inserts a decimal point two from the end; and line 50 puts in the leading dollar sign.

See below for the STRF$ function, a more flexible version of STR$.

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**}

If A$ is the null string, VAL(A$) causes an IQ error.

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**. SnapBASIC 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 string consisting of the N leftmost characters of X$. X$ must be a string; N must be a number.

For example, consider this statement:

```
PRINT LEFT$("ABCDEFG",4)
```

This statement displays 'ABCD', the leftmost four characters of 'ABCDEFG'.

N, the second parameter of LEFT$, may be zero. In this case, LEFT$ 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 is less than zero, LEFT$ returns the entire string except for the rightmost ABS(N) characters. Example: LEFT$("abc",-1) returns 'ab'.

If N is greater than 255, it is forced to 255.

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.

If N is negative, RIGHT$ returns the entire string except for the leftmost -N characters.

If N is greater than 255, it is forced to 255.

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 is negative, MID$ converts it to 0.

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.

If N < 0, MID$ returns the entire string except for the -N characters starting at position P. For example,

```
PRINT MID$("ABCDEF",3,-2)
```

displays 'ABEF'. Note that MID$(S$,N1,-N2) is the same as ERASE$(S$,N1,N2).

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.
    'end' means it.
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$,
    DM%,YR%
250 IF MN$="end"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 MOD%(YR%,400)=0 GOTO 340
320 IF MOD%(YR%,100)=0 GOTO 350
330 IF MOD%(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
```

{B} MORE EXACT FORMATTING: THE STRF$ FUNCTION

SnapBASIC provides a more flexible method of converting real numbers to strings: the STRF$ function. This function (the most complicated in SnapBASIC) allows you to specify the format of the output string quite exactly.

The format of the STRF$ function is:

```
A$ = STRF(N,W,L,H,P,R,F,E)
```

That's right, ***eight*** parameters. Look at the function description in the Reference Manual for the definition of these parameters. What sort of formatting can we do with this? Let's try a few examples.

www.pocketmuseum.com

First, an example dealing with money. Say we want to print out cash values in the usual format (that is, with only two digits representing 'cents' following the decimal point). Thus, we don't want any exponential notation, ever (this will limit us to $9,999,999,999.99—if you have a net worth of greater than ten billion dollars, you can probably afford a much fancier computer than the HHC, anyway). We want the result to the closest cent. Just for this example, lets assume we aren't going to print out anything greater than $9,999,999.99.

So: first of all, the total width of the field. With the ten million dollar limit, the field (include the decimal point!) is 10 places. Thus, W = 10. (We aren't going to get any commas, incidentally.)

The length of the fraction is two digits. So, F = 2.

We never want exponential notation. So set H and L to extreme values: H = 99, L = -99. Set E to 0: it won't matter, anyway, since L is so low. Same with P.

We want to round the cents place: this is the 10^{-2} position, so R = -2.

With all these values, try this program:

```
10 INPUT N
20 PRINT STRF$(N,10,-99,99,0,-2,2,0);
30 GOTO 10
```

Now enter a few values. Note that the output is what you would expect for values like 50, 100, 20.01, and so on. See what happens when you enter .0001:

```
U.00
```

This means that the number was to small to represent with any precision at all (due to the rounding factor). To make sure this does not happen, add the line

```
15 IF ABS(N) < .01 THEN N = 0
```

Now enter a big number: 99E99. See that the result is a bunch of asterisks? This means that the number was too large to fit into the field.

Now let's make it a bit more complicated. Say we want to print a dollar sign before the value. What happens when we do this?

```
10 INPUT N
20 A$ = STRF$(N,10,-99,99,0,-2,2,0)
30 PRINT "$"+A$;
40 GOTO 10
```

Try it. Notice there are a bunch of blanks between the dollar sign and the value. Not too good: this could be especially painful if the program were printing checks (someone could

Not for sale

enter some numbers in between the dollar sign and the real value, and you could be out a million bucks!)

So we need a way to get a "floating" dollar sign, that will occur before the first digit. How can we do this?

Another function to the rescue! The SEARCH function is able to search for the n-th occurrance of a character within a string. So first, count the number of blanks like this:

```
30 I = 1
40 IF SEARCH(A$," ",I) THEN I =
   I+1:GOTO 40
```

We are being a bit sneaky here. The SEARCH function returns the position of the i-th occurrance of a blank within A$. We don't have to say '<> 0', since the IF test will return TRUE for any non-zero value, anyway.

```
50 A$ = INSERT$(A$,"$",I-1)
60 PRINT A$
```

When we get to line 50, I contains the position of the first non-blank. This is why we subtract 1. The INSERT$ function puts the second string into the first at the indicated position.

Try it!

Oh—one more thing. We want to maintain the length of A$ as 10 spaces, so the decimal points will be lined up when we print out several numbers. Do that like this:

```
55 A$ = A$ - " "
```

which deletes one blank (the one we replaced with a "$").

What else can we do? What if we want commas inserted where the number is large? There are a few ways to do this. We know that position 8 contains the decimal point, so we might try this:

```
60 IF N >= 1000 THEN A$ =
   INSERT$(A$,",",5) - " "
70 IF N >= 1000000 THEN A$ =
   INSERT$(A$,",",2) - " "
```

(again, the subtractions are to get rid of superfluous blanks). But we want a more general solution to the problem. Try this, then:

```
56 REM Insert commas. Is there
   a decimal point?
60 C=SEARCH(A$,".",1)
70 IF C < 4 THEN 120:REM No
   decimal point, or too short
80 FOR I=C-4 TO 1 STEP -3
90 IF ABS%(CHAR(A$,I)-ASC("5")) >
   5 THEN 120
100 A$=INSERT$(A$,",",I)
110 NEXT I
120 ...
```

Line 90 is a nice sneaky way to test if a character in a string is within a range of characters, as long as the range is of even length. In this case, the middle of the range is "5". That is to say, line 90 tests if the i-th character in S$ is a digit.

This routine will work for absolutely any string! Line 80 returns if the string is too short to have any commas at all. If there is stuff in the string besides digits, they will (maybe) be caught by line 90, if they are in the right place.

Here, then, is the final program, which accepts a number as input, and prints out the result with leading dollar sign and commas inserted in the right places. We've inserted a few comments for clarity.

```
10 INPUT N
20 A$ = STRF$(N,10,-99,99,0,-2,2,0)
30 I=1
35 REM Insert leading dollar sign
40 IF SEARCH(A$," ",I) THEN I=I+1:
   GOTO 40
50 A$=INSERT$(A$,"$",I-1)
55 A$=A$-" "
56 REM Insert commas. Is there a
   decimal point?
60 C=SEARCH(A$,".",1)
70 IF C < 4 THEN 120:REM No decimal
   point, or too short
80 FOR I=C-4 TO 1 STEP -3
90 IF ABS%(CHAR(A$,I)-ASC("5")) >
   5 THEN 120
100 A$=INSERT$(A$,",",I)
110 NEXT I
120 PRINT A$;
130 GOTO 10
```

Time for another example of STRF$. Let's say that we are engineers, and thus are really partial to our results looking like nanoseconds, and picofermies, and things like that. This is a good place to use the power-increment field of STRF$. Try the following example:

```
10 INPUT N
20 PRINT STRF$(N,16,0,0,1,-12,6,5);
30 GOTO 10
```

Now enter a bunch of numbers. Note that they are all printed in exponential notation, with one digit to the left of the decimal point. (Exception: E00 is never displayed.)

Now change line 20 to

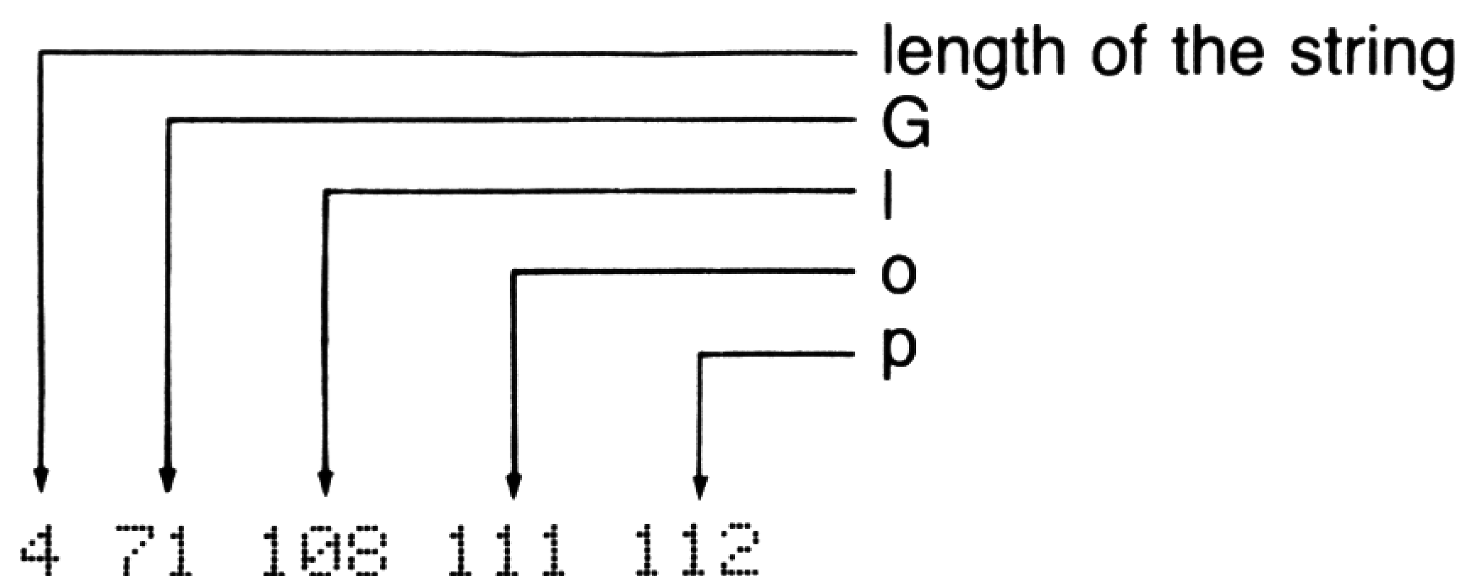
```
20 PRINT STRF$(N,16,0,0,3,-12,6,5)
```

and run the program. You will see that the exponents are always multiples of 3, the number we have specified for the power increment.

{B} MORE ABOUT CHARACTERS

Inside your HHC, each character is actually stored as a number between 0 and 127. SnapBASIC 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:



Note that string values are represented by numbers between 0 and 127, yet we all know that a byte can represent numbers between 0 and 255. It is important not to store values greater than 127 inside of string variables and arrays. Doing so will cause the "garbage collection process" in SnapBASIC to fail. Thus remember, only store ASCII characters inside of string variable and arrays.

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 (values 97 through 122)	>	Every upper case letter (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, 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.)

Not for sale

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	@	32	space	64	@
1	A	33	!	65	A
2	B	34	"	66	B
3	C	35	#	67	C
4	D	36	$	68	D
5	E	37	%	69	E
6	F	38	&	70	F
7	none[2]	39	'	71	G
8	none[2]	40	(	72	H
9	I	41	)	73	I
10	J	42	*	74	J
11	K	43	+	75	K
12	L	44	,	76	L
13	none[2]	45	-	77	M
14	N	46	.	78	N
15	O	47	/	79	O
16	P	48	0	80	P
17	Q	49	1	81	Q
18	R	50	2	82	R
19	S	51	3	83	S
20	T	52	4	84	T
21	U	53	5	85	U
22	V	54	6	86	V
23	W	55	7	87	W
24	X	56	8	88	X
25	Y	57	9	89	Y
26	Z	58	:	90	Z
27	none[2]	59	;	91	[
28	\	60	<	92	\
29	]	61	=	93	]
30	^	62	>	94	^
31	←	63	?	95	←

[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
96	`	112	p	128	↑
97	a	113	q	129	←
98	b	114	r	130	→
99	c	115	s	131	↓
100	d	116	t	132	A
101	e	117	u	133	P
102	f	118	v	134	M
103	g	119	w	135	÷
104	h	120	x	136	×
105	i	121	y	137	█
106	j	122	z	138	▯
107	k	123	{	139	ä
108	l	124	|	140	ö
109	m	125	}	141	ü
110	n	126	~	142	ñ
111	o	127	▒		

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")

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")
```

This statement converts two one-character strings, "g" and "G", to numeric form, subtracts one from the other, and assigns the difference to N.[3]

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 "IQ 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.

The argument to CHR$ is taken mod 256.

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:

[3] - 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.


```
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$>="A" AND C$<="Z"THEN
    C$=CHR$(ASC(C$)+32)
580 OU$=OU$+C$
590 NEXT I
```

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 SnapBASIC 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 to M$ the string '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 the string '2' to N$.

{B} THE GET STATEMENT

SnapBASIC 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$
```

wher 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 is 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 string containing the character that is entered.

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 SnapBASIC considers equivalent to ENTER in most contexts, such as the SEARCH key. 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 SnapBASIC might have for it.

www.pocketmuseum.com

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.[4]

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, and so on).
- 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.

---

[4] - But be careful; the C1 key is your tool for halting tle 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.

Not for sale

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? ...

CHAPTER 17: 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:

```
5 DIM A(5)
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 5 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 dependent on the HHC and the number and type of the FOR/NEXT loops in actual use when the GOSUB is activated).

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$,DM%
1080 GOSUB 2000
1090 PRINT "Difference is ";DR%-D1%;
     " days."
1100 INPUT "Again?";YN?
1110 IF YN? GOTO 1030
1120 END
1150 REM
2000 REM Subroutine to calculate
     day of year.
2010 REM In: YR%=year, MN$=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
```

www.pocketmuseum.com

Not for sale

```
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%<=2 THEN RETURN
2120 IF MOD%(YR%,400)=0 GOTO 2150
2130 IF MOD%(YR%,100)=0
     THEN RETURN
2140 IF MOD%(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 must often 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:

```
RT ERROR
```

meaning "**Ret**urn without Gosub error."

If you try to nest GOSUBs more deeply than Basic allows, you will get the message

```
CX error
```

meaning, "**C**omple**x**ity 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 18: PEEKS AND POKES

INTRODUCING THE PEEK FUNCTION {B}

In normal use, SnapBASIC only allows you to examine parts of memory that you have established through variables, and there is no way you can control exactly where in memory a variable is located. However, there are times that you might want to look directly at specific locations in memory. For this, you can use the **PEEK** function.

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 ***least*** significant, and the ***second*** byte is the ***most*** 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	1

You could PEEK and reconstruct the value like this:

```
VL=PEEK(1000)+256*PEEK (1001)
```

Example: Is a Device Attached To a LUN? {H}{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
```

{B}

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 SnapBASIC interpreter are stored.

SnapBASIC has a very powerful statement, **POKE**, that lets you manipulate the HHC's memory in this way.

Say, for example, you have a program that will run unattended for a long time. Problem is, the HHC has an auto-off timer that will power down the HHC after 10 minutes.

You can disable the auto-off timer by POKEing a number into a location that tells the HHC not to turn itself off.

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 -32767 and 32767 that identifies a particular byte. Addresses in the HHC actually go from 0 to 65535; but because of the way integers are handled, it is necessary to use negative numbers for values greater than or equal to 32768. Specifically, -1 corresponds to absolute address 65535; -2 is 65534, and so on.

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: Disabling the Auto-Off Timer

{B}

The HHC controls the Auto-Off Timer by a byte at location 101. You may disable the Auto-Off Timer by changing one of the bits in this byte; you may also periodically make the HHC think that a key has been pressed by setting one of the bits in this byte.

To completely disable the Auto-Off Timer, execute this POKE:

```
POKE 101, BAND(PEEK(101),127)
```

To turn it back on,

```
POKE 101,BOR(PEEK(101),128)
```

And to make the HHC think a key has been hit,

```
POKE 101,BOR(PEEK(101),64)
```

The BAND and BOR's are necessary to keep from changing any of the other bits in address 101, which should really be left alone.

CAUTION!!!

{B}

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.

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. This is what has happened if the message "RESTART" shows up on the LCD...a catastrophe, since every RAM file in the system is gone.
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.

OTHER PEEKS AND POKES

For a list of useful PEEKs and POKEs on the HHC, see the ***Reference Guide***, Chapter 8.

CHAPTER 19: USING THE FUNCTION KEYS {H}

The HHC has three special keys, called **function keys**, that are labelled **f1**, **f2**, and **f3**.

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 {H}

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 a non- blinking underscore cursor to the right of the definition.
3. Enter the string of keystrokes you want this function key to represent. This string may include any key except ON, OFF, a 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 same function key you are defining (or 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.

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.

{H} 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 F1, press HELP, then CLEAR. When you are done, press CLEAR.

Note that pressing CLEAR will return you to the SnapBASIC menu! Re-enter the same file you were editing previously.

{H} 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-video** '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 (inverse)	ROTATE	G (inverse)	C1	[illegible]
►	↓	I/O	K (inverse)	INSERT	A (dotted)	C2	[illegible]
▲	←	STP/SPD	N (inverse)	DELETE	E (dotted)	C3	[illegible]
▼	→	SEARCH	■	ENTER	M (inverse)	C4	[illegible]

You cannot put the following special keys into a function key definition: another function key, ON, OFF, LOCK, CLEAR, SHIFT, and 2nd SFT. (But SHIFT and 2nd SFT have their usual effects on other keys.)

HOW TO CORRECT A FUNCTION DEFINITION {H}

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 the original function key.
2. Press HELP, then the appropriate function key, as though you are reviewing the definition.
3. Start entering the correct definition. When you enter the first keystroke, the old definition disappears from the LCD, to be replaced by the one you are entering.
4. To end the new definition, press the same function key.

HOW TO ERASE A FUNCTION DEFINITION {H}

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

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.
- The BYE command.
- 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 again at a particular point in your work; for example, a sequence of keystrokes to perform an editing operation that you must apply to many lines in your program.
- Frequently used responses to INPUT or GET statements. You can use the function keys at run time as well.

A hint: It is possible that a function key definition can be convenient, but dangerous. As we were preparing this book, we had one function key for RUN, and another for the

www.pocketmuseum.com

Not for sale

commands NEW:AUTO 10,10. Well, great: the first time we pressed the wrong button, the entire example we were working on got erased. So a word of caution: dangerous sequences probably should not end with ENTER.

CHAPTER 20: ADVANCED I/O TECHNIQUES

In this chapter we will cover some ways of doing sophisticated I/O operations with SnapBASIC.

CONTROL CHARACTERS {B}

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 an 7 to the LCD like this:

```
BP$=CHR$(8)
PRINT BP$;
```

Notice the semi-colon after BP$. This prevents SnapBASIC from writing a carriage return to the LCD after executing the PRINT statement. A carriage return at that point would be

---

[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.

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.

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 73, 'I'.
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$(27)+"I"+CHR$(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. (They may have meanings on other computers, but those meanings aren't likely to have anything to do with the meaning 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.

SQUEAK COMMAND

One of the more HHC-specific features of SnapBASIC is the **SQUEAK** command. This command allows you to play music (of a sort) on the HHC "beeper". The format of the SQUEAK command is

```
SQUEAK pitch,time
```

Pitch is converted into an integer. 36 gives the highest note; 0 gives the sound of silence. 32 is approximately a middle C. A difference of 1 is approximately 1 semitone.

Time is converted to an integer. The time unit is approximately 5.88 msec. There are 170 counts/second.

Note that a very long value for the time will cause a very long squeak, and you cannot use the BREAK key to get out of it.

SQUEAK behaves interestingly for pitch values outside the range 0 to 36. If the pitch is greater than 36, the following formula applies:

```
SQUEAK A,time = SQUEAK MOD(A,12)+
    25,time
```

In other words, for large values, thee scale starts again at F below middle C.

For negative values of pitch, an interesting assortment of ticks, clicks, and squeaks are generated. (What you actually get are octaves below the normal range; of course, the beeper is not an ideal musical instrument, so mostly you get noise).

Example: To make the HHC play a chromatic scale,

```
10 FOR I=24 TO 35
20 SQUEAK I,10
30 NEXT I
```


CHAPTER 21: ADVANCED FILE TECHNIQUES

In this chapter we will cover some ways of doing sophisticated file operations with SnapBASIC.

VARIOUS FILE TYPES {H}

In this tutorial you have already encountered two different file types:

The first is used by the HHC general file system. It consists of lines of ASCII characters. These lines are called records, and can be displayed using the HHC system Editor.

The second file type you have encountered is the SnapBASIC program file. This type of file can be transferred through the use of the HHC File System, but it can only be examined and displayed via SnapBASIC itself.

We will now discuss the file types that can be used with your SnapBASIC programs:

1. The binary file.
2. The general ASCII file.

The structure of any file is shown below:

The "file information" field contains the following data:

Byte 0,1: Length of this file in bytes (including this field).

Byte 2: File type code. See the ***Reference Guide*** for an explanation of the file types.

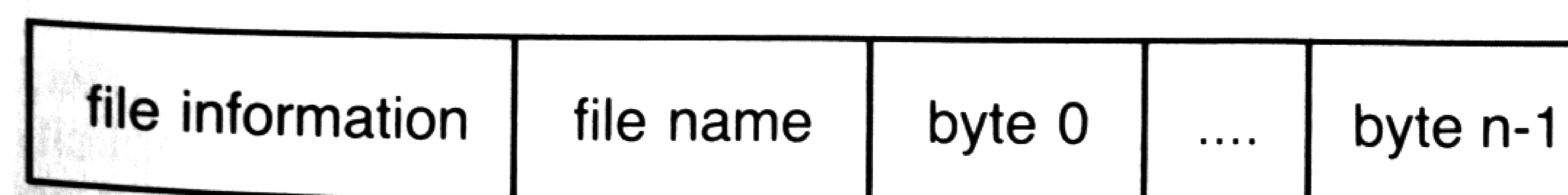
Byte 3: Length of the file name.

The file name part contains the name of the file as a sequence of ASCII characters. The file name is limited to 255 characters.

The file contents part contains the information that the user thinks of as being the 'file', i.e. the information that can be used by the programs, or the program itself.

BINARY FILES {H}{B}

The structure of a binary file is as follows:



The contents of the file is a sequence of arbitrary bytes. This last is in contrast to text files, which are limited in content to ASCII characters.

Accessing and manipulating binary files is facilitated by the following commands:

FOPEN sc

Opens a file with name ***sc***. If the named file already exists, it is made available for use with other file commands. If there is no file with this name, then a new file is created. (If it is necessary to test whether the file is already present, the EXIST function should be used; see below.)

FPUT place, value

Put the byte ***value*** at position ***place*** in the file. File positions are numbered from 0 to n-1, where n is the number of bytes in the file.

FREVISE position,length

Modify the size of the file by ***length*** bytes. If ***length*** is positive, then the file is extended; if ***length*** is negative, then the file is shortened. Bytes are deleted or added at ***position***. Note that the contents of the new part of the file are undefined.

Example: Create the file BINARY.EXAMPLE, make it 256 bytes long, and fill each position in the file with the number of that position.

```
10 FOPEN "BINARY.EXAMPLE"
20 FREVISE 0,256
30 FOR I=0 TO 255
40 FPUT I,I
50 NEXT I
```

The following functions are available to obtain information about the file and its contents:

EXIST(sv)

Tests for the existance of the file ***sv***. Returns TRUE if the file exists, and returns FALSE otherwise.

FADR(0)

Returns the address of the start of the File information of the currently open file at the time the call is made. Returns 0 if no file is open.

Note: This address may change! Files "float" in HHC memory, and many operations (both explicit and implicit) may change this address. The function FADR(0) should be called immediately preceding any operation that depends upon the information.

Note also that the 0 is a dummy parameter.

FGET(place)

Fetches the byte at position ***place*** in the currently open file.

FLEN(0)

Returns the number of data bytes in the currently open file.

Example: We will modify the previous example so that no new file is created if there is already one present, and we will print out the contents of the resulting file.

```
5 XXX = FREE(FREE(0) - 300)
10 NAME$ = "BINARY.EXAMPLE"
20 IF EXIST(NAME$) THEN FOPEN NAME$ :
    GOTO 80
30 FOPEN NAME$
40 FREVISE 0,256
50 FOR I=0 TO 255
60 FPUT I,I
70 NEXT I
80 FOR I = 0 TO FLEN(0) - 1
90 PRINT FGET(I);" ";
100 NEXT I
```

Note line 80. FLEN(0) returns the number of bytes in the file; the bytes are numbered from 0 to FLEN(0) - 1.

Line 5 is included to allocate space for the new file, since SnapBASIC eats up all available memory when it is entered, and an OM (out of memory) error is likely to occur otherwise.

TEXT FILES

{**H**}{**B**}

Text files are a special kind of binary file, where the contents consist of a number of records, each specified by the number of characters followed by a string of characters. The maximum number of characters in a record is 255; the minimum is zero.

The structure of a text file is as follows:

file info	file name	len1	s1	...	lenn-1	sn-1

This type of file is identical to that used by the HHC File System and Editor, so that it is possible to exchange information between SnapBASIC and the File System.

The "len" fields are bytes containing the length of the following strings. The strings are "len" bytes long. Note that unlike some file systems, records are not terminated by any explicit character (such as carriage return); instead, the length byte is the sole consideration in determining the end of the string.

A string file is a subset of the binary file, in that the byte sequence follows specific rules and the contents of the bytes are (theoretically) limited to character representations.

The SnapBASIC commands and functions for use with binary files may be used with string files. In addition, there are a number of specific operations designed for use with string files.

FWRITE recnr, sv

Writes the string ***sv*** to record number ***recnr*** in the currently opened file.

FREAD recnr,sv

Reads record number ***recnr*** from the currently opened file, putting the result in ***sv***.

FDEL recnr

Deletes record number ***recnr*** from the currently opened file. Moves any higher numbered records down to fill the space of the deleted one.

FINS recnr,s

Inserts the string ***s*** at position ***recnr***, moving any existing line with number recnr and higher upwards to make space for the new one.

FREC(0)

Returns the number of records in the currently opened file. The (0) is a dummy parameter.

(Warning: Though it is generally safe to use the binary file commands with text files, it is dangerous to attempt to alter certain information—specifically, altering the length bytes will destroy the integrity of your file. What's more, if you FPUT a value greater than 127 into a text record, it will be accepted by the system, but SnapBASIC will be unable to interpret the resulting string appropriately.)

With these commands, you can manipulate any line (record) of any text file. Together with the HHC file edit commands, you can change, edit and string (up to 255 characters long, HHC file editor=80) and use it with a SnapBASIC program to manipulate the sequence of characters within any file. This opens up possibilities for many applications.

EXAMPLES

The following examples show some possibilities for the use of the file handling commands and functions.

To generate a file of any desired file type:

```
FOPEN <filename>
POKE FADR(0)+2, <filetype>
```

To read the thirteenth line (record) of the file into the string variable A$:

```
FOPEN <filename>
FREAD 12,A$
```

(Note that this will read record number 12; since the first record is number 0, this will be the thirteenth.)

To write string B$ into record number 14 of a file:

```
FOPEN <filename>
FWRITE 14,B$
```

Note that any information that was originally in the record will be overwritten. If the length of the string B$ is greater than the length of the original record, then the file will expand (get longer). If B$ is shorter than the original record, then the file will shrink.

To insert the string C$ at the second line of the file:

```
FOPEN <filename>
FINS 2,C$
```

Note that all lines with record numbers 2 and higher will shift to make room for the new line.

FILE ERRORS

The following file errors can occur as FI ERROR:

1. File not opened.
2. Attempting to read or write a non-existant record number.
3. No room in opened file.

SPECIAL FILE TECHNIQUES {**H**}{**B**}

Here are some useful techniques for dealing with files:

To insert string A$ as the ***last*** record of a file:

```
FOPEN <filename>
FINS FREC(0),A$
```

Note that FREC needs a dummy argument, and returns the number of records in the file.

To obtain information on the amount of available memory for SnapBASIC and files, use FREE(0). This function frees as much space as possible.

To obtain 300 bytes for your SnapBASIC program, use the following method:

After all your arrays have been dimensioned, and if you only have a small number of strings:

```
XX = FREE(300)
```

To obtain 500 free bytes for your files, use

```
XX = FREE(FREE(0) - 500)
```

Note that the value that FREE returns is the number of bytes actually allocated to free space. SnapBASIC constantly contends with the file system for space. Since strings are dynamically allocated, as well as arrays (that is, the space is allocated on the fly, when necessary), SnapBASIC tries to grab as much space for its own use as it can. Thus, no room is left for file creation unless explicitly done so with the FREE command.

SUGGESTIONS FOR USE OF FILES IN SnapBASIC

Here follow some suggestions for possible use of the file system in SnapBASIC. The possibilities are only limited by your imagination.

1. You can create a file and fill it with information using the HHC File System Editor. For example, create a personal file with:

   name
   address
   city
   zip code
   phone number

   where the entries have fixed positions within a line.
2. You can then create and use a SnapBASIC program to access the above file, to read it, to sort it. For instance, you could sort the information based on city names, and obtain a list following the alphabetical order of the cities. A program to do this is at the end of this chapter.
3. Another use is to create an ***input*** file for SnapBASIC using the editor. You can now correct any mistakes before the SnapBASIC program runs.
4. A SnapBASIC program can then use the entries in the file as input to the program with little chance of typing errors.

In general, the Editor allows you to generate your own ***data base***, whereas SnapBASIC allows you to manipulate the data in the data base using the powerful SnapBASIC commands.

EXAMPLE: A TEXT FILE SORTER

This is a program that will sort a text file. The file may be sorted on any column. This program is useful, for example, when you have a file with information in fixed fields: say, last name in column 1, first name in column 10, and so on. This is an example of a "bubble sort", which is about the smallest (in size) and slowest (in speed) among the many sort techniques.

```
10 REM Text file bubble sort.
20 REM This sort can be done on
   any column.
30 INPUT "File name? ",NAME$
40 IF EXIST(NAME$) THEN 70
50 PRINT "That file does not exist."
60 GOTO 30
70 XX = FREE(600)
80 FOPEN NAME$
90 IF FREC(0) THEN 120
100 PRINT "That file is empty..."
110 GOTO 30
120 NUBOT% = FREC(0)-1
130 INPUT "Sort on which column? ",
    COL%
140 IF (COL% > 0) AND (COL% <= 255)
    THEN 170
150 PRINT "Column must be 1 to 255."
160 GOTO 130
170 COL% = 1 - COL%
180 IF COL%=0 THEN COL%=255
190 REM COL% is now number of chars to
    the right
195 REM of the sort column
    (negated)
200 SWAP? = FALSE
205 REM SWAP? = any swaps this pass?
210 FREAD 0,A$
220 FOR I% = 1 TO NUBOT%
230 FREAD I%,B$
235 REM See if sorted
240 IF RIGHT$(A$,COL%) >
    RIGHT$(B$,COL%) THEN 270
```

```
250 IF RIGHT$(A$,COL%)<>
    RIGHT$(B,COL%) THEN 330
255 REM The right parts are equal;
    sort on left.
260 IF A$ <= B$ THEN 330
265 REM Get here if pairs are not
    in order
270 FWRITE I%,A$
280 FWRITE I%-1,B$
290 SW=I%
300 SWAP? = TRUE
310 GOTO 340
320 REM Get here if the pairs are
    already sorted
330 A$ = B$
340 NEXT I%
350 NUBOT% = SW
360 IF SWAP? AND (NUBOT% > 1)
    GOTO 200
370 INPUT "Sort complete! Again
    (Y/N)?",YN?
380 IF YN? GOTO 30
390 PRINT "All done."
```

This program shows many special aspects of SnapBASIC.

a) Integers are used everywhere to speed up the process.
b) The file commands are easy to use.
c) The powerful string functions allow negative arguments (as in lines 240 and 250).
d) Logical variables are used.
e) Usually, a bubble sort requires an array. But because the HHC stores files in memory, there are no arrays; instead, the sort is done directly on the file. However, that is also why the routine is fairly slow. It might be faster to read the file into an array; but then twice as much space would be needed.

This program also demonstrates effective validation of user input, always a good idea in an application program..po 0

CHAPTER 22: THE FINAL STAGE—PREPARING A CAPSULE {H}{B}

The final stage in the preparation of a SnapBASIC program for permanent storage consists of executing the **BURN** command. This command has the effect of converting the SnapBASIC program into a "capsule image" that can later be transferred into an EPROM capsule via the HHC's PROM burning facility. Of course, BURNing is completely optional; you are quite free to keep your program in the usual SnapBASIC form for as long as you like.

What are the advantages of an encapsulated program over a normal RAM file? The most obvious one is permamence: RAM files, no matter how carefully you protect them, run the risk of being accidentally or maliciously deleted. Another advantage is portability: it is quite a nuisance to distribute a RAM file for the HHC. Certainly, RAM files can be passed about with Programmable Memory Peripherals; they can also be transferred through the Telecomputing interface to the HHC. But these methods are tedious, at best: what's more, there is still no guarantee that the files, once transferred, will not be lost, necessitating retransferring the files once again.

There are some more advantages specific to the SnapBASIC system. A capsule prepared via the BURN command differs significantly from the SnapBASIC file in several ways:

a) The capsule does not require the SnapBASIC system to execute. It appears to the HHC as just another capsule, no different from any other HHC capsule. This means that the users of the capsule do not need any familiarity with the SnapBASIC system.
b) The encapsulated program executes considerably faster than it would under the control of SnapBASIC.
c) The program cannot be used with the SnapBASIC system at all. Users do not have the ability to examine or alter the program, giving a good level of security and protection to the program. (This is either an advantage or a disadvantage, depending upon your own desires for how the program is to be used.)

HOW TO PREPARE A SnapBASIC PROGRAM FOR BURNING {H}{B}

There are certain things you have to do to prepare a SnapBASIC program for BURNing. There are several things you can do in a SnapBASIC program that you must change if you intend to burn the program.

Most important, there are a number of constructs in SnapBASIC that cannot be BURNed at all. These are:

a) The following statements may not be included in your program: HISTORY ON, HISTORY OFF, BYE, NEW, RUN, TRON, TROFF, LIST, DEL, and PAREN.
b) Obviously, only statements in deferred mode can be burned, since only the numbered lines in the program are burned.
c) GOTO <expression> and GOSUB <expression> are not allowed. GOTOs and GOSUBs must refer to line numbers. (At any rate, GOTO <expression> and its kin are not good programming practice.) Such statements must be replaced by ON...GOTO or ON...GOSUB.
d) REM statements must be the first and only statements on their lines. You may not use the construct

```
10 A=1:REM Don't do this!
```

or you will get a BURN error.
e) Every line number referenced by a GOTO or a GOSUB must be present. Remember, your program might seem to run quite well—but that is no guarantee that every GOTO or GOSUB is being executed. For example, this line:

```
100 IF 5=4 GOTO 1200
```

will never attempt to branch to 1200, and so will never cause any problems when you RUN the program; but it is not BURNable if line 1200 does not exist. The only way to really make sure that all of your line numbers are present is RESEQuence the program; non-existant line references will be replaced by -1 in the resulting program.

These constructs must be removed from your program before a BURN is possible.

You must have sufficient RAM available to execute the BURN and to store the BURNed file. The BURNed file is going to be larger than the SnapBASIC source file, unless the source file has loads and loads of very long remarks and a lot of very long variable names.

The very smallest program, that consists of nothing but remarks (quite useless!) takes something on the order of 1300 bytes in BURNed form. The largest program requires on the order of 6500 bytes just for the run-time support: this would be a program with every function called and every construct used.

More RAM is also required during the BURN process—the amount varies depending upon your program; at any rate, figure at least 25% more memory than the program is going to take.

www.pocketmuseum.com

A BURNed program functions somewhat differently from the way it would function in SnapBASIC. First of all, and most obviously, the BREAK key no longer functions. Type-ahead is still possible, but the amount of acceptable type-ahead is reduced by fifteen characters. (For example, on the keyboard, it is reduced from 24 to nine.)

Since the BREAK key does not exist as such, the CLEAR key can be used to accomplish the same purpose. This is consistant with the operation of other HHC capsules. Be careful, though: if you press CLEAR twice, very quickly, you will end up with a scratch file lying around with the same name as the capsule. You will want to get rid of this file as soon as possible.

The capsule will use a scratch file of the same name as the capsule name, during running of that capsule.

HOW TO BURN

To BURN a program, proceed as follows:

a) ***First and foremost***, make a copy of the file using the File System utility. The BURN command destroys the old file, even if the BURN command is unsuccessful, and you certainly don't want to lose all of your work. You should put the copy to be BURNed in a RAM bank that has plenty of free space.
b) Next, look over your program and make sure that you are not using any of the aforementioned unBURNable statements.
c) RUN your program, and make sure it still works.
d) Decide on what name you want your program to have when it appears on the HHC menu. For example, le's call it "My program".
e) (This step is optional.) Execute the following command:

```
POKE 535,0
```

This will speed up the display considerably (and thus the BURN).
f) (This step is optional, and is only really necessary for very long BURNs.) Execute the following command:

```
POKE 101,BAND(PEEK(101),127)
```

This disables the Auto-Off timer. BURNing sometimes takes quite a while, and you don't want the HHC going to sleep on you. (Not that it hurts anything: you can always press ON to continue. But it is a nuisance.)
g) If you want to, attach your micro printer (or whatever hard copy device you have). A permanent record of the BURN process could be helpful.

Not for sale

h) Now you can execute the BURN command. Say

```
BURN "My Program"
```

It will take a little while for anything to start happening. This is because it takes SnapBASIC some time to get its own affairs in order before the actual BURNing can happen.

Watch what happens:

The LCD will display the first line of your program.

Then there is a wait, while SnapBASIC processes the lines. Notice that only the line numbers of REM and DATA statements are listed. SnapBASIC is busy merging DATA statements, eliminating variable names, and converting GOTOs and GOSUBs to absolute jumps and calls; SnapBASIC is also determining what parts of the runtime system need to be loaded for this statement.

After all this has happened, the second line is displayed, and there is yet another small wait .

Towards the end of your program, you may notice that the process is faster. This is because the run-time support for these lines has already been included due to the earlier lines.

You can let the process work unattended: If a BURN error occurs, SnapBASIC will display the line with the error, beep, and display the error message (which will be BU, OM, or GO) over and over again. You can get out of this by pressing any kep (except CLEAR), at which point you will be returned to the primary menu. Your program is lost and unsalvagable. Aren't you glad you backed it up?

If no error occured, the message

```
Size = nnnn
```

will be displayed, where nnnn is the number of bytes in the new capsule. Press any key, and you will be returned to the primary menu. Note that the SnapBASIC menu no longer shows the file you burned (though you can see it in the File System menu). This is because it is no longer a SnapBASIC file; rather, it is a capsule image file.

i) You can now transfer your file into EPROM, using whatever method you have available for burning EPROMs. If you have a Memory Extender with a RAM/ROM switch, you canb test your final program in ***simulated*** ROM space. Due to addressing conventions in the HHC, however, you cannot test your program in RAM space.

Of course, you should never press CLEAR from SnapBASIC. But if you press CLEAR twice (out of reflex, or whatever) in the middle of a BURN, there will be a capsule image file with the same name as the SnapBASIC file, but it will be full of garbage. You must go into the file system and delete that file, unless you want a booby trap lying around.

www.pocketmuseum.com

A SAMPLE BURN

An example of BURNing a working program: Try the example for STRF$ in the chapter on strings. That program looked like this:

```
10 INPUT N
20 A$ = STRF$(N10,-99,99,0,-2,2,0)
30 I=1
35 REM Insert leading dollar sign
40 IF SEARCH(A$, " ",I) THEN I=I+1:
   GOTO 40
50 A$=INSERT$(A, "$",I-1)
55 A$=A$-" "
56 REM Insert commas. Is there a
   decimal point?
60 C=SEARCH(A$,".",1)
70 IF C < 4 THEN 120:REM No decimal
   point, or too short
80 FOR I=C-4 TO 1 STEP -3
90 IF ABS%(CHAR(A$,I)-ASC("5")) > 5
   THEN 120
100 A$=INSERT$(A$,",",I)
110 NEXT I
120 PRINT A$;
130 GOTO 10
```

First of all, the program needs to be checked out for illegal constructs. It looks OK: but try to burn it. Line 70 will be displayed, with a beep and a BU. Oh, right—it is illegal to have a REM statement unless it stands alone on a line. Aren't you glad you made a copy of the file? Press a key to return to the primary menu. Re-enter SnapBASIC, and select the copy of the file. Edit the line:

```
70 IF C < 4 THEN 120
```

Just get rid of the REM. Now make a new file, and try the BURN again. You get the following display (after a wait):

```
10 INPUT N
20 A$ = STRF$(N,10,-99,99,0,-2,2,0)
30 I=1
35
```

Yes, line 35 is blank. It was a REM, so nothing is displayed.

```
40 IF SEARCH(A$," ",I) THEN
   I=I+1:GOTO 40
50 A$=INSERT$(A$,"$",I-1)
55 A$=A$-" "
56
```


Not for sale

Another blank line.

```
60 C=SEARCH(A$,".",1)
70 IF C < 4 THEN 120
80 FOR I=C-4 TO 1 STEP -3
90 IF ABS%(CHAR(A$,I)-ASC("5")) >
   5 THEN 120
100 A$=INSERT$(A$,",",I)
110 NEXT I
120 PRINT A$;
130 GOTO 10
Size = 3783
```

and now SnapBASIC is waiting for you to hit a key. When you do, the HHC main menu is displayed. You are done! Notice that the file no longer appears on the SnapBASIC menu. You are now ready to actually transfer to EPROM. This program will fit into a 4K byte EPROM capsule (with 313 bytes to spare).

CHAPTER 23: EXCEPTION HANDLING—THE ONERR STATEMENT

Suppose you have written a Basic program designed for the most naive user. This person knows nothing about programming, nothing about computers—all this person wants to do is use the program you have provided to do some very important task. Now, midway through the program, suddenly a message pops up on the LCD saying

```
**** IQ ERROR
```

"What an insult! What's wrong with my intelligence?" What's worse, all the work that had gone on is wiped out, deleted, and there is really nothing the user can do but start all over again. Most unpleasant. Result: one very irritated user. (Also, maybe one lost customer.)

Of course, we know what has happened. We have used BASIC, and know that the IQ error probably happened because a really big number was entered, or something like that. But there should be a way to catch problems like this before a message is displayed.

Certainly, it would make more sense for the program to check all of the numbers entered by the user, to make sure that they are in the proper range. Good programming practice mandates that all input be validated, and as such numeric problems can ***usually*** be avoided. But not always.

SnapBASIC provides a statement called **ONERR**, which causes control to pass to a particular line in the program when an error condition occurs, or when the BREAK key is pressed. This statement looks as follows:

```
ONERR GOTO linenumber
```

Certain information is lost when an error occurs, even if there is an ONERR in a program. Most important, any subroutines, FOR/NEXT loops, and functions are terminated. Hence, there is no simple, automatic way to get back to the place in the program where the error occurred.

There is one piece of information available from the system, and one only. Specifically, the error code can be determined by PEEKs at locations 852 and 853. Stored here will be two characters: either the two-letter error code (like IQ) or the letters 'Br' if the break key had been pressed.

Any ONERR statement remains effective until the program ends, or until the BREAK key is pressed. The first time the BREAK key is pressed, any pending ONERR is disabled; the

next time, therefore, BREAK will interrupt the program, as usual (unless another ONERR is encountered in the meantime.)

It is possible to seed your programs with flags (variables) that can tell the error-handling routine what was happening when the error occurred. For example:

```
10 REM A program to do something with
     files.
20 REM FLAG tells what we were doing.
30 ONERR GOTO 2000
40 FLAG = 1
50 INPUT "File name? ";FN$
60 FOPEN FN$
70 FLAG = 2
.
.
.
2000 ER$=CHR$(PEEK(852))+
     CHR$(PEEK(853))
2010 ON FLAG GOTO 3000,4000
2020 PRINT "Unknown error.":STOP
3000 IF ER$<>"OM" GOTO 2020
3010 REM Free up some memory
3020 XXXX = FREE(FREE(0)-300)
3030 GO TO 30

4000 REM Handle other errors...
```

This code then will free room for the file, but only if the system first complains that there was not enough room in the first place. Other errors can be handled similarly.

ANOTHER USEFUL TRICK

Another useful thing to do with ONERR is to wipe out information that just should not be accessible to the user—passwords, in particular. Say the user presses BREAK in the middle of the program. Then there is nothing to stop the user (assuming the user knows how) from examining memory till he finds something that looks like a password, and poof! there goes your security. Deal with that problem like this:

```
10 ONERR GOTO 1000
.
.
.
1000 ONERR GOTO 1000
1010 CLEAR:END
```

Why are there two ONERR statements? Simply because a BREAK disables any previous ONERR—and if there were only one ONERR, a second quick BREAK would stop things before the CLEAR was executed.

It should be pointed out again and again that the use of ONERR to control errors caused by mathematical problems and the like should really be eliminated ahead of time, by validating input and not trying to operate on invalid parameters. Most of the time, there will be nothing salvagable after an error: all you can really do well is to start all over again. This works nicely:

```
10 ONERR GOTO 1000
1000 GOTO 10
```

There is one oddity in the operation of ONERR that you should be aware of. If you have the statement

```
10 ONERR GOTO 1000
```

and line 1000 does not exist, SnapBASIC will go into an infinite loop! Why? Think for a second. When the first error occurs, SnapBASIC tries a GOTO 1000. But this causes another error, since line 1000 can't be found. So SnapBASIC does what it is supposed to: it tries to GOTO 1000 again—causing the same problem. Ad infinitum. Moral: Make sure ONERR really has a place to go.

CHAPTER 24: LOADING PROGRAMS FROM FILES AND PERIPHERALS {H}{B}

There are certain situations in which you might want to get programs into SnapBASIC in some other way than by typing them in.

Chief among these is the fact that if you make a single mistake on a line, you have to type the entire line in from scratch (since none of it is recoverable). If you are a sloppy typist, this can be aggravating.

Compatability is another factor. If you have a program that is in some other version of BASIC, it might contain statements that are not proper SnapBASIC statements, and will cause errors if sent directly to the compiler. However, you could change these statements very slightly to make them SnapBASIC statements.

Another reason is that some programs are ***long***, and you might want to be able to get the program automatically loaded into the HHC. But SnapBASIC only understands files created by SnapBASIC. So what do you do?

THE LOAD COMMAND {B}

You are free to create and edit your file using the Portawriter capsule, or the HHC File System Editor. Once you have done this, the **LOAD** command is quite useful. This command looks like this:

```
LOAD "filename"
```

The LOAD command expects a text file as input, and turns it into a SnapBASIC program. When it is executing, it prints out a number for each line it is reading in. LOAD will generate error messages, just as if you typed the lines yourself. The erroneous lines will not be in the program—but you can go back to the File System Editor (or Portawriter), correct them, and reLOAD; or you can just type the missing lines in SnapBASIC.

A file that is to be LOADed may not contain any commands. Only line-numbered statements are allowable. Some immediate commands are executable through LOAD, but this is not reliable and should not be counted upon.

The file that is to be LOADed must reside in the same RAM bank that you are running SnapBASIC from; you cannot execute from internal RAM and load from external RAM, for

example. Use the File System COPY command to get around this problem.

You can only LOAD text files; you may not LOAD other SnapBASIC files.

The LOAD command is only available in immediate mode, and cannot be executed from within a program.

{H} DOWNLOADING PROGRAMS FROM OTHER COMPUTERS

Another thing you can do is download your program from another computer, as text files to be LOADed, or directly to the SnapBASIC compiler. As a matter of fact, when we were writing this book, we downloaded all of the example programs from the word processing system we were using to HHC, so that they could be tested. (That's how we know they all work!) You can also transfer programs from one HHC to another.

You have several options, depending upon what computer you are trying to download from, and what HHC peripherals you have available. Probably the easiest method is via the Serial Interface Adaptor with the RS-232C Capsule. This requires direct access to the "host" computer. (For the sake of brevity and clarity, we will refer to the computer from which programs are being obtained as the "host".) This host computer might be a personal computer, a word processing system, or another HHC.

One of the primary advantages of this method is that it compiles "on the fly": in other words, the actual source text is not preserved, only the compiled SnapBASIC code. Here's what happens: SnapBASIC accepts characters (about thirty at a time), and compiles them directly into SnapBASIC's internal form. As such, the actual source text is never permanently stored in the HHC. This represents a considerable space savings (at the expense of compilation time).

This method is not suitable for downloading programs that are not SnapBASIC. See the following section for ways to download other types of files.

The host computer, of course, must have the ability to transfer ASCII files over an RS-232C cable. Set up the host computer telecommunications capability in whatever fashion is most fitting for the host. The only absolute requirement is that the host computer support some sort of protocol: as a bare minimum, the host should be able to receive XON/XOFF and suspend transmission accordingly. Alternately, the host could support the ACK/ETX protocol also available to the RS-232C peripheral, or DTS hardware handshaking. (If your host computer can not handle this, another method is required. See the next section.)

The source file must be prepared in a somewhat special way so that SnapBASIC can process the file properly. First, and most important: all line feed characters (and other form controls, with the sole exception of carriage return) must be stripped from the file. SnapBASIC does not understand linefeeds, and will produce error messages (usually CH for illegal character) if they are encountered in the file. Also, other control characters should be removed. (Some word processors insert control characters into the file). This file preparation can either be done before the file is transmitted, or by the file transmission program itself.

Append the following statement to the end of the source file:

```
ATTACH 129 TO #0
```

This will return control of the HHC to the keyboard once the transmission is complete. (Note that command lines can be included in the source file!)

Set up the RS-232C Adaptor on the HHC to correspond to the requirements of the host computer. See the RS-232C manual for this procedure. You should ignore the data parity bit; you ***must*** select some sort of handshaking protocol (SEND XON/XOFF will suffice).

Now, get into SnapBASIC. Take the following step to speed up the process:

```
POKE 535,0
```

All the text that is transmitted will be displayed on the LCD; this poke sets the display rate so that it is considerably faster than what you would get from pressing STP/SPD 0.

Make the RS-232C Adaptor the input device by executing the following command:

```
ATTACH 134 TO #0
```

Note that none of the keys on the HHC (except the dangerous CLEAR key) work any more.

Now, tell the host computer to start the file transmission. Note that the lines of the source program are displayed on the LCD, just as if you were typing them in yourself. If there are any errors in the source code, the error messages will be displayed on the LCD. Program transmission will continue after the error; you might have to re-enter the lines by hand. (If there are a lot of very strange errors, something is probably wrong with your transmission setup.)

Because you appended the ATTACH line to the source file, once the source file is completely transmitted, the keyboard will once again be active. This would be a good place to examine the file (using LIST) to make sure that it was transmitted properly.

If CLEAR is pressed during the file transmission, the contents of your file will be unreliable, and inaccessable. You must press CLEAR twice, to be able to get back into the file at all—and then the only thing you can really do is delete the file.

There is no easy way to stop the transmission process in mid-stream. Because you have lost control of the keyboard, the BREAK key no longer functions. What you have to do is:

a) Stop the host computer from transmitting.
b) Make the host computer send the line

```
ATTACH 129 TO #0
```

to give control back to the keyboard. (It is wise to preceed this line with a carriage return to clear any partially-transmitted lines from the buffer).
c) Now you have full control of the HHC again. Use LIST to see what has happened.

If you can't get the host computer to return control, drastic measures are necessary. Press CLEAR twice. Immediately select SnapBASIC again, and select the file you were working on. If you are lucky, you will be able to salvage some of what was transmitted...but you might be better off to start from scratch all over again.

{H} OTHER WAYS TO DOWNLOAD

Suppose one of the following considerations applies:

a) Your host computer cannot handle any handshaking protocols.
b) The BASIC program you want to download is not in SnapBASIC.
c) You cannot remove control characters (like linefeed) on the host computer side.
d) You want to download from a computer over the phone lines.

In these cases, downloading directly into SnapBASIC will not be appropriate. In this case, you will need one of the Telecomputing series capsules. These capsules have the ability to save text into files for you. (They can also save binary files, which makes it possible to transfer SnapBASIC programs in object form from one HHC to another).

If your host computer cannot handle any handshaking, the Telecomputing 2 capsule can load files with no handshaking at speeds of 300 baud or less. This is quite slow, certainly; but it is one way to work. (Of course, if you are transferring over the phone lines, 300 baud is about as fast as you can go with the HHC modem.)

Follow the directions in the Telecomputing manual to establish communications with the host computer. Save the BASIC file with the RCVE key. You then have a way to edit the non-SnapBASIC statements with the File System Editor (or Portawriter) into the proper format. You can then use the SnapBASIC LOAD command to get the file into SnapBASIC.

If you need to massage the text file to remove line feeds and control characters, here is a SnapBASIC program that will do it for you:

```
10 INPUT "File to massage: ";FILE$
20 IF EXIST(FILE$) GOTO 50
30 PRINT "That file does not exist."
40 GOTO 10
50 FOPEN FILE$
60 IF PEEK(FADR(0)+2) = 8 GOTO 90
70 PRINT "That is not a text file."
80 GOTO 10
90 FOR I=0 TO FREC(0)-1
100 FREAD I,L$
110 FOR J=LEN(L$) TO 1 STEP -1
120 IF CHAR(L$,J) >= ASC(" ") GOTO 140
130 L$ = ERASE$(L$,J,1)
140 NEXT J
150 FWRITE I,L$
160 NEXT I
```

If you want to transfer a SnapBASIC file from one HHC to another, just use the Telecomputing 2 binary file transfer capability to get the file into your HHC.

www.pocketmuseum.com
Not for sale

INDEX

A

◄ key, 4-6, 6-2
► key, 4-6, 6-2
▲ key, 4-5, 6-5
▼ key, 4-5, 6-5
  LIST and, 4-4, 4-6
ABS function, 15-3
AC Adaptor, 2-1
Adaptor
  I/O, 2-3, 7-7
Address, 18-2
ALL OFF switch, 2-1, 19-1
American National Standards Institute, 16-19
AND, 9-9
ANSI, 16-19
Argument, 15-1
Array, 11-1
  dimension, 11-2
Arrays
  Boolean, 10-6
  integer, 10-5
  limits, 11-6
  multi-dimensional, 11-5
ASC function
  compared with VAL, 16-23
ASCII, 20-1
ASCII representation, 16-19
Assignment
  Strings, 16-3
Assignment statement, 3-1
Attach codes, 14-9
ATTACH statement, 14-4, 14-8
AUTO command, 4-8
Auto-off timer
  disabling, 18-3
Auto-repeat
  Speed of, 5-8
Auto-repeat feature, 4-8
Auto-shutoff feature, 2-13

B

Back-up, 7-4
BAND function, 10-4, 15-4
Batteries, recharging, 2-1
Binary file structure, 21-1

Blip, 2-2
  DELETE, 6-3
  INSERT, 6-1
  LOCK, 6-6
Body of a function definition, 15-6
Boolean
  arrays, 10-6
  mixing, 10-5
  operations, 10-5
  values, 10-6
Boolean variable, 10-5
  name, 10-5
BOR function, 10-4, 15-4
Boundary condition, 12-5
BREAK key, 4-6, 9-2, 12-10
  disables ONERR, 23-1
Breaking execution, 9-2
Bug, 12-8
BURN command, 1-2, 22-1
  preparing for, 22-1
  REM statements, 8-2
BXOR function, 10-4, 15-4
BYE command, 2-2, 4-9, 7-3
BYE key, 7-2

C

: statement separator, 9-10
C1 key, 9-2
Call to a subroutine, 17-1
Capsule
  advantages of, 22-1
  differences from SnapBASIC file, 22-1
Character, 16-18, ***see also*** String
  Case translation of, 16-22
  Conversion to number, 16-22
CHR$ function
  compared with STR$, 16-23
CLEAR key, 2-2, 7-2, 14-6, 18-3, 19-3
  avoid!, 9-2
Code, 4-3
Coding, 4-3
Column, 11-5
Command, 4-2
  AUTO, 4-8
  BURN, 22-1
    REM statements, 8-2
  BYE, 2-2, 4-9, 7-3
  CONT, 12-12
  LIST, 4-4, 14-7
  LOAD, 24-1
  RESEQ, 4-9
  RUN, 4-2, 4-10, 12-12
Commands
  BURN, 1-2
Communications protocols, 24-2
Comparison
  String, 16-7
Computer program, 1-1
Concatenation, 16-5
Constant, 2-6
CONT command, 12-12
Control character, 20-1
Copying files, 7-3, 7-4
Current memory area, 7-6
Cursor, 2-4
  checkerboard, 6-1
  Empty box, 6-3
  Left movement, 4-6
  Right movement, 4-6

D

Data base, 1-2
Data byte in escape control sequence, 20-2
DATA statement, 14-1, 16-4
Debugging, 12-8
DEF statement, 15-6
Deferred mode, 4-2
DELETE Key, 6-3
Deleting a line, 4-7
DETACH statement, 14-5
Device
  peripheral, 2-3
Device independence, 14-7
DIM statement, 11-2
  required, 11-2
Dimension, 11-2
Division
  by zero, 5-2
Documentation, 8-1
Downloading, 24-2
  preparing program for, 24-3

E

E (mathematical constant), 15-3
Echo
  GET suppresses, 16-25
Efficiency, 1-2, 17-5
Element, 11-1

www.pocketmuseum.com

Not for sale

END statement, 9-12, 12-12
ENTER key, 2-3, 19-1, 19-2, 20-1
EPROM, 4-1, 22-1
Erasable PROM, ***see*** EPROM
ERASE$ function, 16-7
Error, 2-5
  Checking for, 16-10
  Debugging, 12-8
  Deferred mode, 5-1
  NEXT without FOR, 13-2
  Return without GOSUB, 17-7
  Subroutines and error conditions, 17-4
  Syntax, 2-5
  Undefined statement, 9-2
Error message, 2-5
Escape control sequence, 20-2
Execution, 2-6
  Halting, 9-2
EXIST function, 21-2
EXP function, 15-3
Exponent, 5-3
Extrinsic RAM, 7-5

F

FADR function, 21-2
False value, 9-4
FGET function, 21-3
File
  text structure, 21-3
File names, 2-4
File system, 1-2, 2-4
  copying, 7-3, 7-4
  deletion, 7-1
  POKE and, 18-3
  renaming, 7-3
File type, 4-10
FLEN function, 21-3
Floating point number, ***see*** Real number
Flow of control, 9-1
FOPEN function, 21-2
FOR statement, 13-1
FOR/NEXT
  in immediate mode, 13-7
  relationship with GOSUB, 13-7
  STEP, 13-3
  with BREAK, 13-7
FOR/NEXT loop, 13-1
  index, 13-1
  initial value, 13-1
  limit, 13-2
  nesting, 13-4
Formal parameter of a function definition, 15-6
FPUT function, 21-2
FREE function, 15-3
Freezing the HHC's activity, 5-8
FREVISE function, 21-2
Function, 15-1
  ABS, 15-3
  BAND, 10-4, 15-4
  bitwise, 10-4
  BOR, 10-4, 15-4
  BXOR, 10-4, 15-4
  ERASE$, 16-7
  EXP, 15-3
  FREE, 15-3
  INSERT$, 16-6
  INT, 15-3
  integer, 10-4, 15-4
  LEFT$, 16-11
  LEN, 16-10
  limit to depth, 15-6
  LN, 15-3
  LOG, 15-3
  MAX%, 10-4
  MID$, 16-12
  MIN%, 10-4
  MOD%, 10-4
  PEEK, 18-1
  POS(0), 5-7
  RIGHT$, 16-12
  RND, 15-3
  SPC$(N), 5-7
  SQR, 15-3
  SQRT, 15-3
  STR$, 16-10
  STRF$, 16-14
  trigonometric, 15-3
  VAL, 16-11
Function key
  defining, 19-1
Function keys, 19-1

G

Generality, 17-5
GET command, 14-6
GET statement, 16-24
GOSUB statement, 17-1
  Error, 17-7
GOTO statement, 9-1

Not for sale

H

Hard copy, 14-3
HELP key, 19-1, 19-2
HHC capsule, 1-1, 2-1
History, 12-11
HISTORY OFF statement, 12-11
HISTORY ON statement, 12-11
Host computer, 24-2

I

I/O, 4-11
I/O adaptor, 7-7
I/O key, 7-1
  Copying files, 7-6
IF Statement
  IF/GOTO, 9-3
  IF/THEN, 9-6
  Multi-statement lines, 9-11
Immediate mode, 4-2
  editing, 6-6
  FOR/NEXT, 13-7
Increment, ***see*** Step
Index of FOR/NEXT loop, 13-1
Initial value of FOR/NEXT loop, 13-1
Initial value of variable, 3-2, 4-10
INPUT statement, 4-11, 14-6, 16-3
  Scientific notation, 5-3
  Without LUN, 14-7
INSERT Key, 6-1
INSERT$ function, 16-6
INT function, 15-3
Integer, 10-3
  arrays, 10-5
  conversion to real, 10-4
  functions, 10-4
  mixing with reals, 10-4
  name, 10-3
  operations, 10-3
  real number
    conversion, 15-4
  representation, 10-4
Integer arithemetic
  mixed with reals, 10-4
Integer arithmetic, 10-4
Interface to subroutine, 17-6
Intrinsic
  Application, 7-1
Intrinsic RAM, 4-1
Inverse video, 7-2, 19-2
Inverse video display, 4-3

K

Key
  2nd SFT, 2-3, 19-3
  ←, 4-6, 6-2
  →, 4-6, 6-2
  Auto-repeat, 4-8
  BREAK, 4-6, 9-2, 12-10
  BYE, 7-2
  C1, 9-2
  CLEAR, 2-2, 7-2, 14-6, 18-3, 19-3
    avoid!, 9-2
  DELETE, 6-3
  ENTER, 2-3, 19-1, 19-2, 20-1
  Function, 19-1
    defining, 19-1
  HELP, 19-1, 19-2
  I/O, 7-1, 7-6
  INSERT, 6-1
  LOCK, 6-2, 6-6
  OFF, 2-2, 19-3
  ON, 2-2, 19-3
  ROTATE, 6-5
  SHIFT, 2-3, 4-13, 19-3
  STP/SPD, 5-8
  ↓, 4-4, 4-5, 4-6, 6-5
  ↑, 4-5, 6-5
Keyboard, 14-9
  Auto-repeat speed, 5-8

L

LCD, 2-2, 14-9
  Scrolling, 5-4
  STP/SPD key, 5-8
LEFT$ function, 16-11
LEN function, 16-10
Limit of FOR/NEXT loop, 13-2
Line
  Maximum length of program line, 9-11
Line number, 4-2, 17-5
Line numbers
  usage, 4-9
Liquid crystal display, ***see*** LCD
LIST command, 4-4
  line too long, 4-3
  long lines, 4-3
  Printer, 14-7

www.pocketmuseum.com

Not for sale

LN function, 15-3
LOAD command, 24-1
LOCK key, 6-6
  Editing with, 6-2
LOG function, 15-3
Logarithm
  common, 15-3
  natural, 15-3
Logical operator, 9-9
Logical unit number, ***see*** LUN
Loop, 9-2
  FOR/NEXT, 13-1
LUN, 14-4
  Assignments of, 14-6, 14-7, 14-8
  PEEK and, 18-1

M

Main routine, 17-2
Mantissa, 5-3
MAX% function, 10-4
Maximum length of program line, 4-3
Memory, 4-1, 7-5
  Current, 7-6
Menu, 2-4
  Destination RAM, 7-5
  Primary, 2-4
  SnapBASIC, 2-4
Message
  error
    AE, 5-2
    AS, 16-10
    CH, 5-1
    CO, 5-2
    CX, 15-6, 17-7
    DA, 14-3
    DE, 16-5
    during ONERR, 23-1
    GO, 9-2
    IO, 14-5
    IQ, 5-2
    NX, 13-2, 13-5
    OM, 5-2
    RT, 17-7
    SY, 5-2
  NO ROOM,DELETE FILE, 7-5
  Rest Ignored, 4-12
  Retype Line, 4-12
Messages, 2-3
  errors, 2-5
  RESTART, 2-3
Micro printer, 14-9
  Use with HHC, 14-3
MID$ function, 16-12
MIN% function, 10-4
MOD% function, 10-4
Mode, 4-2
  deferred, 4-2
  immediate, 4-2

N

Nesting
  FOR/NEXT loops, 13-4
  functon calls, 15-6
  Subroutine call, 17-2
NEXT statement, 13-1
NO ROOM, DELETE FILE message, 7-5
NOT, 9-9
Null string, 16-1
Number
  Conversion to character, 16-22
  integer, 10-3
  real
    range of, 10-1
    representation, 10-1
  Scientific notation, 5-3
Numbers
  range, 5-3
Numeric constant, 2-6

O

OFF key, 2-2, 19-3
ON key, 2-2, 19-3
ON/GOSUB statement, 17-7
ON/GOTO statement, 9-11
ONERR statement
  lost information, 23-1
Operation code, ***see*** Opcode
Operation code in escape control sequence, 20-2
OR, 9-9

P

Peek function, 18-1
  Address or integer value, 18-1
Peripheral device, 2-3, 7-4, 7-7
  How to connect, 14-4
  Use with HHC, 14-3
  Uses memory area space, 7-6

www.pocketmuseum.com
Not for sale

PI (mathematical constant), 15-3
Plotter, 14-9
POKE statement, 18-2
  dangers and precautions, 18-3
Precedence
  relational operators, 9-5
Primary menu, 2-4
PRINT
  empty line, 5-7
  several statements on one line, 5-6
  SPC$(N), 5-7
  Zones, 5-4
    avoiding, 5-5
PRINT statement, 2-6
  Debugging aid, 12-11
  Without LUN, 14-7
Printer
  Debugging aid, 12-11
Program design, 9-8
Program development, 12-1
Program line
  Maximum length of, 4-3
Program security, 22-1
Programmable, 1-1
Programmable Memory Peripheral, 4-1, 7-4, ***see*** PMP
Programmable Read-Only Memory, ***see*** PROM
Programming language, 1-1
PROM, 4-1
Prompt, 2-4
  INPUT, 4-11, 4-12

Q

?
  PRINT statement, 2-7

R

RAM, 4-1
  Extrinsic, 7-5
  Free space in, 7-1
  Intrinsic, 4-1
  Programmable Memory Peripheral, 4-1
Random access memory, ***see*** RAM
Random number, 15-3
READ statement, 14-1, 16-4
Read-only memory, ***see*** ROM
Real number, 2-6
  integer
    conversion, 15-4
  precision, 2-7
  representation, 10-1
Real numbers
  mixing with integers, 10-4
Relational operator, 9-4, 9-5
  True and false values produced by, 9-4
Releational operator
  precedence, 9-5
REM statement, 8-1
  advantages, 8-2
  disadvanatages, 8-2
  during BURN, 8-2
  what to write, 8-3
Renaming a file, 7-3
RESEQ command, 4-9
Reserved word, 2-6
  Not allowed in variable name, 3-2
RESTART message, 2-3
RESTORE statement, 14-3
RETURN statement, 17-1
  Error, 17-7
Returning a value, 15-1
RIGHT$ function, 16-12
RND function, 15-3
ROM, 4-1
ROM socket, 2-1
ROTATE key, 6-5
Rotation, 6-4
  Speed of, 5-8
Row, 11-5
RS-232, 14-9
RS-232C, 24-2
RUN command, 4-2, 4-10, 12-12

S

2nd SFT key, 2-3, 19-3
Saving a program, 4-9
Scientific notation, 5-3
Scrolling, 5-4
SDT, 18-1
Second shift key, ***see*** 2nd SFT key
Serial Interface Adaptor, 24-2
SHIFT key, 2-3, 4-13, 19-3
SnapBASIC
  menu, 2-4
SnapBASIC compiler/interpreter, 1-1
  special features, 1-2
SnapFORTH, 1-2
SPC$(N), 5-7
SQR function, 15-3

www.pocketmuseum.com

Not for sale

SQRT function, 15-3
SQUEAK command, 20-3
Statement, 2-6, 4-2
  ?, 2-7
  Assignment, 3-1
  ATTACH, 14-4
  DATA, 14-1, 16-4
  DEF, 15-6
  DIM, 11-2
    required, 11-2
  END, 9-12, 12-12
  FOR, 13-1
  GET, 16-24
  GOSUB, 17-1
  GOTO, 9-1
  HISTORY OFF, 12-11
  HISTORY ON, 12-11
  IF, 9-3, 9-6
  IF/GOTO, 9-3
  IF/THEN, 9-6, 9-11
  INPUT, 4-11, 14-7, 16-3
  NEXT, 13-1
  ON/GOSUB, 17-7
  ON/GOTO, 9-11
  POKE, 18-2
  PRINT, 2-6, 12-11, 14-7
  READ, 14-1, 16-4
  REM, 8-1
    advantages, 8-2
    disadvantages, 8-2
    during BURN, 8-2
    what to write, 8-3
  RESTORE, 14-3
  RETURN, 17-1
  STOP, 12-12
  TROFF, 12-10
  TRON, 12-10
Statements
  Several on one line, 9-10
STEP operand of FOR/NEXT, 13-3
STOP statement, 12-12
Storage, 4-1, 4-2
STP/SPD key, 5-8
STR$ function, 16-10
  compared with CHR$, 16-23
STRF$ function, 16-14
String
  Assignment, 16-3
  Comparison, 16-7, 16-18, 16-19
  constant, 16-1
  Length of, 16-1, 16-10
  Null, 16-1
  PRINT, 5-5
  Substring, 16-11
  subtraction, 16-6
  Variable, 16-2
    name, 16-2
String constant, 4-12
Strings
  INPUT
    quote rules, 16-3
Subroutine, 17-1
  Error, 17-7
Subscript, 11-1
  Column, 11-5
  Row, 11-5
Substring, 16-11

T

Telecomputing, 14-9
  downloading, 24-4
Text file
  structure, 21-3
Trace, 12-10, 12-11
Trade-off, 17-5
Trig functions, 15-3
TROFF statement, 12-10
TRON statement, 12-10
True value, 9-4
TV Adaptor, 14-9
Type conversion, 10-4

V

VAL function, 16-11
  compared with ASC, 16-23
Variable, 3-1
  boolean, 10-5
  Initial value of, 3-2, 4-10
  name, 3-1
  Name rules, 3-2
    length, 3-2
  value, 3-1
Variable names
  usage, 3-2

X

XOR, 9-9

www.pocketmuseum.com

Not for sale

Z

Zones
PRINT, 5-4

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.

www.pocketmuseum.com

Not for sale