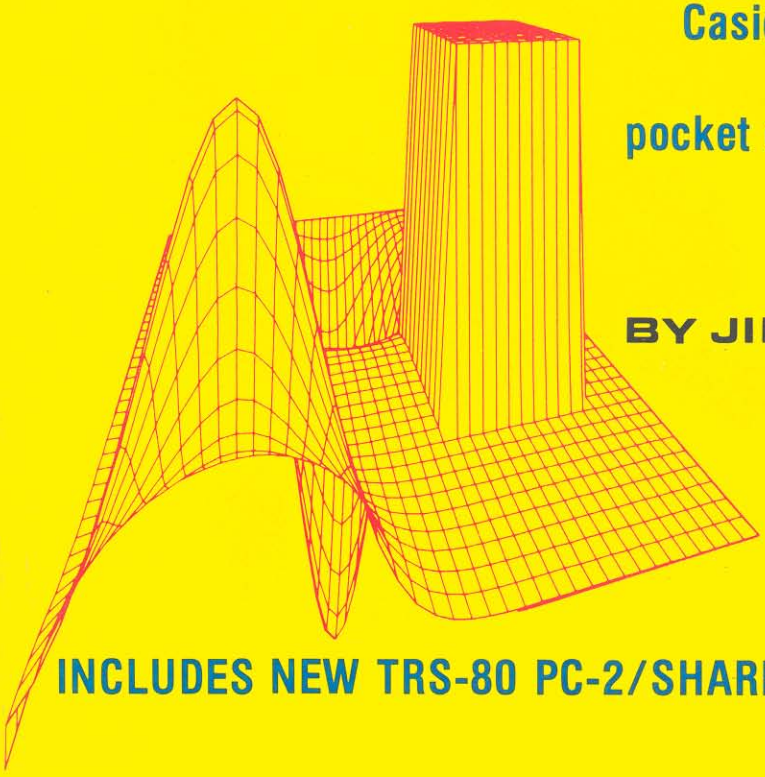
$8.95

POCKET COMPUTER PROGRAMMING MADE EASY

latest up-to-the-minute info for using
TRS-80 PC-2 and PC-1
Sharp PC-1500 and PC-1211
Casio FX-702P
and other
pocket computers



BY JIM COLE

INCLUDES NEW TRS-80 PC-2/SHARP PC-1500

ARCsoft pocket-computer books by Jim Cole:

I	Murder in the Mansion and Other Computer Adventures
II	50 Programs in BASIC for the Home, School & Office
III	50 MORE Programs in BASIC for the Home, School & Office
IV	101 Pocket Computer Programming Tips & Tricks
V	Pocket Computer Programming Made Easy
VI	35 Practical Programs for the Casio Pocket Computer

POCKET COMPUTER PROGRAMMING MADE EASY

POCKET COMPUTER PROGRAMMING MADE EASY

BY JIM COLE

ARCsoft Publishers
WOODSBORO, MARYLAND 21798

FIRST EDITION
FIRST PRINTING

Library of Congress cataloging-in-publication data:
Cole, Jim 1940—
Pocket Computer Programming Made Easy
Includes index
Summary: Advice for beginning computer programmers using the BASIC program language.
1. TRS-80 (Computer)—Programming.
2. BASIC (Computer program language).
3. Computers.
4. Programming (Electronic computers).
I. Title
QA76.8.T18C66 001.64'2 81-14882
ISBN 0-86668-009-8 AACR2

Trademark credits:
TRS-80 is a trademark of Tandy Corporation/Radio Shack.
PC-1211 is a trademark of Sharp Electronics Corp.
FX-702P is a trademark of Casio Computer Company.

ISBN 0-86668-009-8

Preface

Today you can hold in the palm of your hand what just one generation ago required a room full of hot, chugging computer behemoths. A half-dozen electronics manufacturers are selling hand-held battery-powered pocket-sized computers as fast as they can make them. And a dozen or more are feverishly researching, developing, designing and testing even more powerful versions of these tiny wonders.

True pocket computers are here. They have computation power almost beyond imagination and can communicate in near-English. These micromachines are at the forefront of the new wave of computers for the 1990s.

With such sophisticated tools at your fingertips—literally—we won't need the desktop dinosaurs. Stick one of those boat anchors in your attic. It'll make a great antique for a turn-of-the-century museum.

Meantime, keep your eyes peeled for ever-greater accomplishments in handheld pocket computers.

—Jim Cole

Table of Contents

Introduction

Introduction

Computers are practical, useful, fun, even exciting. But writing programs can be a drag unless you know BASIC, the most popular program language. In this book we will introduce you to BASIC in an easy-to-understand explanation of the most-used words.

This will be a straight-forward introduction to programming. We assume you have tried to read the owner's manual which came with your computer. You know how to turn it on. You know that pushing its buttons can't break it. Don't be afraid to experiment. We'll show you how to make it work for you.

This book is especially written for owners of the exciting new Radio Shack TRS-80 PC-2, the new Sharp PC-1500, the Radio Shack TRS-80 PC-1, Sharp PC-1211, Casio FX-702P Pocket Computer, Panasonic Link Hand Held, Quasar Hand Held Computer (HHC) and others.

However, the knowledge of BASIC which you will gain from this book will be applicable to *any* microcomputer,

minicomputer, or main-frame computer using the BASIC language. And all of today's popular microcomputers use BASIC.

Our simple down-to-earth instruction will help you quickly understand how to talk to your computer and make it do what you want.

The name of the language is BASIC. That stands for *Beginner's All-purpose Symbolic Instruction Code.* What does that mean? Well, you know *beginner.* That's you. *All-purpose* means it's generally useful for lots of different things. *Symbolic* reflects the fact that the comupter uses symbols to receive *instructions* from you. That is, symbols like the word PRINT or IF or THEN or FOR or NEXT. The symbols mostly are words you already know. *Code* is a buzz-word used by programmers to mean instructions to a computer.

So, you can translate *Beginner's All-purpose Symbolic Instruction Code* to mean "You use familiar words to tell your computer how to do just about anything."

BASIC was invented at Dartmouth College in the 1960s to be used by students, beginners, novices, newcomers, to computers and programming. It's very much like everyday English, as you'll see as we go through this book. We'll point out the familiar look-alike words which have meanings you already know and understand. Words like *end, for, go, to, if, then, list, new, next, step, print, return, run, stop,* and others.

Building on what you already know, we'll show you how the computer receives your instructions and uses them to do what you want.

Universal BASIC

We will use what we consider the most-universal form of BASIC, simplified so it is applicable to just about any contemporary computer—large or small. These words, when used to instruct a computer, would be understood by just about any hardware. If you have a computer, other than the pocket computers mentioned here, be sure to

check your owners' manual to see how its BASIC words differ (if they do) from those we use here.

There are many complete programs in this book. Each has been tested on the TRS-80 Pocket Computer, the Sharp PC-1211 Pocket Computer or the Casio FX-702P Pocket Computer. Keep your owners' manual handy as you type in and run these programs. You may need it to make sure you are properly turning on your equipment.

Please remember, no two programmers write identical programs from scratch. Even when working toward the same goal, different writers will create different logic patterns. If your program doesn't exactly match a suggestion in this book, yours still may be correct.

Assuming your program runs and gets the required result, judgment of writing quality should be made on brevity, quickness of running time, and organizational clarity. It's always best to write as few lines as possible. The faster a computer completes its work, the better. And instructions should appear to flow in a logical order so they can be followed by others who might read your writing.

Comparison table

The Appendix to this book includes a large table comparing approximately 175 BASIC words used in the most popular pocket computers. You may find the table useful in looking up information about whether or not your particular computer offers use of a particular BASIC word. The chart also will permit comparison of features if you plan to buy a new pocket computer. This is the first time such a chart has been published. We hope you find it useful.

What's Inside Your Computer?

What's Inside Your Computer?

Suppose you shrink to the size of an electronic mouse, hiding in a corner inside your computer, watching the action. There are four main areas which grab your attention: the *input* keyboard, the tiny *microprocessor,* a great hulking *memory,* and the *output* display.

Processor, input, output and memory are the important parts of any computer. There are many accessory parts and sections but those four are where the most-interesting activity occurs.

Input and output, often abbreviated as I/O, allow a computer to receive work orders from its operator, to receive information or data for use during a work period, and to send out messages and work results to the operator. See figure 1.

Input most often is a typewriter *keyboard* although it could be a microphone to pick up sounds, an electric eye to read printed matter, or other inventions. In today's pocket computers, it is a keyboard.

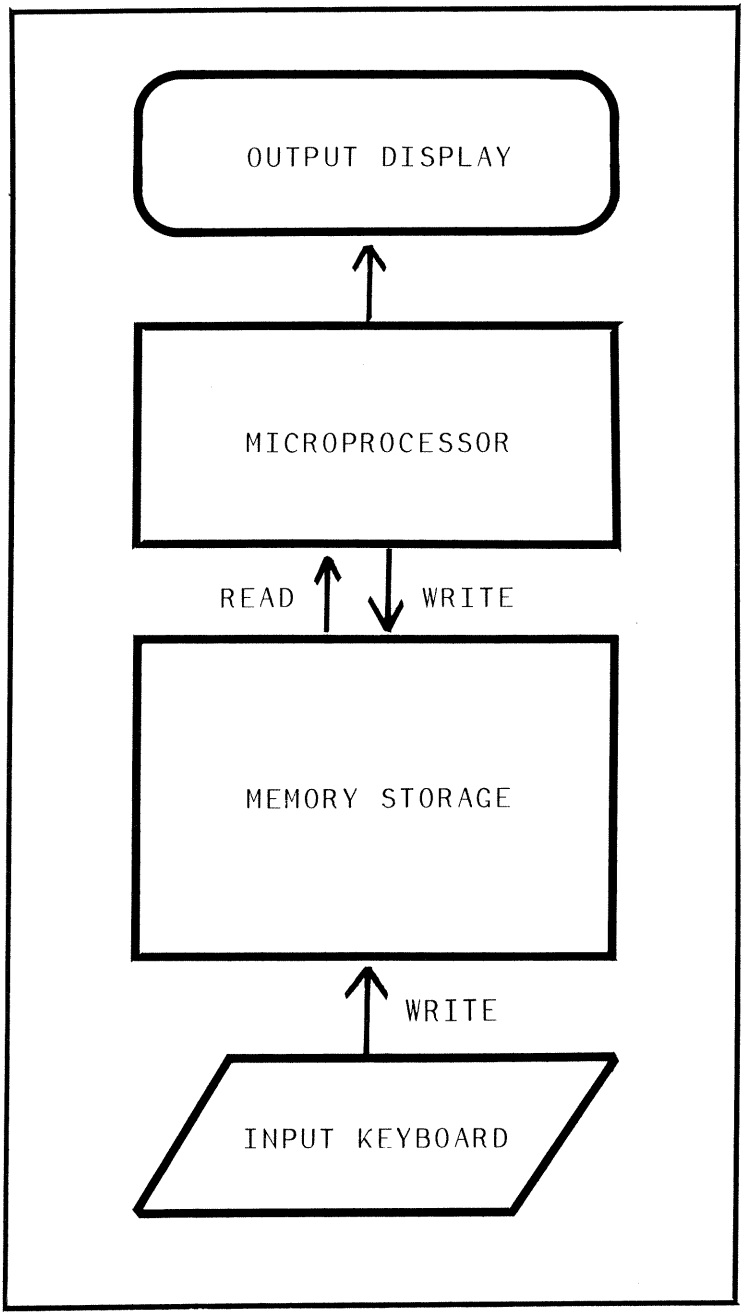


FIGURE 1: COMPUTER

Through the keyboard, an operator gives the computer a list of instructions for carrying out one or more jobs. That list, or *program* of action, is followed by the computer whenever told to do so. It does not have to be acted upon immediately. The program can be remembered for later action.

To achieve some of its work goals, the computer must have additional information or *data.* That information also is typed in through the keyboard.

So, the keyboard has two functions: sending in programs of instructions and sending in additional data.

The output display in pocket computers is a tiny electronic typewriter and an LCD (liquid-crystal display) panel. In other computers it might be a television set or a TV-like *monitor* or a larger electric typewriter. The output display has one main duty: showing messages and work results to you.

Memory

The convenience of a computer would be lost if we had to send in instructions, one at a time, and await action after each instruction. The beauty of the beast lies in its ability to memorize a long list of instructions and then, upon later command, execute those orders. The computer has a memory to store its various lists of instructions. It is called *program memory* and it can hold more than one complete program at a time.

At the same time, things would be slowed considerably if each extra piece of information has to be keyed in repeatedly every time the computer needed it. The computer can accept data one time and then store it away for repeated use later. To keep such extra information on hand, the computer has *data memory.*

Shrunk to an electronic mouse as you are, you see that great hulking memory, stuffed with program lists and bits and pieces of miscellaneous data. Let's take a closer look at data memory.

Our microinspection tour reveals what can be imagined as a large quantity of storage boxes. See figure 2.

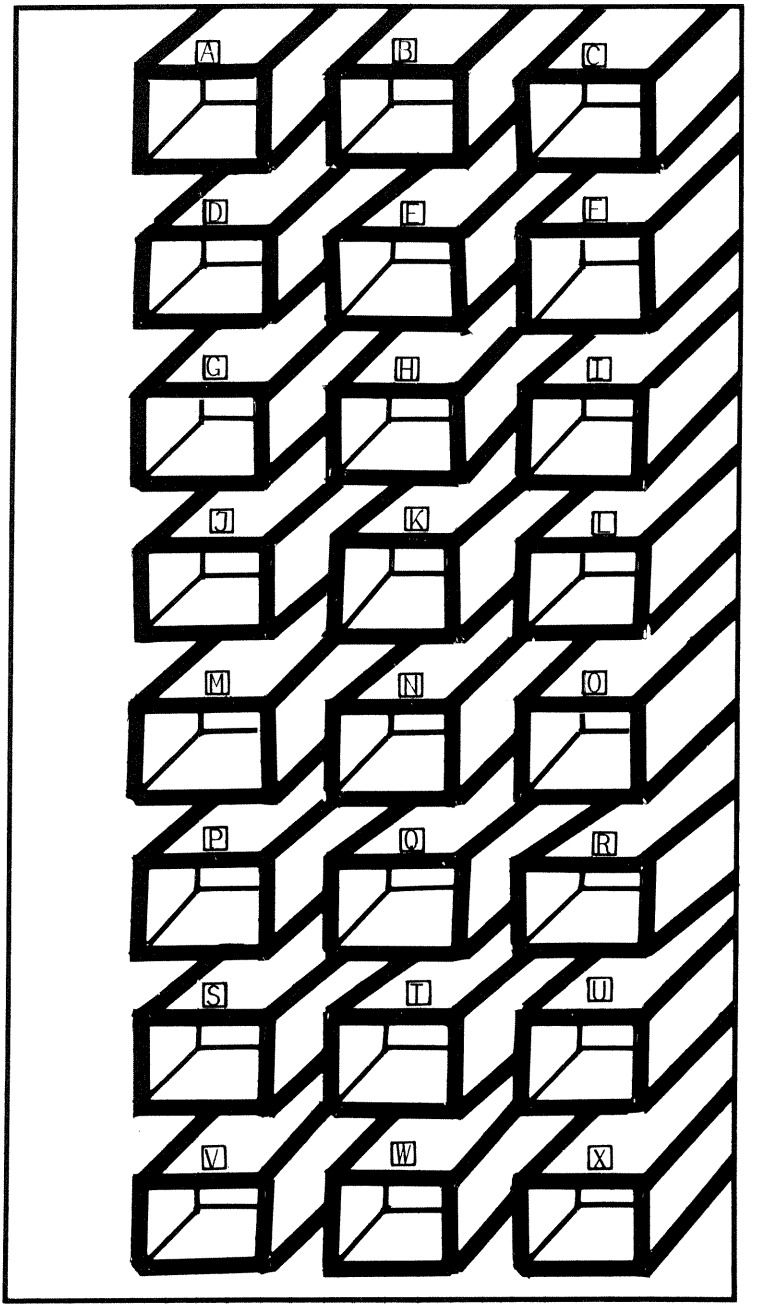


FIGURE 2: EMPTY BOXES

Imagine 26 boxes labeled A through Z. The contents of the boxes can be changed. Some contain something. Some contain nothing. All are *variable* in that their contents can be changed.

Among the boxes containing something, you see some with numbers and some with letters and words.

Consider each box to be a single memory location, identified by its label A or B or C on through Z. Let's look more closely at memory locations A, B and C. See figure 3.

Notice that box A is empty. On the other hand, box B has a word stored in it and box C has a number stored inside. Variable memory location A is empty for the moment while variables B and C are holding information.

Strings

The boxes can contain either *numerical* information or words composed of combinations of letters, symbols and even numbers. Such a word is thought of as a *string* of data. Whenever one of our memory location boxes is storing a word, it is a *string variable.* If it holds only numbers, with no letters or other keyboard symbols, it is a *numerical variable.*

The quantity of letters, symbols and numbers which can be tied together in a string and stored in one memory location is limited. In larger desktop computers, one string in one memory location can hold hundreds of characters. But in our pocket computers, one string is limited to seven characters.

This limitation applies only to string variables, not to numerical variables. Here are some examples of what variables contents might look like:

String Variables	**Numerical Variables**
JIM	86
@#$%ABC	1234567890
1/12/83	22.66
BIRTHDY	1

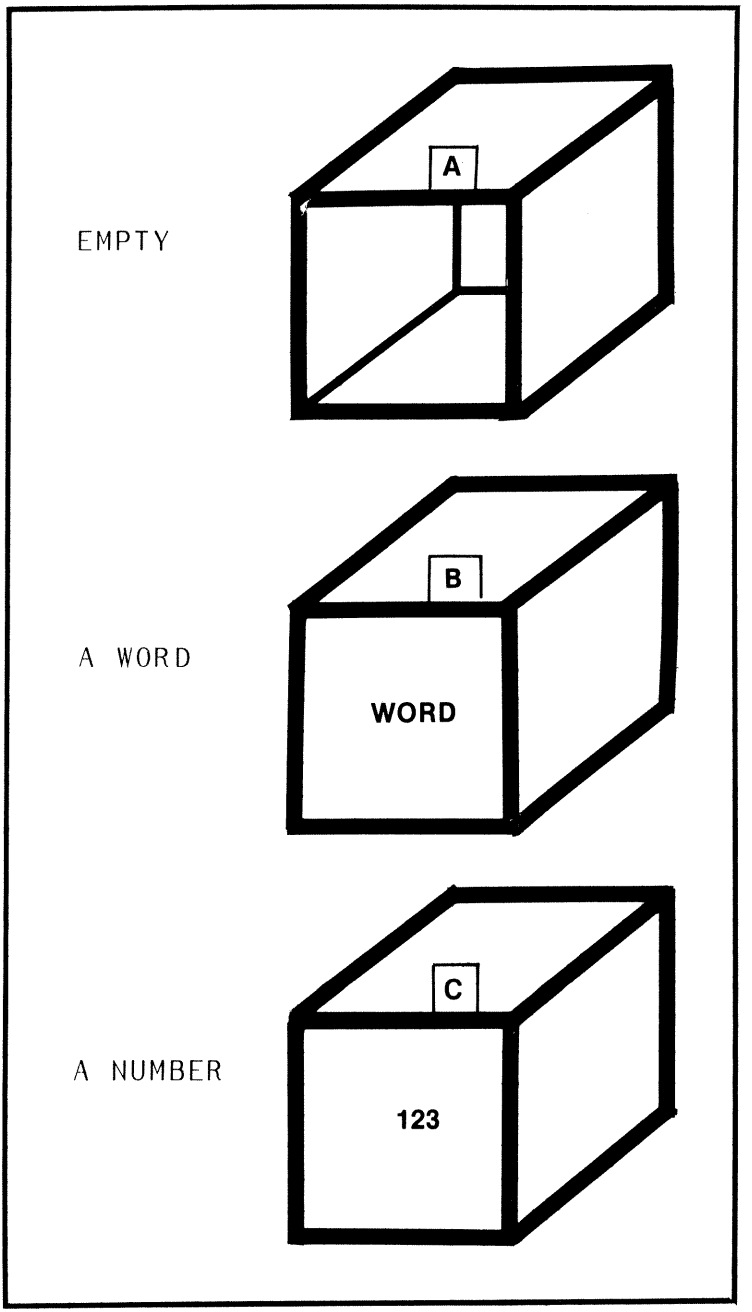


FIGURE 3: A, B, & C

The program writer must keep track of which kind of variable is being used in a particular memory location. For example, if you store a word in location B and then try to use that data in a math problem, an *error* will occur and you'll get a message from your computer.

Only when you have numerical information stored in a memory location can you use that data for math.

One way programmers keep such things straight is by labeling string variables with a dollar sign ($). The dollar sign means *string* and should be read as "string."

Empty boxes

Let's change what we saw in figure 3 to use this new idea. See figure 4.

Memory location B now is correctly labeled B$, since it contains a word, while memory location C has no dollar sign since it contains numerical information.

If we were to put a number in A we would label it A. If we were to put a word in A we would label it A$. A and A$ are the same storage box but you can only keep one kind of data inside at one time.

By the way, you can change the contents of the various boxes during the running of a program. A location can go from empty to full or from full to empty. Or a full location can have its value changed. A program can be written so the computer will continually check memory locations to see what has been stored there.

Obviously, when we say a memory location is empty we mean it has nothing in it. In effect, it has a big fat zero inside. As a matter of fact, if you were to look at the contents of an empty variable, you would see that it contains a zero. If you ask the computer to show you the contents of an empty memory location, the output display will show Ø if it is a numerical variable. If it is a string variable with nothing stored inside, the display will show nothing. Not even a zero. It will be blank.

You write in data memory by setting the data location

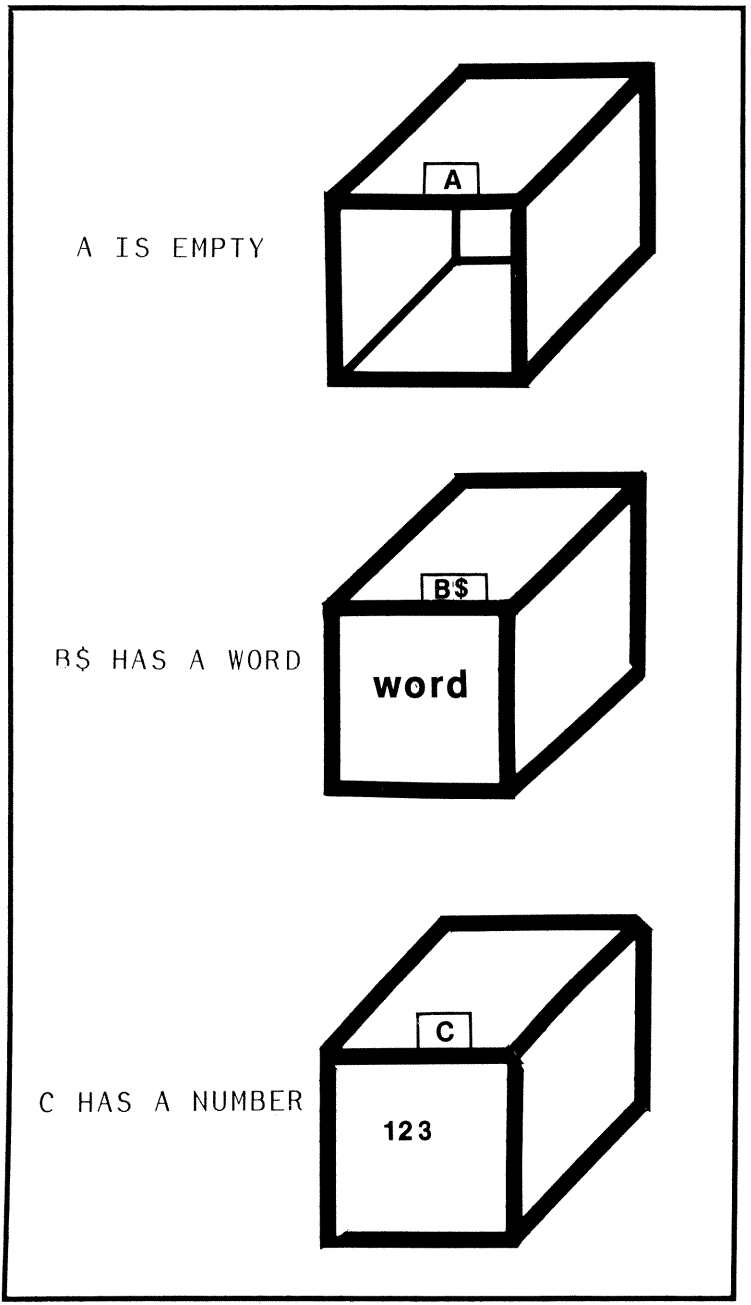


FIGURE 4: B LABELED $

letter equal to the value you want to write in it. For instance:

A = 1234

The value on the right is transferred into the storage location on the left.

Testing

Here's a handy test of memory locations. Turn your pocket computer on. If you are using the TRS-80/Sharp computer, press the MODE button repeatedly until the display says RUN. Type in this line:

A$ = "WORD"

Press the ENTER key and the red CLEAR key. The main display will be blank.

Note: EXE on the Casio keyboard means EXEcute and is the same as ENTER on the TRS-80/Sharp computer. Casio owners should press EXE when we call for ENTER.

Similarly, Casio owners will want to press MODE and the zero key to get into the RUN mode. Pressing the MODE key and the number one will put the Casio in the program-writing mode, the same as PRO on the TRS-80/Sharp computer.

The orange AC key, which stands for All Clear, on the Casio is the same as the red CLEAR key on the TRS-80/Sharp computer.

Okay, at this point you have typed in:

A$ = "WORD"

and pressed the ENTER and CLEAR keys. The main display is blank.

What has been accomplished? You have actually stored the letters WORD in memory location A$ inside your computer. How do you know for sure? In the RUN mode press A$ and ENTER. The string of letters forming WORD magically reappear on the display!

You can CLEAR the display and recall WORD as often as you like. You have successfully stored string data in variable A and recalled it from memory for display.

Now try writing something else overtop of WORD in memory location A$. In the RUN mode, type in:

A$ = "NAME"

Press ENTER to complete the storage action and CLEAR to blank the screen. Now type in A$ and key ENTER. The new word, NAME, reappears. The old string, WORD, has been lost forever.

Now, let's put something in location B. Make it numerical data this time (not string data). Try 1234.56789 which is a nice long entry. Type this in the RUN mode:

B = 1234.56789

Press the ENTER key to complete the storage and CLEAR to remove it from the display.

You have stored the number 1234.56789 in memory location B. Now key in B and press ENTER. The number stored in B will reappear instantly on the display. You have successfully stored numerical data in variable B and recalled it.

What's an A$(20)?

In the TRS-80/PC-1211, memory locations can be specified as A-plus-number. The number can be from 1 through 26. For instance, memory location A(01) is the first A location or, just plain old memory location A.

Count down the alphabet from A. What is the thirteenth letter? It is M. Thus, A(13) is the same memory location as M. What is the twentieth letter of the alphabet? It's T. That means A(20) is the same memory location as T.

If you wish to store string data in one of these, insert the dollar sign between the A and the first parenthesis. For example, A$(02) is the same as B$ or A$(26) is the same as Z$.

Your ability to specify a memory location by number like this will come in handy later. Watch for it.

By the way, more powerful pocket computers such as the TRS-80 PC-2 and Sharp PC-1500 have additional memory storage boxes. These are labeled by combinations of two letters. Desktop computers have even more storage boxes available in data memory.

Flexible memories

So far we have been talking mostly about *data memory* and ignoring *program memory.* We'll get to program memory in a moment. First there's one more thing you should know about data memory locations.

Among data memory locations, some are *fixed* and some are *flexible.* The fixed memory locations are those we have discussed and known as A through Z. They are always there for your use. They cannot be removed.

But what if those 26 storage boxes aren't enough? You can borrow some memory locations from program memory!

Borrowed memory locations are known as *flexible memory.* Flexible memory in the TRS-80/Sharp computer are labeled by the A-plus-number method. Since A(01) through A(26) are the same as A to Z fixed memories, what numbers do you use for flexible memories? They start at A(27) and move up through A(28), A(29), etc., up to A(204). Storing string data in these locations makes them A$(27), A$(28), A$(29), etc.

Each flexible memory used removes some storage capacity from program memory. The more flexible memories used, the shorter the programs which can be stored in program memory.

You have to remember that you are using some program storage space so you don't try to write more programs and erase important data. The computer will always take back flexible memories to use as program memory when you ask it to. So, data in flexible memories could be lost.

Program memory

Now you know how to write information in data memory, and recall it. How about writing in program memory?

To write into program memory you must put the computer into the program-writing mode. That's PRO on the TRS-80/Sharp and MODE 1 on the Casio pocket computer.

When in that mode, anything you key in and ENTER will be stored in the program section of memory.

Your computer is built to use the BASIC program language. BASIC requires each line of a program to start with a *line number.* Here's a typical three-line program. Notice the numbers at the beginning of each line:

```
10 CLEAR
20 A$ = "WORD"
30 PRINT A$
```

The computer needs those line numbers to be able to follow your instructions in sequence. It knows that line 20 comes after line 10 and line 30 comes after line 20. Here's the same program with different line numbers:

```
5 CLEAR
21 A$ = "WORD"
189 PRINT A$
```

This program will run just the same as the first one. The line numbers are in the same sequence and the commands within each line are the same.

To write this program into program memory, type in one line at a time. When you are satisfied that the line is typed on the display correctly, press ENTER to complete the storage in program memory. You can't enter more than one line at a time so press ENTER at the end of each line. Try it:

```
10 CLEAR
```

Now press ENTER. Type in the next line:

```
20 A$ = "WORD"
```

Press ENTER and type in the next line:

```
30 PRINT A$
```

Press ENTER. Now you have stored the complete three-line program in memory.

Now press the CLEAR key to blank the display. The program is sitting in memory but you can't see it. To look at it, use the LIST command.

Type in LIST and press ENTER. The computer will show you line 10. Use the down-pointing arrow key to go down through the line numbers. Press the arrow key once and

you'll see line 20. Press it again and you'll see line 30. Press the up arrow and you go up through the program lines.

In the Casio computer, you don't have the ability to scan up like this. Pressing the EXE key during a LIST will cause the next line in sequence to be displayed.

LIST doesn't work in the RUN mode on the TRS-80/Sharp computer but it does work in the running mode on the Casio.

How much is left?

Anything you store in program memory takes up space and space is limited. You can tell how much space remains open for storage by using the MEM function on the TRS-80/Sharp computer. Type in MEM and press ENTER. With our simple three-line program typed in, the TRS-80 PC-1/Sharp PC-1211 display shows 1402 program steps or 175 memories available for programming.

Suppose you wanted to get rid of the simple three-line program. In the run mode, type in NEW and press ENTER. The NEW command clears *all* program memory at one time. Try it. Type in NEW. Now press ENTER.

Try LIST. You should get nothing since program memory has been completely cleared.

Try MEM. You now should get a message from the computer that it has 1424 program steps, or 178 memories, available for use. Remembering that with the program in memory there were only 1402 steps left, you can see the program used up 22 steps of program capacity.

It is possible to write a program which uses every single step of program memory!

CLEAR vs. NEW

The command NEW erases everything stored in program memory, no matter how many different programs you have there.

To erase all data memories, use the CLEAR command. It will get rid of all information in all data memories.

Casio note: VAC in the FX-702P is the same as CLEAR in the TRS-80/Sharp computer. CLR is the same as NEW in the program-writing mode. CLR will clear the particular program you are working in. CL R ALL will empty all programs from program memory. Similarly, LIST displays only the program you are working on while LIST ALL shows all programs in program memory.

The processor

Be an electronic mouse inside the computer again. Notice the master-controller in charge of everything. That's the microprocessor. *Micro* means small. *Processor* means it follows instructions in manipulating data to do work. It's not very big but it sure is powerful!

The processor is a very logical worker, dutifully going about its business in a proper order, carrying out instructions, doing work.

Built into the processor are instructions for how to handle its chores. As it follows that internal set of instructions, it knows how to follow your external set of instructions and do the work you want done.

To make a long story short, the processor takes information from memory, does something with it, and then either returns data to memory or displays it as output for you to read. It is able to do this many, many times each second and that's why we love the microprocessor!

Suppose you tell the microprocessor to fetch the contents of memory location B. It looks in there and finds WORD there. It *reads* that word, leaves the original behind in memory location B, and takes the information about what is in B away to work with it. The processor actually has a tiny memory inside itself so it can remember what it read in B.

If we instruct the processor to store something in memory location C, it *writes* data to that memory location. When it writes in that memory location, it destroys whatever was there before. For example, suppose we have the number 1234 stored in memory location C. As a result of an operation, we instruct the microprocessor to

store the number 6789 in memory location C. It will put 6789 into C and we will lose the original number, 1234, forever.

Remember: reading destroys nothing but writing replaces old information with new.

In carrying out activities, the processor follows exactly the set of instructions you gave it as a *program.* It can't do anything else. If you make a mistake, it makes a mistake. If your work was perfect, its work will be perfect.

Program language

A program is composed of sets of alphabet letters which the processor understands as *words.* A complete set of such words makes up a *language.* BASIC is a language composed of words such as GO, TO, FOR, NEXT, IF, THEN, STEP, PRINT, RETURN, INPUT, PAUSE, WAIT, SET, STOP, END, SAVE, LOAD, GET, PUT, RUN, LIST, NEW and many others.

Since our pocket computers are so very small, they have been given only the very best, most useful, of these words. The TRS-80/Sharp computer has all the most important words. The Casio has a few more. Larger desktop computers have even more. The Radio Shack TRS-80 PC-2 Pocket Computer and Sharp PC-1500 Pocket Computer each have an extensive vocabulary, similar to the largest desktop computers. There are very few words not in their vocabularies.

The more extensive the BASIC vocabulary, the more flexible the writer can be in creating programs. The total number of BASIC words invented to date is well over 500. You have the best three dozen of these in your pocket computer.

It's easy to see why BASIC is the most popular computer language today. It's most like everyday English and, therefore, most readily used.

Writing and Running Programs

Writing and Running Programs

Writing programs means creating line lists of instructions and storing them, one at a time, in program memory.

Running means having the computer recall those sets of instructions, one line at a time, and do them.

Here's a simple one-line program. Put your computer in the program-writing mode (as explained earlier) and type in this line:

10 PRINT "WORD"

When you are satisfied that you have the line correct on the display, press ENTER. That completes the act of storing it in program memory. Now you can press the CLEAR key to remove the line from the display and use the LIST command to recall it.

Switch to the RUN mode and type in RUN and press ENTER. What is on the display?

WORD

Your one line program instructed the computer to print the word, WORD, and it did just that. Now return to PRO

mode, type in NEW and ENTER to get rid of the one-line program, and type in this new program:

```
10 CLEAR
20 PRINT "WORD"
```

Go back to the RUN mode and run it. What did you get as a result?

```
WORD
```

The same result. Go back to the PRO mode and type in this three line program:

```
10 CLEAR
20 A$ = "WORD"
30 PRINT A$
```

In the RUN mode, what is the result of a run?

```
WORD
```

Again the same result. Each of these programs looks different but each has the same final result: printing the letters WORD on the computer's display.

Here's a more-elaborate program with six lines. These lines will enhance the work result by making a sentence:

```
10 CLEAR
20 A$ = "WORD"
30 B$ = "THE"
40 C$ = "IS"
50 D$ = "THIS"
60 PRINT B$;A$;C$;D$
```

Here's the result of running this program:

```
THEWORDISTHIS
```

If you had typed in a blank space at the end of each English word the sentence would have been properly spaced. For instance, if you had typed line 20 like this:

```
20 A$ = "WORD "
```

the result would have looked better.

```
THE WORD IS THIS
```

Now you know what a BASIC-language computer program looks like. Such programs range from one-liners to lists of hundreds of lines. The pocket computer easily can store a 100-line program.

ENTER

Whether you are trying to put program lines into program memory or general information into data memory, the computer won't memorize typed-in info until the ENTER key is pressed. The computer's display will echo what you keyed in, showing what you have typed, but the act of storing in memory isn't complete until you press ENTER. Try this test. Type:

WORD

Press the CLEAR key. The letters WORD disappear. How can you get them back? You can't since you didn't tell the computer where to store them.

In fact, the TRS-80/Sharp pocket computer thinks you want it to multiply W times O times R times D when you type in WORD and press ENTER. If you had stored numbers in those locations, the result of multiplication would have been displayed.

Remember, you have two places to store information. Program memory for program lines with line numbers. And data memory for information labeled for one of the memory locations. To write in program memory, put the computer in the PRO mode, key in a line number, and type in WORD. Here's an outline of the action.

Your Action	**Computer Displays**
type **10 WORD**	**10WORD**
press **ENTER**	**10: WORD**
press **CL**	blank display
type **LIST**	**LIST**
press **ENTER**	**10: WORD**

What's happened? You have entered an instruction into program memory, erased the display, and recalled the program line to the display. Now press the RUN key repeatedly to get to the RUN mode:

Your Action	Computer Displays
type **WORD**	**WORD**
press **ENTER**	**Ø**

What's happended? The letters WORD have not been stored anywhere and when you tried to recall them you got a zero. Now, we know WORD is a string. Let's turn memory location A into a string location and put WORD in it. Follow this action:

Your Action	Computer Displays
type **A$ = "WORD"**	**A$ = "WORD"**
press **ENTER**	**WORD**
press **CL** key	blank display
type **A$**	**A$**
press **ENTER**	**WORD**
press **CL** key	blank display
type **A$**	**A$**
press **ENTER**	**WORD**

You have stored WORD as a string in data memory, in location A to be exact, and erased and recalled it repeatedly.

RUN

Let's put an instruction in program memory and then switch to the RUN mode and run it. First, press the MODE key repeatedly to get to the PRO mode. Type in NEW and press ENTER to get rid of all old programs from program memory. Then do this:

Your Action	Computer Displays
type **10PRINT"WORD"**	**10PRINT"WORD"**
press **ENTER**	**10: PRINT "WORD"**
press **MODE** key 3 times to **RUN**	blank display
type **RUN**	**RUN**
press **ENTER**	**WORD**
press **CL**	blank display
type **RUN**	**RUN**
press **ENTER**	**WORD**

You have successfully stored the string WORD in program memory in a command to print it when called for. Then, in the RUN mode, you called upon the computer to run the program. It did and the result was correct. The letters WORD were printed on the computer display.

RUN is an instruction to the computer to start at the lowest program line number and begin executing commands it finds there.

You can make the computer start its run at a different line number by typing that line number immediately after the word RUN, before ENTER. For instance, to start at line 100, type:

RUN 100

and then press ENTER. The computer will skip over any program lines with numbers less than 100.

Similarly, in the TRS-80/Sharp computer, you can tell the computer to start at a particular line number by putting a special label at the head of that line and asking the computer to RUN starting at that label. Suppose you had this three-line program in your computer.

```
10  CLEAR
20  PRINT "NAME"
30  "X" PRINT  "WORD"
```

To make the computer skip over lines 10 and 20 and start its run at line 30, type in

RUN "X"

and press ENTER. The computer will search for a line starting with "X" such as line 30 in our program here. When it finds such a line it will start its run.

REMarks

Suppose you were to write a very long, 50-line program of instructions for your computer. You might forget what each line was to accomplish. You need some way to put information in program memory which won't be acted upon by the computer during a run. Information such as notes to yourself so that when you list your program you can recall what the various parts of the program were supposed to do. These notes to yourself, and for other programmers to read, are called *remarks.* The REM command is used. Anything in a program line after REM will be ignored by the computer during a run. For example:

```
10  REM PRINT "NAME"
20  PRINT "WORD"
```

Type in this program and run it. You'll see that the computer has ignored, or skipped over line 10 and done line 20. Anything on a line after REM is ignored. Here's something different. Try this one:

```
10 PRINT "WORD": REM PRINT "NAME"
```

Here the program did the first part of line 10 but ignored the last half of the line following REM.

By the way, you can put multiple statements in one program line by using the colon (:).

The colon indicates to the computer that a new statement is coming. Thus, you can place several statements in one line if you wish. Separate them with colons. Here's an example of a one-line program including several statements:

```
10 PRINT "WORD":PRINT "NAME": PRINT "DOG"
   :PRINT "CAT"
```

The computer will follow these statements of command in sequence as it reads through line 10. It will print WORD first, then NAME, then DOG, then CAT.

REMarks are good for notes but very wasteful of memory. And we don't have much memory to spare in the pocket computer. Use REM infrequently!

BREAK

What to do when your computer goes *blitzo!*

In the PRO mode, type in this sample program:

```
10  PAUSE "WORD"
20  GOTO 10
```

PAUSE means display momentarily and then go on. GOTO means go to somewhere, in this case line 10. This small program creates a *loop*. First the computer does as it is asked in line 10. It momentarily prints on the display the letters WORD. Then it goes on to line 20. At line 20 it finds a command to go back to line 10. At line 10 it momentarily prints WORD and goes on to line 20. This will cycle and recycle forever until you break the loop. Use the BREAK key.

Note that BREAK is also the ON key in the TRS-80/Sharp keyboard and the STOP key on the Casio keyboard.

Let's run our two-line loop program. Put your computer in PRO mode and type it in. ENTER each line. Then switch to RUN mode and run it.

Notice that the computer is continually displaying, erasing, displaying, erasing, displaying, etc., as it loops through its instructions. Now press BREAK. The display says:

```
BREAK AT 10
```

or

```
BREAK AT 20
```

depending upon which line it was doing when you pressed BREAK.

BREAK is used whenever you need to stop a RUN dead in its tracks. It's your panic button.

STOP

But suppose you want the program to STOP automatically at some point in a run? Use the STOP command. Write it into your program as one line.

How to continue after STOP? Use CONT. Try this sample program:

```
10 PAUSE "WORD"
20 STOP
30 GOTO 10
```

Run it. The computer prints the letters WORD and then stops, displaying the message BREAK AT 20. To make the computer go on to line 30, type in CONT and press ENTER. The computer will proceed from where it left off. In this example, it will go on to line 30. At line 30 it finds an instruction to go back to line 10. At line 10 it momentarily prints WORD and then goes on to line 20. At line 20 it finds a STOP command so it quits and displays the message BREAK AT 20. And so on. STOP stops it. CONT makes it continue.

END

You can, at your option, tell the computer a program has ended. Use the END command. Try this sample program:

```
10 PAUSE "WORD"
20 END
30 PAUSE "NAME"
40 GOTO 10
```

Upon running this program, the computer will do line 10, printing WORD on its display. When it moves on to line 20 it will find an END statement so it will think it is at the end of the program. It ceases operations and goes to a blank display.

If you were to start at line 30 with a RUN 30 command, it would first print NAME, then print WORD and then stop.

The CONT function won't work after an END command.

Input and Output

Input and Output

Input means giving the computer something to store in memory, whether data or program.

Output means displaying messages and work results for you to see.

INPUT

Information can be permanently placed in memory when you write a program. That is, data will actually be part of the program as written. This fixed information could look like this:

```
10  A$ = "WORD"
```

Whenever you run the program the computer will always start with the memory that WORD is the data in memory location A$.

But, suppose you want the computer to pick up changeable data during a run? Use the INPUT function. Try this program:

```
10  A$ = "IT IS"
20  INPUT "WHAT IS THE WORD",B$
30  PRINT A$;B$
```

When you run this program, the computer starts at line 10 and stores the string IT IS in memory location A. At line 20, the computer displays the question, WHAT IS THE WORD, and waits for a reply. You type in any string of up to seven characters in reply. Press ENTER to give your answer to the computer. The computer stores your answer in B$. Then, the computer moves on to line 30 where it recalls the contents of memory locations A$ and B$ and prints them on the display.

Let's see how INPUT works when you want to collect numerical data. It works the same. Try this short program:

```
10  Q = 111
20  INPUT "PICK A NUMBER",N
30  R = Q + N
40  PRINT N;"PLUS ";Q;" = ";R
```

Here, line 10 puts the value 111 into memory location Q. Line 20 displays the message, PICK A NUMBER, and awaits your response. Whatever number you select, key it in and press ENTER. The computer will store your number in memory location N.

Line 30 does the math work for you by adding. It recalls that 111 was stored in Q and your number was stored in N. It adds those two values to get a new total. The total is stored in memory location R. The program moves on to line 40.

At line 40 the computer prints the results in sentence form. Try it with several different numbers. It's fun!

Suppose your number were 59. The program result, after printing line 40, would look like this:

```
59  PLUS 111  =  170
```

You don't have to use the message part of the INPUT function if you don't want to. For instance:

```
10 INPUT N
20 INPUT P
30 PRINT N
40 PRINT P
```

This program allows the computer to take in your numerical data and store it in memory locations N and P and then print the values on the display. The computer will start at the lowest line number, as usual, line 10. Since no message has been supplied, the computer will display only a question mark (?). The ? tells you the computer wants some information. Try it on your pocket computer.

Casio users note: use the INP key for INPUT and PRT for PRINT. Your program would look like this:

```
10 INP N
20 INP P
30 PRT N
40 PRT P
```

Casio users also have available the powerful KEY function which we'll discuss later in a special section devoted to the FX-702P.

PRINT

You already have used the PRINT output command but here's some further information.

PRINT and PAUSE are output functions of the TRS-80/Sharp computer. Similar Casio commands are PRT and WAIT.

PRINT in the TRS-80/Sharp is the same as PRT in the Casio. PAUSE in the TRS-80/Sharp is the same as the combination of WAIT and PRT in the Casio. Here's how they work:

PRINT causes a message to be displayed on the pocket computer's display. The printed message consists of whatever is contained within the quotation marks follow-

ing the PRINT command. For instance:

```
10  PRINT "I LIKE ICE CREAM"
```

The computer reproduces exactly what you place between the quotes, including blank spaces. Try it in your pocket computer. Now, type in this program:

```
10  PRINT "I LIKE ICE CREAM"
20  PRINT "DO YOU?"
```

Run it. Notice that the computer stops after printing the message, I LIKE ICE CREAM. Why? Because the PRINT command in effect contains a built-in STOP command.

How can you make the computer show its message and then go on to the following program lines? Use PAUSE in place of PRINT. Make this change:

```
10  PAUSE "I LIKE ICE CREAM"
20  PRINT "DO YOU?"
```

Now when you run the program, the computer shows the message, I LIKE ICE CREAM, for about a second and then replaces it with the line 20 message, DO YOU?

If you are a Casio user, you'll note that the TRS-80/Sharp computer's PAUSE combines two of your commands, WAIT and PRT. Your version of the same program would look like this:

```
10  WAIT 20
20  PRT "I LIKE ICE CREAM"
30  PRT "DO YOU?"
```

Line 10 tells the Casio to display all upcoming PRT messages for about one second. Line 20 contains the message, I LIKE ICE CREAM. The combination of instructions in line 10 and line 20 is the same as the single-line command PAUSE in the TRS-80/Sharp computer. And line 10 in the Casio will effect line 30 in the same way.

These PRINT messages need not be in the same line as the PRINT command, by the way. Rather, you can store a

message in data memory and recall it for PRINTing. For example:

```
10  N = 1234.56789
20  PRINT N
```

The computer, at line 10, stores the number 1234.56789 in memory location N. At line 20, the computer recalls the value of N and prints it on its display. Here's another example:

```
10  G$ = "WORD"
20  PRINT G$
```

Here the computer stores the string of characters, WORD, in memory location G. At line 20 it recalls G$ and prints it.

Here's an even more complex program:

```
10  A = 6
20  B = 7
30  C = 2
40  D = A + B + C
50  PRINT D
```

The computer stores the number 6 in memory location A; the number 7 in location B; and 2 in C. At line 40 it recalls the values in A, B and C and adds them together. The result of that addition is stored in D. Line 50 recalls the contents of D and prints the number on the display. Try it.

Remember the *loop* we created earlier? Here's a different one to try on your pocket computer:

```
10  N = N + 1
20  PAUSE N
30  GOTO 10
```

What results? Your computer can count! It starts at one and counts upward endlessly, one at a time, until you press the BREAK key. Here's how it works.

Line 10 recalls the value stored in memory location N and adds one to it. Since we are just starting our run, there is nothing, or zero, stored in N. Thus, one added to zero equals one. The result of the addition is stored in N.

Line 20 recalls the current value stored in N which, for now, is a one. Line 20 causes that number one to be displayed for you to see. The computer advances to line 30 where it finds a command to go back to line 10.

When it gets to line 10 this time it finds a one stored in N and adds one to that. The new total, two, is stored in N, erasing the old total. The program goes on to line 20. At line 20, the computer displays the two now stored in memory location N.

Line 30 pushes action back to line 10 again. At line 10 the computer finds a two in N and adds one to it. The new total, three, is stored in N, erasing the old total. And so on. The output result is a continuing upward count on the computer display. Try it.

This looping action will continue until you tell it to stop by pressing BREAK. Your computer can count. Lots of fun!

CLEAR

Make your computer count up to 50 and then, quickly, press the BREAK key so it stops at 50. The operation will stop. Now type in RUN and press ENTER. Where does the count start? It picks up at 51!

Why? Because the data memory held 50 in location N even though the computer stopped running. When you started a new run, N still held 50 so line 10 added a one to that 50 and got a new total of 51. Line 20 printed that value, 51.

How can you make a count automatically restart at zero? Try this program:

```
10 CLEAR
20 N = N + 1
30 PAUSE N
40 GOTO 20
```

It's the same program except we've added a CLEAR command at the beginning. And the GOTO statement in line 40 makes action jump back to line 20.

When the program starts running at line 20, the CLEAR command erases *all* data memories. So, when you let it count up to 50, BREAK to stop it, and then restart the run, it will pass through line 10 at restart. Line 10 will CLEAR data memory N and, in effect, put a zero in it, erasing the 50 which had been in N. When the computer looks in an empty or CLEARed memory location it sees zero. CLEAR in this program causes all new starts to be from zero. Why is line 40 set to make the program jump to line 20 rather than line 10? Try this change:

```
40 GOTO 10
```

You see a continual display of ones. That's because jumping to 10 puts CLEAR in the loop everytime with N always being reset to zero. The value in N doesn't get a chance to increase beyond one.

Casio users note: use VAC rather than CLEAR in your programs. Yours should look like this:

```
10 VAC
20 N = N + 1
30 WAIT 20
40 PRT N
50 GOTO 20
```

Let's make a novelty "alphabet counter." Write a program so the computer will "count down" through the alphabet by displaying A, then B, then C, then D, then E, etc. Try this:

```
10 PAUSE "A"
20 PAUSE "B"
30 PAUSE "C"
```

```
 40 PAUSE "D"
 50 PAUSE "E"
 60 PAUSE "F"
 70 PAUSE "G"
 80 PAUSE "H"
 90 PAUSE "I"
100 PAUSE "J"
110 PAUSE "K"
120 PAUSE "L"
130 PAUSE "M"
140 PAUSE "N"
150 PAUSE "O"
160 PAUSE "P"
170 PAUSE "Q"
180 PAUSE "R"
190 PAUSE "S"
200 PAUSE "T"
210 PAUSE "U"
220 PAUSE "V"
230 PAUSE "W"
240 PAUSE "X"
250 PAUSE "Y"
260 PAUSE "Z"
270 GOTO 10
```

Fun, isn't it? A long program with 27 lines, it causes the computer to print momentarily each letter of the alphabet. When the computer reaches line 270, it finds a command to go back to line 10 and start the whole thing over. For more fun, change line 270 like this:

```
270 BEEP 5
280 GOTO 10
```

The BEEP command is a very special form of pocket computer output that very, very few other computers have. Sound cues can be especially helpful in long program runs. If you program your computer to make a sound, you will be alerted when the program has reached an important stage or when a run is complete. You won't have to

sit and look at your computer to make sure you miss nothing.

The BEEP command is followed by a number telling the computer how many times to repeat its sound. For example, BEEP 5 causes the computer to make the same sound five times before going on. BEEP 100 would cause the computer to make its sound 100 times before going on. There are lots of uses you can imagine for the BEEP function.

The Real Computer Power!

The Real Computer Power

When folks talk about a computer having power, they often are referring to its ability to make decisions. And its looping ability. And its jumping ability. These capacities, when combined, make for some very powerful computing ability.

FOR/NEXT/STEP

You already know *loops* are fun but we need a way to control them to put them to a useful purpose. Here's one way:

```
10  FOR L = 1 TO 100
20  PAUSE L
30  NEXT L
40  PRINT "END OF COUNT"
50  BEEP 5
```

Lines 10 and 30 create a FOR/NEXT loop. A FOR/NEXT loop probably is the most frequently used of the super-power BASIC commands.

In this program, line 10 actually contains a built-in counter which advances the value stored in L by one every time the program reaches line 30. In fact, until the count reaches 100, line 30 causes the program to jump back to line 10. When the value in L reaches 100, then and only then will the FOR/NEXT loop let the action drop on down to line 40. Here's a variation:

```
10 FOR A=10 TO 100
20 PAUSE A
30 NEXT A
```

The memory location used in the loop can be any of those available to you in your computer. The count can start anywhere and go up to 999 in the TRS-80/Sharp pocket computer. In this program, the value in A will start counting at 10 and loop up to 100.

Unless you tell it otherwise, the count will step up by ones. Try this change:

```
10 FOR X=2 TO 40 STEP 2
20 PAUSE X
30 NEXT X
```

Here the count goes up by twos. Try this program to make the computer count down by ones:

```
10 FOR J=100 TO 1 STEP -1
20 PAUSE J
30 NEXT J
```

The computer starts at 100 and counts down to 1, and then stops. Very convenient. Very powerful!

The STEP statement is not used unless you want increments other than +1. Minus numbers after STEP will cause the computer to count down in numbers while positive numbers will cause it to count up. Now make the computer take some giant steps:

```
10 FOR R=999 TO 1 STEP -100
20 PAUSE R
30 NEXT R
```

The computer counts down by hundreds. At that rate, it doesn't take very long to run out of numbers.

Sometimes you need a time delay in the middle of a program as it is running. The loop can be used to create such a time delay. Type in this program:

```
10  FOR N = 1 TO 999:NEXT N
20  BEEP 1
```

Get a stopwatch and keep an eye on the running time for the program. Line 10 is a FOR/NEXT loop all on one line, without any output during the loop. The computer merely counts internally up to 999 and then moves on to line 20 where it BEEPs. How long does it take such a loop to run its course? Use a stopwatch to time it from RUN/ENTER to BEEP.

How long did it take to get to 999? Over four minutes? A nice long delay! Now try counting to 100:

```
10  FOR N = 1 TO 100:NEXT N
20  BEEP 1
```

How long is the delay before the BEEP? Just over 25 seconds is average. If yours took less time, your computer runs faster than ours. If it took a longer time, it runs slower. Except to purists, it doesn't matter much in the pocket computer. We trade speed for handheld portability!

```
10  FOR N = 1 TO 10:NEXT N
20  BEEP 1
```

Counting only to 10 reduces the delay to about 2.5 seconds in our pocket computer. How about yours?

```
10  FOR N = 1 TO 5:NEXT N
20  BEEP 1
```

Counting only to 5 makes things happen even more quickly. It reduces delay to about 1.5 seconds in our computer. Now, for the one-second delay:

```
10  FOR N = 1 TO 3:NEXT N
20  BEEP 1
```

In our computer, counting to three and beeping takes exactly one second. Experiment with yours. It may take a longer or shorter count to make the loop last one second.

Why is a one-second loop useful? Well, maybe you would like to turn your pocket computer into a clock!

Here's a simple timer, for starts:

```
10 CLEAR
20 T=T+1
30 FOR N=1 TO 3:NEXT N
40 PAUSE T;" SECONDS"
50 GOTO 20
```

This is a crude clock. You can adjust its speed by changing the number 3 in line 30. It will count seconds until you stop it with the BREAK key.

Can you figure out why it takes a bit longer for the first display, 1 SECONDS, to appear? Because the computer uses up time as it works it way through lines 10, 20 and 30. You planned on it using up time at line 30 but you may have overlooked the amount of time it takes to carry out the instructions at line 10 and line 20.

IF/THEN

Did we say earlier the computer has the ability to make decisions? Yes! The IF/THEN statement is an important part of the decision-making process.

IF something happens or is true, THEN and only then will something else happen. IF nothing happens or something is not true, THEN nothing will happen. The IF/THEN test is one of the superpowers of the pocket computer.

Here are examples of typical IF/THEN program lines:

```
IF A=222 THEN PRINT A
IF B$="DOG" THEN 200
IF J=A/2 THEN PRINT J
IF Q$="WORD" THEN INPUT X$
IF T=2*4 THEN 900
IF A$ PRINT B$
```

IF something is true, THEN some action is taken. That action can be a GOTO jump to a new program line. Or a PRINT command. Or an INPUT or any of the many BASIC statements.

Try this simple program in your pocket computer:

```
10  A$ = "DOG"
20  B$ = "BONE"
30  IF A$ PRINT B$
```

The computer first stores string data DOG in memory location A and then BONE in B. Line 30 then causes the computer to examine location A and make a decision. The phrase IF A$ means "if there is anything in A$" then do whatever comes next in the same line.

In this case, we place DOG in A$ so we know the computer will find something there. Finding something there, it goes on to the last part of line 30 and carries out the specified action. It prints BONE on its display. If it would have found nothing there, it would have ignored the last half of line 30.

That was a simple test, merely to see if there happened to be anything in location A. Now let's change the program to make a harder test for the computer:

```
10  A$ = "DOG"
20  B$ = "BONE"
30  IF A$ = "BONE" PRINT A$
40  IF B$ = "BONE" PRINT B$
```

As before, line 10 stores DOG in memory location A$ and BONE in B$. Having done that, the computer moves on to do line 30.

At line 30, it finds an instruction from you to do a test and make a decision. The test is to look at the contents of A$ and see if they are BONE. If, and only if they are BONE, then go on to the last half of line 30. The last half of line 30 calls for the computer to recall the contents of A$ and print them on the display.

We know we stored DOG in A$. When the computer checks A$ it finds DOG, not BONE. Therefore, it uses its decision-making ability to proceed to line 40 rather than do the last half of line 30. It found that the IF A$ = "BONE" was not true so it could not go on to the last half of that line.

Since the line 30 test failed, the computer moved on to line 40. At line 40 it finds another test. It follows orders and checks the contents of B$. At B$ it finds BONE so the idea that B$ = "BONE" is true. With that found to be true, the computer decides to go ahead with the action called for in the last half of line 40. It recalls BONE from B$ and prints it on the display.

To recap, we stored DOG in A$ and BONE in B$. We asked the computer to print the word DOG if it found the word BONE in A$. It looked and did not find BONE so it did not print DOG. Then we asked it to print BONE if it found the word BONE in B$. It looked at B$, found BONE, and printed BONE on the display.

GOTO

You know that the computer does your list of BASIC instructions by following line numbers. First it does line 10, then line 20, etc. But, suppose you want the computer to do things in a different order. Maybe you would like it to *jump* over a group of lines. Or skip down to a different part of the program. This ability to *branch* out and around some lines to do other lines is an important power in the computer. It involves the GOTO and GOSUB statements.

GOTO means "go to a line." The GOTO statement must include the destination where you wish the program to go. For example:

```
GOTO 100
```

When the computer finds a GOTO statement, it immediately leaves the list, searches for and finds the destination line, and reenters operations at that point. Here's a small example:

```
10 GOTO 30
20 PRINT "NAME"
30 PRINT "WORD"
```

In this program, the computer starts at line 10 where it immediately finds a command to GOTO line 30. It skips down the list until it finds line 30. At line 30 it resumes doing what you asked. It prints WORD. In this case, the instruction in line 20 never gets done.

You can jump backward and forward within the program. Here's an example:

```
 10  INPUT "ENTER A NUMBER",A
 20  INPUT "ENTER ANOTHER NUMBER",B
 30  GOTO 100
 40  PRINT"THE TOTAL IS ";T
 50  GOTO 10
100  T=A+B
110  GOTO 40
```

Again the program starts running at the lowest line number, line 10. At line 10 it asks you for a number which it stores in memory location A. At line 20 it asks for another number which it puts in B.

At line 30 it finds an order to branch down to line 100 which it does. When it finds line 100 it does the instruction in line 100. It recalls the contents of A and B and adds them together, storing the total in T. Having completed line 100, it moves on down to line 110.

At line 110 the computer finds your instruction to jump back up to line 40. Doing that, it finds at line 40 an instruction to print THE TOTAL IS and the value in T. Putting that message on the display, it goes on to line 50.

At line 50 it comes upon your command to go up to line 10. It does that, thereby starting the entire process over again. The computer will go through this elaborate loop as long as you are willing to keep giving it numbers.

GOTO is, in fact, one of the most-used words in the BASIC language. Our programs are strewn with such jumps. Here's an eight-line program which includes five jumps!

```
 10  INPUT "ENTER A NUMBER",N
 20  IF N > 100 THEN 200
 30  IF N < 50 THEN 100
 40  GOTO 10
100  PAUSE "NUMBER LESS THAN 50"
110  GOTO 10
200  PAUSE "NUMBER MORE THAN 100"
210  GOTO 10
```

Lines 20, 30, 40, 110 and 210 contain GOTO jumps. The program is a mini-sorting operation. It sorts out numbers greater than 100 or less than 50 and rejects all others. Here's how:

You give the computer a number at line 10 and it stores the number in N. At line 20 it recalls the value stored in N and tests to see if it is greater than 100. If it is greater than 100, the computer does the last half of line 20 which calls for a jump to line 200. The GOTO word does not actually have to be present in an IF/THEN statement. GOTO is understood between THEN and the destination line number.

If line 20 finds your number not to be more than 100, the computer goes on to line 30 where it tests your number again. This time it tests to see whether or not your number is less than 50. If it is less than 50, the program does the last half of line 30. It jumps to line 100. If your number is not less than 50, the program drops down one notch to line 40.

Line 40 is a *trap* to catch all numbers which fail the tests in line 20 and 30. If a number fails line 20 and line 30, program action drops to line 40 where it is sent via the GOTO statement to line 10 for a new start. In effect, the computer is accepting numbers from you which are greater than 100 or less than 50 but doing nothing if your number is in the range of 50 to 100.

But what happended when your number passed the test in line 20? The computer then did the last half of line 20 and jumped to line 200.

At line 200 it found an instruction to display briefly the message, NUMBER MORE THAN 100. Having completed that display, the next instruction in line 210 was a jump back up to line 10 for a new start.

And what if the number failed the line 20 test but passed the line 30 test? The last half of line 30 caused the operation to jump down to line 100 where the computer briefly displayed the message, NUMBER LESS THAN 50.

At line 110 it found a GOTO causing a jump back up to line 10.

GOSUB/RETURN

Often you will need to repeat the exact same set of instructions at different points in a program. You could type the required program lines into the program each time they are needed. Or you can type them once and make the program jump to them when needed.

Typing of repeating sequences wastes your program-writing time, and, more importantly, wastes program memory space. It's easier for you and uses less memory when you create one *subroutine* to be repeatedly used by the computer.

Why not use a GOTO statement to get to a subroutine? The answer lies in the RETURN from the subroutine. If you were to use GOTO to get to a subroutine from several different places in a program, the designation of where to return to after completion of the subroutine would be long and clumsy. GOSUB was invented to take care of just that problem.

A subroutine is a small program which you can imagine as being set aside from the main program. A subroutine can be used as often as you like while running the main program. Each time a subroutine is completed, the computer automatically returns to the line in the main program immediately following the line from which it earlier had left the main program. Here's a small example:

```
10  A = 555
20  GOSUB 100
30  PRINT T
40  END
100 T = A + 1
110 RETURN
```

The main program is contained in lines 10, 20, 30 and 40. The subroutine is lines 100 and 110. The jump to the subroutine is the instruction in line 20. Note that it contains the destination line number. The return from the subroutine is from line 110 to line 30.

At line 10, we assign the value 555 to memory location A. At line 20, we ask the computer to branch to the subroutine at line 100.

At line 100 the computer finds an instruction to recall the value of A and add one to that value. The new total is stored in memory location T.

The program moves on to line 110 where it finds RETURN. That instruction, which must *always* be at the end of a subroutine, tells the program to jump back to the line immediately following the line where it left the main program. In this case, the program left the main routine at line 20 so RETURN will kick it back to line 30.

At line 30 the computer finds a command to recall the contents of T and to display it. It does that and moves on to line 40. At line 40 it finds the END command and ceases operations.

Why an END in line 40? Because you need to make sure the subroutine is entered only from the GOSUB instruction. After line 30, without an END in line 40, the program would automatically move from line 30 to the next available higher line number which is 100. At line 100 it would enter the subroutine. At line 110 it would find a RETURN which did not come from a GOSUB and an error message would occur.

Just as a GOSUB must have a RETURN, the RETURN statement must come after a GOSUB.

The computer has a tiny private "scratchpad" bit of memory within itself where it writes temporary notes to itself. When it executes a GOSUB command, it makes note of the line number from which it left the main program. Later, when it finds a RETURN, it refers to its scratchpad to see where it left the main program. It determines the next available program line after that exit point and re-enters the main body of the program at that point.

If the computer encounters a RETURN without having left the main program via GOSUB, it won't be able to find a "where to" note on its scratchpad and will send you an error message. You don't want error messages so you pre-

vent the computer from getting into subroutines by means other than GOSUB jump commands.

Here's another example of a program with a subroutine:

```
 10  INPUT "ENTER A NUMBER", N
 20  GOSUB 100
 30  PAUSE "SQUARE ROOT IS ";T
 40  GOTO 10
100  T= √‾N
110  RETURN
```

This program computes the square root of a number you give it. The main body of the program is in lines 10, 20, 30 and 40. The subroutine is lines 100 and 110.

At line 10, the computer asks for a number and stores what you give it in memory location N. At line 20 it encounters a GOSUB instruction.

The computer immediately makes note of the fact that it is leaving the main body of the program at line 20. When it returns later, it will return to line 30.

The GOSUB is to line 100. The computer does that and finds an instruction at line 100 to recall the value of N, compute the square root of it, and store the answer in location T. Completing that the computer moves on to line 110.

At line 110, the RETURN statement makes the computer jump back to line 30 as predicted. At line 30 it recalls the contents of T and prints the specified message.

Line 40 is a different way to keep the program from going into the subroutine in lines 100 and 110. A jump is a way around a subroutine. Or, in this case, since the jump is back up to line 10 the program never sees line 100 except when properly sent there by GOSUB.

Two entries, one exit

A subroutine can have more than one entry point while having only one exit:

```
 10 CLEAR
 20 INPUT "ENTER A NUMBER", A
 30 IF A > 100 GOSUB 220
 40 IF X = 2 THEN 10
 50 GOSUB 200
 60 GOTO 10
200 X = 1
210 GOTO 230
220 X = 2
230 PAUSE X
240 RETURN
```

The main routine is in lines 10, 20, 30, 40, 50 and 60. The subroutine is lines 200, 210, 220, 230, and 240.

Note that line 50 contains a GOSUB branch to subroutine line 200 while line 30 has a GOSUB to line 220. Depending upon the IF/THEN decision in line 30, the computer will leave the main routine either at 30 or 50 and will enter the subroutine either at 220 or 200.

In either case, the subroutine ends at line 240 with a RETURN. It will return to the proper line in the main body of the program because its scratchpad note will tell it where it last left via GOSUB.

Line 10 CLEARs all of data memory. Line 20 asks for a number from you and stores that number in memory location A.

Line 30 is a test. If your number, as stored in A, is greater than 100, the program will complete the end of line 30 and GOSUB to line 220. If your number was not greater than 100, the test will fail and the program will go on to line 40.

Since we cleared all data memories back at line 10, X will be equal to zero so the test in line 40 will fail. It will not be true that X = 2. The program will go on to line 50.

Line 50 forces the computer to GOSUB to line 200. But what does this subroutine do?

The subroutine assigns a value of 1 to memory location X if the number you selected to enter was 100 or less. It

assigns a 2 to location X if your number was greater than 100.

If the subroutine is entered at line 200, a 1 is stored in X. Line 210 jumps around line 220 to line 230. Line 230 displays briefly the contents of X. Line 240 RETURNs action to the appropriate place in the main body of the program.

If, on the other hand, the subroutine is entered at line 220, the computer will store the value 2 in memory location X. At line 230 it will display briefly the contents of X. At line 240 it will RETURN. Lines 200 and 210 will not be used.

What's the practical result? The computer asks you for a number. You give one. It decides whether or not the number is greater than 100. If so, it shows a 2 on its display. If your number was not greater than 100, the computer will display a 1.

Try this one for fun:

```
10  CLEAR
20  INPUT "DO YOU LIKE ICE CREAM?",A$
30  GOSUB 200
40  INPUT "DO YOU LIKE HOT DOGS?",A$
50  GOSUB 200
60  INPUT "DO YOU LIKE CAKE",A$
70  GOSUB 200
100 PAUSE "YOU LIKE ";Y;" THINGS"
110 PAUSE "YOU DISLIKED ";N;" THINGS"
120 PAUSE "YOU GAVE ";I;" INVALID ANSWERS"
130 GOTO 10
200 IF A$ = "YES" Y = Y + 1:GOTO 230
210 IF A$ = "NO" N = N + 1:GOTO 230
220 I = I + 1
230 RETURN
```

The main program is in lines 10 through 130. The subroutine is lines 200 through 230.

The computer starts the program run at line 10, clearing all data memory. At line 20 it asks a YES-or-NO question. Your reply is stored in memory location A$.

Line 30 causes action to jump to the subroutine at line 200. The subroutine is a score keeper. If you answer no, it counts by adding one to N. If you answer YES, it counts that answer by adding one to the value stored in Y. If you offer any other answer, it counts that an invalid answer and adds one to memory location I.

We use Y to symbolize YES so we can remember it. For the same reason we use N to represent NO and I to mean invalid answer.

The computer completes the subroutine at line 230 where RETURN sends it back to line 40 in the main program.

The same thing happens after it asks its line 40 question. Line 50 sends operations to the subroutine where your answer is tallied and RETURN sends it back to line 60. The same for lines 60 and 70.

However, when the computer returns from the subroutine to the line following 70, it finds itself at line 100 with a different instruction.

Lines 100, 110, and 120 cause the computer to display the results of its question session. Line 100 reports to you how many YES answers you gave. Line 110 displays how many NO answers you entered. And line 120 tells how many times you typed in some answer other than YES or NO.

When the displays are complete, line 130 makes the computer jump back up to line 10 where data memory is erased and the entire process starts over.

Beeper

Earlier, we said the BEEP sound output was an unusual, very important function built into the pocket computer. It's available in the TRS-80 PC-2, TRS-80 PC-1, Sharp PC-1500 and Sharp PC-1211.

Here's how to put BEEP to use. Suppose you want your computer to make its sound five times at two-second intervals:

```
10  GOSUB 200
20  PAUSE "BEEP"
30  BEEP 1
40  GOSUB 200
50  PAUSE "BEEP"
60  BEEP 1
70  GOSUB 200
80  PAUSE "BEEP"
90  BEEP 1
100 GOSUB 200
110 PAUSE "BEEP"
120 BEEP 1
130 GOSUB 200
140 PAUSE "BEEP"
150 BEEP 1
160 END
200 FOR L = 1 TO 5
210 NEXT L
220 RETURN
```

The subroutine contains a time-delay loop set for approximately two seconds. Each time the computer returns from the loop, it displays the message BEEP and makes the beeping sound. When it has completed five message/sound cycles, it reaches the END in line 160 and quits.

Superpower
In Your Pocket

Superpower In Your Pocket

The Casio FX-702P, Radio Shack TRS-80 PC-2 and Sharp PC-1500 pocket computers have BASIC words available which give them extra computing power not found in the TRS-80 PC-1 or Sharp PC-1211. Here's a look at what you'll need to know to get started using these added capabilities:

FX-702P

The Casio pocket computer finds extra power in its ability to count and report the total number of characters in a string. And its ability to find portions of a string. It also can set up its data memory in *arrays.*

LEN is the instruction to count the number of characters stored as a string in data memory. The LENgth of the data then can be known. Here's an example:

```
10  A$ = "DOG"
20  L = LEN(A$)
30  PRT L
```

Line 10 stores the string DOG in memory location A. Line 20 examines the contents of A$, finds the total number of characters stored there, and puts that total number in memory location L.

We have chosen L. You can use any unused data memory location.

Line 30 causes the computer to recall the contents of L and show the number on its display. Here's an even more interesting set of instructions:

```
10  INP "GIVE ME A WORD",A$
20  L = LEN(A$)
30  WAIT 50
40  PRT L
50  GOTO 10
```

Here the computer asks you for a word to be stored in A$. Line 20 finds the length of the word. Lines 30 and 40 cause the length to be displayed for a couple of seconds and line 50 makes the computer go back to line 10 for a new word.

Do you remember a limitation on string size in the pocket computer? The limitation has a bearing on this program. Strings are limited to a total of seven characters.

Run the program. Give the computer the word DOG. It will flash a 3 to indicate there are three letters in DOG. Now give it BONE. It will report 4 letters in BONE. Next give it DOGBONE and it will display 7.

However, give it DOGBONES, which has eight characters, and you will get an error message telling you you have tried to sneak in a string longer than seven characters.

Only string data can be measured by the LEN function. Numerical data can't be measured. If you need to know how long a number is, store it as characters in a string. Like this:

```
10  N$ = "1234567"
20  L = LEN(N$)
30  PRT L
```

Even though you see 1234567 as a number, if you label its

memory location with the dollar sign, meaning string, the computer will treat it as a string. Then this program can be used to find there are seven characters in the string 1234567.

The only problem: it's easy to get a number into a string but you can't get it back out (in the FX-702P). And you might want it out as numerical data to do math with it. Try this:

```
10  INP "GIVE ME A NUMBER",A$
20  INP "GIVE ME ANOTHER NUMBER",B$
30  C$ = A$ + B$
40  PRT C$
```

At line 10 you give the computer a number. It stores the number in A$, rather than in A. That is, it puts it in a memory location with a string label so it is not stored as numerical data.

At line 20 the computer takes your number and stores it in B$.

Line 30 asks the computer to recall the contents of A$ and B$ and add them together. And here's the rub: adding strings means tying them together to make a longer string.

Suppose you gave the computer the numbers 11 and 22. Mathematically, they would add up to 33. But, when stored as strings and the strings are added, they result in 1122. Storing numbers in strings makes them the same as letters or other keyboard symbols. You lose the ability to do math with them.

Make this minor change to the program:

```
10  INP "GIVE ME A WORD",A$
20  INP "GIVE ME ANOTHER WORD",B$
30  C$ = A$ + B$
40  PRT C$
```

Run the program. Give it the words DOG and BONE. What do you get? DOGBONE. The two separate word strings are tied together by line 30 and printed as the combination at line 40.

When playing with this program, remember that A$, B$

and C$ can't store more than seven characters each, at one time.

Later we'll see how the TRS-80 PC-2 and the Sharp PC-1500 pocket computers can use the extraordinarily powerful BASIC instruction VAL to convert string data to numerical data!

Now you know about the special power in the FX-702P to find the LENgth of a string and to tie strings together by "adding" them. Combine these with the MID instruction and you have some unusual string-handling ability.

First, you must know the FX-702P has a special string-data storage place labeled only by the dollar sign. It doesn't have a letter coupled to the dollar sign. You designate this special storage box as in this example:

```
$ = "DOGBONES"
```

Letter-labeled memory locations are limited to seven characters per string. But $ can have up to 30 characters. Very interesting!

```
10 $ = "ABCDEFGHIJKLMNOPQRSTUVWXYZ1234"
```

Now that's a long string for a pocket computer. Suppose you want to extract a few of the letters from the middle of that string? Use the MID instruction.

```
10  $ = "ABCDEFGHIJKLMNOPQRSTUVWXYZ1234"
20  A$ = MID(5,6)
30  PRT A$
```

Run the program. It displays EFGHIJ. What happened?

Line 10 caused the 30-character string to be stored in "exclusive character variable" known as $. Then line 20 used the MID function to extract six characters starting at the fifth character in $.

MID always must be followed by numbers in parenthesis. Those numbers tell the computer where in the string to look for desired characters. For instance, MID(5,6) means extract six characters starting at the fifth character. MID(3,4) means get four characters starting at the third character into the string.

Let's look at input first.

When the program starts running, line 10 clears all data storage locations in memory. Line 20 asks for a letter from you and stores that letter in A$.

You can indicate to the computer that you have completed entering a word by entering the number zero in response to the GIVE ME A LETTER QUESTION. Line 30 tests your responses each time to see if you have keyed in a zero. If you have, line 30 will cause action to jump to line 100, ending the input routine. If you haven't typed in a zero, line 30 will allow action to drop to line 40.

At line 40, your letter is added to the special string location labeled $ and the program goes on to line 50. Line 50 makes the input routine skip back up to line 20 where the computer asks you for another letter. You can spell words of up to 30 characters total in this way. When you are done putting in letters, type in zero and action will move on to the string-manipulation section of the program.

At line 100, the computer determines the length of the string data stored in $. In other words, it counts how many letters there are in your word. The count total is stored in L.

Line 110 does some arithmetic to find how many letters are in half of your word.The value, which is half of L, is stored in H.

Line 120 then uses H to extract the letters which compose the first half of your word. Those letters are stored in B$.

Remember that H is half of the length of your word. MID(1,H) makes the computer start at the first character and get the first several characters equal to the number H. Suppose you had a four-letter word. L would equal 4 and H would equal 2. MID(1,H) would get two letters starting with the first letter in your word.

With MID(5,6) the computer started counting into $ at the letter A. The fifth letter in the string is E. Starting with E, it collects six letters. They are EFGHIJ. Those letters are stored as a new string in memory location A$.

Then line 30 causes the contents of A$ to be recalled and displayed.

MID is the same as MID$ in the TRS-80 PC-2, Sharp PC-1500 and other larger computers.

Try this bit of fun which includes all of the special FX-702P string-manipulation power:

```
10  VAC
20  INP "GIVE ME A LETTER",A$
30  IF A$ = "Ø" THEN 100
40  $ = $ + A$
50  GOTO 20
100 L = LEN($)
110 H = INT(L/2)
120 B$ = MID(1,H)
130 C$ = MID(L-H + 1,H)
200 PRT "THE WORD IS:",$
210 PRT "IT HAS ";L;" LETTERS"
220 PRT "FIRST HALF:",B$
230 PRT "SECOND HALF:",C$
240 PRT "WHOLE WORD:",$
250 GOTO 10
```

This program allows you to make up words on the computer. In effect, it seems the computer can spell!

The program is divided into three main blocks of lines: lines 10 to 50 are input. Lines 100 to 130 are string manipulation. Lines 200 to 250 are output.

Similarly, line 130 causes the computer to get the last half of your word and store it in C$.

In our four-letter word, L equals 4 and H equals 2. L-H + 1 would be 4-2 + 1 which would equal 3. Thus MID(L-H + 1,H) is the same as MID(3,2). And MID(3,2) tells the computer to get two characters starting at the third character into the string.

Suppose our four-letter word is WORD. The first two letters are WO and the last two are RD. MID(1,H) would extract WO and store it in B$. MID(L-H + 1, H) would get RD and store it in C$. The word is broken into its two halves.

The output part of the program is lines 200 to 250. Line 200 displays the entire word you entered.

Line 210 reports how many letters are in your word. Line 220 prints the letters in the first half of the word. Line 230 prints the letters in the second half of the word. Line 240 prints the entire word again. And line 250 finds the job is completed and sends action back to line 10 where your word is removed from memory and the game can start over.

By the way, what if you have a word with an odd number of letters? The middle letter is ignored by this program!

Try DOGBONE which has seven characters. Line 200 shows the entire word to be DOGBONE. Line 210 reports it has 7 characters.

Line 220 says the first half is DOG. Line 230 recalls the second half has ONE. The middle letter, B, has been lost. Why? Because half of an odd number of letters is half of a letter and we can't display half of a letter. So, we allow it to be lost. You can change that if you like.

Note: you will need to use the CONT key after each display when running this program.

TRS-80 PC-2 and PC-1500

The pocket computer model PC-2 from Radio Shack and the Sharp PC-1500 have many additional BASIC words which give them a superpowerful advantage over the TRS-80 PC-1, Sharp PC-1211, or the Casio FX-702P. In fact, they have more than 100 instructions in their vocabulary. Except for minor differences in keyboard arrangement, the PC-2 and the PC-1500 are identical pieces of hardware.

String Manipulation

The PC-2/PC-1500 have the important MID$ and LEN functions discussed earlier in the FX-702P section. And they have the ability to add strings together to make longer strings. And they have LEFT$ and RIGHT$ to get those halves of a string.

But, in addition, they take a major step forward by allowing you to convert strings to numbers and numbers to strings.

VAL is the BASIC word which converts strings to numbers. Suppose you had stored the number 555 in A$:

```
10  A$ = "555"
20  N = VAL(A$)
30  PRINT N
```

VAL actually looks inside the quotation marks to see what the string is composed of and converts that information into a number. That numerical data can be used for math.

Similarly, STR$ converts a number to a string. Like this:

```
10  N = 555
20  A$ = VAL(N)
30  PRINT A$
```

Type both of these programs into your PC-2 or PC-1500 and see how they run.

Try this program as a test of converting strings to numbers so arithmetic can be done:

```
10  INPUT "GIVE ME A NUMBER",A$
20  INPUT "GIVE ME ANOTHER NUMBER",B$
30  C = VAL(A$) + VAL(B$)
40  PRINT C
```

C actually contains the mathematical total of the numbers which had been stored as strings in A$ and B$. Suppose you had used the numbers 11 and 22. If the program line 30 were

```
30  C = A$ + B$
```

the result would have been stored in C$ as 1122. But if the program line 30 were

```
30  C$ = VAL(A$) + VAL(B$)
```

then the result will be the arithmetic addition of 11 and 22 which gives a total of 33.

INKEY$

The computer can be made to watch constantly for any press of a key on its keyboard. The instruction is INKEY$. It's the same as the instruction KEY on the FX-702P.

INKEY$ gets, as a string, whatever keyboard key you

have pressed. Therefore, INKEY$ must be used with a storage location at all times. For instance:

A$ = INKEY$

The computer notes which key you have pressed and stores that information in memory location A$. Try this simple program:

```
10 CLEAR
20 A$ = INKEY$
30 IF A$ = "" THEN 20
40 PAUSE A$
50 GOTO 10
```

As the computer enters its run at line 10, all data memory locations are cleared. At line 20, the computer scans the entire keyboard one time. If it finds you are pressing a key, it takes that data and stores it in A$. If you are not pressing any key, it does not store anything in A$.

Either way, after one scan the computer goes on to line 30. Line 30 is a test of whether or not the computer found you had pressed any key. If you had pressed a key, something would be stored in A$. If you had not pressed any key, nothing would be stored in A$. A pair of quotation marks together with no space between, like this"", indicates "nothing " is inside the string.

If A$ contains nothing then the computer will complete the action called for in the last half of line 30. It will jump back to line 20.

If A$ contains something, the computer will go on to line 40. At line 40 it displays briefly for you to see the contents of memory location A$. Afterwards, it goes to line 50 which causes it to jump back to line 10. At line 10 A$ and all other memory locations are cleared and the process starts over.

The net result of all this program action is a blank display until you press a key. The key you pressed is displayed momentarily and then a blank display resumes until you press another key. And so on. Pretty neat!

Just imagine how you can couple the use of INKEY$ with the PC-2 Printer/Plotter or CE-150 Printer to turn your

pocket computer into an electric typewriter. Or even a micro-wordprocessor.

DATA/READ/RESTORE

The ability to store large amounts of data in program lines can be valuable. Here's a program line with information stored in it:

```
10 DATA ALABAMA,FLORIDA,GEORGIA,TEXAS
```

Use the READ instruction to make the computer read these one at a time, left to right. Like this:

```
10 DATA ALABAMA,FLORIDA,GEORGIA,TEXAS
20 READ A$
30 PAUSE A$
40 GOTO 20
```

The computer will start at line 10 where we have placed the names of four states in a DATA line. Line 20 causes it to read the first state name. Line 30 makes it display the name briefly. Then line 40 jumps action back to line 20 where the computer reads the next piece of DATA found in line 10. DATA lines can be located anywhere in your program. The computer will search for them and read the items contained in them one at a time.

We placed four pieces of data in the DATA line. After the fourth pass through this READ/PAUSE/GOTO loop the computer will run out of DATA. It will give you a message that it is out of data. Use the RESTORE instruction to force the computer to go back to the first DATA item. Here's a program:

```
10 DATA ALABAMA,FLORIDA,GEORGIA,TEXAS
20 FOR L=1 TO 4
30 READ A$
40 PAUSE A$
50 NEXT L
60 RESTORE
70 GOTO 20
```

Again, as in the previous example, line 10 contains the DATA. Lines 20 and 50 create a FOR/NEXT loop to make the computer READ and display data four times.

After the fourth pass through the loop the computer will have exhausted the data in line 10 and will have completed the FOR/NEXT loop. The program will drop to line 60 where the computer finds a RESTORE instruction.

RESTORE tells the computer to start over at the beginning of all DATA lines. The computer reads through DATA lines one item at a time, one line at a time, until all items in all lines are used up. A RESTORE instruction at any time resets the READ to the first item in the first DATA line.

Having RESTOREd via line 60, our program at line 70 jumps action back to line 20 where the FOR/NEXT loop starts again. And so on.

Two-Letter Memory Locations

So far, we have treated memory locations as only A through Z. We have done this for simplicity as well as the fact that those 26 storage boxes are what we have available in the TRS-80 PC-1 and Sharp PC-1211. However, the TRS-80 PC-2 and Sharp PC-1500 have two-letter variables available for storage. As you can imagine, this vastly increases the number of storage boxes in data memory. Here are some examples of two-letter memory locations:

```
AA = 555
MX$ = "WORD"
ZZ = 789.123
```

Use these two-letter variables the same as the familiar one-letter variables. Just imagine you have lots more storage boxes available. See figure 5.

Here's a brief sample program so you can test two-letter memory locations:

```
10  INPUT "GIVE ME A NUMBER",AA
20  INPUT "GIVE ME ANOTHER NUMBER",BB
30  CC = AA*BB
40  PRINT CC
```

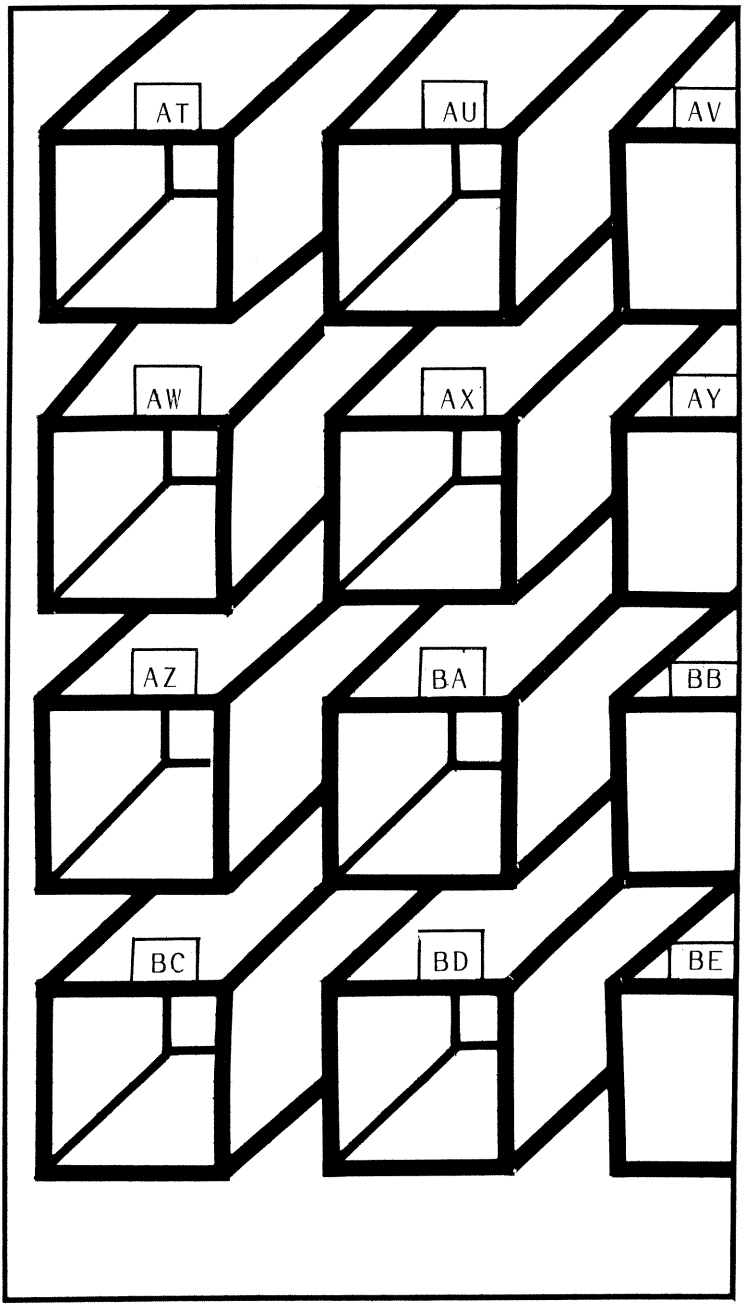


FIGURE 5: TWO-LETTER BOXES

Line 10 takes your number and stores it in memory location AA. Line 20 puts your second number in location BB. Line 30 causes the computer to multiply AA times BB and store the result in memory location CC. Line 40 recalls the contents of memory location CC and prints them on the display.

Especially Useful for Math

Especially Useful For Math

We've seen how to "manipulate" string data. Now let's look at some exciting ways to manipulate numbers.

INT

Suppose you have a long number, including several digits after a decimal point. You want to get rid of all places after the decimal point. Try this:

```
10  A = 1234.56789
20  B = INT A
30  PRINT B
```

Run the program. The result, printed on the display, will be 1234. The 56789 after the decimal point have been hacked off and dropped. If you recall A, you'll see it still contains 1234.56789 but if you recall B, it contains only 1234. Here's another:

```
10  A = 22/7
20  INPUT "GIVE ME A NUMBER",B
30  C = A*B
40  D = INT C
50  PRINT D
```

Run this program, give the computer the number 123 which it stores in B, and the final result will be a display of 386.

Checking various memory locations after the run, you'll find A contains 3.142857143 and B holds 123. C contains 386.5714286 and D holds 386. The contents of D were the result printed by program line 50.

INT is short for integer, meaning the whole number without its fractional part after the decimal point.

Dump integer, keep fraction

You can use the INT instruction to get rid of the integer and keep the fractional part of the number after the decimal point.

```
10  A = 22/7
20  B = A-INT A
30  PRINT B
```

Here the program results in a display of 0.142857143. Line 10 divides 22 by 7 to get the number 3.142857143 which is stored in memory location A. Line 20 then takes the entire number stored in A and subtracts its integer from it. Taking away its integer leaves only its fractional part after the decimal point!

Casio users have a BASIC word to do this: FRAC. Try this program in your FX-702P:

```
10  A = 1234.56789
20  B = FRAC A
30  PRINT B
```

The run result will be a display of 0.56789 with the 1234 missing.

Here's another neat Casio trick:

```
10  A = 1234.56789
20  B = INT A
30  C = FRAC A
40  WAIT 50
50  PRT "INT = ";B
60  PRT "FRAC = ";C
70  IF A = B + C;PRT "NUMBER = ";A
80  PRT "GOOD WORK!"
```

The display shows you the integer portion of the number, the fractional part of the number, and the entire number.

Random numbers

The TRS-80 PC-1 and Sharp PC-1211 do not have a BASIC word which directly instructs the computer to make up a number at random. The TRS-80 PC-2, Sharp PC-1500 and Casio FX-702P do have such BASIC words. So, random number generation in the PC-2, PC-1500, and FX-702P are easy. It takes a bit more work to get a random number out of the PC-1/PC-1211:

```
10 CLEAR
20 INPUT "ENTER A NUMBER";R
30 R=(( π + R) ∧ 5)-INT(( π + R) ∧ 5
40 PAUSE R
50 GOTO 30
```

This program will generate and display a series of random numbers between zero and one. It will continue the series until you press the BREAK key.

The program requires what is called a "seed." You must give it the first number, *any* number, and it will follow with the series of random numbers.

Suppose you gave it the *seed* number 55.99. Here's part of the series of pseudorandom ("seemingly random") numbers you might get from the computer:

```
0.486
0.19224183
0.832078773
0.740159494
0.33092583
etc.
```

You can use the following program to instruct the computer to output a series of 25 pseudorandom numbers from zero to 10:

```
10  CLEAR
20  FOR N = 1 TO 25
30  A = NN:Y = (947*X + A)/199:X = Y-INT Y:X = 10X:W = INT X
40  PAUSE "RANDOM NUMBER = ";W
50  NEXT N
```

Try it. You'll find the seed number is located in line 20!

FX uses RAN#

To make life easier, Casio built a random number generator into their FX-702P pocket computer. It uses a special BASIC word RAN# which is different from the more typical word, RND, used on many computers. Casio chose to use RND for an instruction to round-off numbers which we'll see in a moment.

Here's a Casio random-number generator program:

```
10  N = RAN#
20  PRT N
```

If you run this small program repeatedly, you will get a series of numbers like ours:

```
0.803072212
0.023027387
0.511945457
0.064941969
0.468156276
   etc.
```

How can you put these long decimal numbers to use? Convert them into practical forms of numbers. For example, suppose you wanted 10 whole numbers in the range of zero to nine:

```
10  N = INT(10*RAN#)
20  WAIT 20
30  PRT N
40  GOTO 10
```

Running this program resulted in a series of random numbers including:

```
9
2
```

3
8
Ø
1
4
7

Your numbers will be different since the series is much more random in the Casio than in the PC-1/PC-1211 programs shown earlier.

Imagine a game requiring the use of a pair of dice. Dice give a total of up to 12 when thrown as a pair. Here's a program generating a series of random numbers in the range of one to twelve to simulate dice:

```
10  N = INT(13*RAN#)
20  IF N < 1 THEN 10
30  WAIT 20
40  PRT N
50  GOTO 10
```

Line 20 is in there because the result of line 10 is a series of numbers in the range of zero to twelve. We don't want the zero. We want our series to be from one through twelve. So, we use line 20 to trap out any zeros which show up after line 10. Running this program, we got the series:

6
1
9
11
10
2
Ø
3
12
7
8
5

Try it. You'll soon be able to put together a quickie dice game for hours of fun!

Rounding off

There are two views on how to round off a number. One holds that *if the number is more than five, you round up.* Which means that exactly 0.5 rounds down. The other opinion is that *any number less than five rounds down.* In that case, exactly 0.5 rounds up.

Here's a set of program lines for the *more than five rounds up* idea:

```
10  CLEAR
20  INPUT "NUMBER TO BE ROUNDED",N
30  IF N > INT N THEN 50
40  R = N: GOTO 100
50  D = N-INT N
60  IF D > .5 THEN 90
70  R = INT N
80  GOTO 100
90  R = INT N + 1
100 PAUSE N;" ROUNDS TO..."
110 PRINT R
120 GOTO 10
```

The next set of program lines, below, rounds off on the *less than five rounds down* theory.

```
10  CLEAR
20  INPUT "NUMBER TO BE ROUNDED",N
30  IF N > INT N THEN 50
40  R = N:GOTO 100
50  D = N-INT N
60  IF D < .5  THEN 90
70  R = INT N + 1
80  GOTO 100
90  R = INT N
100 PAUSE N;" ROUNDS TO..."
110 PRINT R
120 GOTO 10
```

Making life easier, Casio built in a rounding-off BASIC word: RND .

```
10  A = 1234.56789
20  B = RND(A,-3)
30  PRT B
```

Running this program results in a display of 1234.57. The number has been rounded off to two decimal places. The amount of rounding off is controlled by the -3 in line 20. It rounds off the third number from the decimal, down to the second number from the decimal.

Here's a list of other possible line 20's for this program and the results on the display after running:

program line	result
B = RND(1234.56789, + 3)	Ø
B = RND(1234.56789, + 2)	1000
B = RND(1234.56789, + 1)	1200
B = RND(1234.56789,0)	1230
B = RND(1234.56789,-1)	1235
B = RND(1234.56789,-2)	1234.6
B = RND(1234.56789,-3)	1234.57
B = RND(1234.56789,-4)	1234.568
B = RND(1234.56789,-5)	1234.5679
B = RND(1234.56789,-6)	1234.56789
B = RND(1234.56789,-7)	same

The list can help you visualize how changing the place number changes the result.

Storing Information

Storing Information

You've just spent an hour typing a 65-line program into your pocket computer. You appreciate the amount of work involved in entering such a list and you would like to *not* have to do it all again the next time you want to run the same program. Use the CSAVE command and store it on magnetic recording tape.

That's right! The exact same kind of magnetic recording tape you might use to store your favorite musical selection or the sound of your kid's birthday party.

Just about *any* cassette or reel tape recorder will work. It must have three jacks to connect to the pocket computer's cassette interface:

First, it must have a microphone-input jack, often labeled MIC on the recorder. This will be used to take sound coming out of the computer into the recorder.

Second, it must have an earphone-ouput or external-speaker output jack, often labeled EAR on the recorder. This will be used to take sound out of the recorder and into the computer.

Third, it must have a remote-control jack, often labeled REM. This will be used by the computer to start and stop the recorder motor.

Is it plugged in?

Refer to your owner's manual to make sure you have the correct plug in the correct jack. Generally, these are color-coded and size-coded so you cannot get the wrong plug in the wrong jack. But make sure you have them firmly plugged in and in the right holes.

You may use an A.C. wall outlet-powered recorder or a battery-powered recorder. If outlet-powered, make sure it is plugged into the house A.C. If battery-powered, make sure you have a good set of batteries in the recorder.

What kind of tape?

Any magnetic recording tape will work as long as it fits your recorder. Can you use a 79¢ dime-store cassette? Yes.

The computer will be sending a continuous warbling high-pitched tone into your tape recorder. The presence or absence of that tone, when returned to the computer in the future, will have a bearing on the ability of the computer to run the program you have stored. You must take care that nothing happens to cause any part of that recording to be erased or garbled or destroyed.

First, don't allow anything magnetic to be near the tape. A magnetic tape recording is erased by magnets. Electric motors, TV sets, and other appliances generate magnetism. Don't permit an accidental erasure by putting your tape atop an electric motor or TV set!

Second, the tone must be continuous on the tape without breaks in the sound, called "dropouts." Once in a while, a cassette tape will have a splice in it. The splice will provide a minute length of tape with no magnetic recording ability. That splice, then, causes a drop out. Better quality audio tape is less likely to have dropouts.

Data tape is an audio tape which has been manufac-

tured with no splices and without a paper or plastic non-recording leader at the beginning of the tape. A data tape is supposed to be nothing but magnetic recording tape. No dropouts!

However, data tapes cost more than plain-jane audio tape. An acceptable compromise is to use a better audio tape but one cheaper than data tape.

Here's a trick to ensure a good recording: send the program to tape *twice*. That's right. Use the CSAVE instruction once. Advance the tape slightly after recording is complete and use CSAVE again. You will have two complete recordings of your program. The odds favor at least one of them being good.

Storing programs

Make sure you have the tape recorder and computer properly plugged together and a tape positioned in the recorder. Press PLAY and RECORD on the recorder. Put the computer in the PRO, RUN or DEF modes if it's a TRS-80 or Sharp. Casio owners use MODE 1.

To tape record the contents of program memory on tape type in:

CSAVE "FILENAME"

and press ENTER. The computer will start the tape recorder, send sound to the recorder, and stop the recorder motor when done.

You may pick any file name you wish, up to seven characters in length.

Casio owners substitute SAVE for CSAVE.

Storing data

To tape record the contents of data memory, set your computer to the DEF or RUN modes and type in:

PRINT#"FILENAME"

and press ENTER. Again the computer will operate the recorder and stop it when recording is complete. The filename may be any set of up to seven characters.

All of data memory will be recorded on tape. If you wish

to record on tape only the contents of *one* data memory location, follow this example:

PRINT#"FILENAME";A$

Use the semicolon as shown and follow it with the specific data memory location. Remember that data memory locations are labeled A through Z or A$ through Z$ or A(1) through A(204) or A$(1) through A$(204).

Reloading program

Loading is a computer buzzword meaning putting a program into a computer. You speak of "loading from tape" when you want to get a program stored on tape and enter it into the computer. The sound on tape goes from the recorder through the wires into the computer where the computer stores it in program memory, erasing whatever might have been there before.

Sometimes you also hear computer hobbyists speak of "loading to tape," meaning the computer sending its program to the recorder for storage. You know how to do that using the CSAVE instruction. Now let's *load* the stored program from tape back into the computer using CLOAD. Here's the instruction:

CLOAD "FILENAME"

followed by ENTER. Use it in the DEF, RUN and PRO modes.

To use CLOAD, you must rewind the tape to a point immediately before where the computer-sound starts on the tape. Then type in CLOAD "FILENAME" and ENTER. The computer will start the recorder motor, receive sound from the recorder, store the incoming program in program memory, and, when done, stop the recorder motor.

File names are limited to seven characters. Casio users use LOAD#n"FILENAME" since you have to specify which of the 10 available program storage areas you wish the program to be placed in. Substitute the appropriate number for the "n" in LOAD#n"FILENAME".

Casio file names for SAVE and LOAD may be eight characters in length.

Checking your load

You can instruct the computer to verify its new program by using the CLOAD? command. The question mark must be there.

Set up recorder, tape, and computer as if you were going to load into the computer a new program from tape. Then use:

CLOAD?"FILENAME"

and press ENTER. The computer runs through its program memory as it listens to the tape, verifying that the program did in fact load properly the first time. If it did not, and the computer finds a mistake, it will send you Error Message 5.

Casio owners use VER"FILENAME" rather than CLOAD? to verify a program from tape.

Reloading DATA

To recall data from tape, use this instruction:

INPUT# "FILENAME"

and ENTER. The file name can be any set of up to seven characters, in the TRS-80 PC/Sharp PC. Up to eight characters in the Casio.

INPUT# gets all data stored on tape and puts it into data memory in the computer, erasing whatever was there before. To get data for only one memory location, use:

INPUT# "FILENAME";A$

and ENTER. You may substitute for the A$ we use in the example any of the proper memory location names. Remember they are A through Z or A$ through Z$ or A(1) through A(204) or A$(1) through A$(204).

Casio data storage

The FX-702P uses PUT and GET instructions to store data on tape and retrieve it.

Data is sent to tape and recorded with this instruction:

PUT "FILENAME" variable

and EXE. The variables are $, A and B to T9. To store the

contents of memory locations A through Z, for example, use:

PUT "FILENAME" A,Z

and the EXE key.

To reload data from tape into the FX-702P, use the GET instruction like this:

GET "FILENAME" variable

and EXE. For example, to recall from tape the contents of data memories A, Z use this instruction:

GET "FILENAME" A,Z

The computer will control the recorder and take in the information stored on tape, storing it in locations A through Z.

Loading while running

The computer can be made to go to tape, during a program run, to get new instructions. Here's the format:

CHAIN "FILENAME"

The computer, when it comes to a CHAIN instruction, immediately goes to the tape to look for new program lines stored there. You can even have the computer start at a particular program line. Suppose you wanted the program to start at line 100 from tape:

CHAIN "FILENAME",100

Or if you have labeled a line "A" use this:

CHAIN "FILENAME","A"

The computer will automatically search the tape for the specified file name and transfer the program it finds to program memory.

Appendix

Pocket Computer
BASIC Instructions

Pocket Computer Instructions

This table compares 175 BASIC instructions available on the various Radio Shack, Sharp and Casio pocket computer models. The **X** in a column indicates that feature is available on that particular computer.

The column headed **I** includes TRS-80 PC-1 Pocket Computer and Sharp PC-1211.

The column headed **II** includes TRS-80 PC-2 Pocket Computer and Sharp PC-1500.

The column headed **FX** includes Casio FX-702P.

The **notes** column is a very brief key to what the particular command, statement, function or operator does in the computer.

Instruction	I	II	FX	Notes
ABS	x	x	x	absolute value
ACS	x	x	x	arc cosine
AHC			x	arc hyperbolic cosine

Instruction	I	II	FX	Notes
AHS			x	arc hyperbolic sine
AHT			x	arc hyperbolic tangent
AREAD	x	x		read display
ARUN		x		start
ASC		x		string first character number
ASN	x	x	x	arc sine
ASTAT			x	display statistics
ATN	x	x	x	arc tangent
BEEP	x	x		make beep sound
BEEP ON		x		sound on
BEEP OFF		x		no sound
CHAIN	x	x		load from tape & run
CHR$		x		convert number to graphics
CLEAR	x	x		empty data memory
CLOAD	x	x		load from tape
CLOAD?	x	x		verify tape
CLR			x	erase one program
CLR ALL			x	erase all program memory
CLS		x		clear the display
CNT			x	count entries
COLOR		x		control printer color
CONT	x	x	x	go on after STOP
COR			x	correlation coefficient
COS	x	x	x	cosine
CSAVE	x	x		save to tape
CSIZE		x		printer size control
CSR			x	cursor control
CURSOR		x		cursor control
DATA		x		stored in program line
DEBUG	x			one-step checking run
DEFM			x	specify number of memories
DEG	x	x	x	degrees to decimal
DEGREE	x	x		set DEG mode
DEL			x	delete
DIM		x		dimension the memory
DMS	x	x	x	decimal to degrees
END	x	x	x	end of run

Instruction	I	II	FX	Notes
EOX			x	estimate X
EOY			x	estimate Y
EXP	x	x	x	exponent
FOR	x	x	x	FOR/NEXT loop
FRAC			x	fractional part of number
GCURSOR		x		control graphics cursor
GET			x	get data from tape
GLCURSOR		x		printer graphics control
GOSUB	x	x		jump to subroutine
GOTO	x	x	x	jump to line
GPRINT		x		graphics display
GRAD	x	x		set GRAD mode
GRAPH		x		printer control
GSB			x	GOSUB
HCS			x	hyperbolic cosine
HSN			x	hyperbolic sine
HTN			x	hyperbolic tangent
IF	x	x	x	decision maker
INKEY$		x		watches keyboard input
INP			x	INPUT
INPUT	x	x		take in data
INPUT#	x	x		data from tape
INT	x	x	x	integer part of number
KEY			x	same as INKEY$
LEFT$		x		gets left part of string
LEN		x	x	finds length of string
LET	x	x		make
LF		x		printer control
LINE		x		printer draw line
LIST	x	x	x	display program memory
LIST ALL			x	see LIST
LIST V			x	LIST variables
LLIST		x		LIST on printer
LN	x	x	x	natural logarithm
LOAD			x	get one program from tape
LOAD ALL			x	get all programs from tape
LOCK		x		stay in RUN mode

Instruction	I	II	FX	Notes
LOG	x	x	x	common logarithm
LPRINT		x		print on printer
LRA			x	linear regression constant
LRB			x	linear regression coefficent
MEM	x	x		memory available
MERGE		x		tape command
MID			x	see MID$
MID$		x		get middle part of string
MODE	x	x	x	write vs. run program
MX			x	X mean
MY			x	Y mean
NEW	x	x		erase all programs
NEXT	x	x	x	FOR/NEXT loop
ON ERROR GOTO		x		if error jump to line
ON GOSUB		x		controls sub jump
ON GOTO		x		controls line jump
PASS			x	password
PAUSE	x	x		display briefly
PI	x	x	x	math function
POINT		x		graphics function
PRC			x	polar to rectangular
PRINT	x	x		display info
PRINT#	x	x		send data to tape
PRT			x	PRINT
PUT			x	send data to tape
RADIAN	x	x		set RAD mode
RANDOM		x		randomize
RAN#			x	create random number
READ		x		get info from DATA line
REM	x	x		remarks
RESTORE		x		reset READ start DATA
RET			x	RETURN
RETURN	x	x		jump back from subroutine
RIGHT$		x		get right part of string
RLINE		x		printer control
RMT ON		x		turn it on
RMT OFF		x		turn it off

Instruction	I	II	FX	Notes
RND		x		random number
RND			x	rounding off number
ROTATE		x		printer control
RPC			x	rectangular to polar
RUN	x	x	x	start program execution
SAC			x	clear statistics memory
SAVE			X	CSAVE one prog. to tape
SAVE ALL			x	CSAVE all progs. to tape
SDX			x	standard deviation of X
SDXN			x	normal standard deviation x
SDY			x	standard deviation of Y
SDYN			x	normal standard deviation Y
SET			x	control display digits
SGN	x	x	x	sign of number
SIN	x	x	x	sine
SORGN		x		printer control
SQR	x	x	x	square root
STAT			x	accumulate X & Y
STATUS		x		command query
STEP	x	x	x	FOR/NEXT loop increment
STOP	x	x	x	halt program run
STR$		x		convert number to string
SX			x	sum of X
SX2			x	sum of X^2
SXY			x	sum of X*Y
SY			x	sum of Y
SY2			x	sum of Y^2
TAB		x		printer print location
TAN	x	x	x	tangent
TEST		x		printer check
TEXT		x		printer control
THEN	x	x	x	use with IF
TIME		x		displays clock
TO	x	x	x	FOR/NEXT loop
TRACE ON		x	x	turn on DEBUG
TRACE OFF		x	x	turn off DEBUG
UNLOCK		x		remove LOCK

Instruction	I	II	FX	Notes
USING	x	x		control display format
VAC			x	same as CLEAR
VAL		x		convert string to number
VER			x	verify data tape
WAIT		x	x	cause momentary display
√‾	x	x		square root
π	x	x	x	pi
<>	x	x		not equal to
≠			x	not equal to
&		x		AND
=	x	x	x	equals
+	x	x	x	plus
—	x	x	x	minus
*	x	x	x	multiply by
/	x	x	x	divide by
^	x	x	x	power
( )	x	x	x	parenthesis
>	x	x	x	greater than
<	x	x	x	less than
> =	x	x	x	equal to or greater than
< =	x	x	x	less than or equal to
AND		x		and
OR		x		or
NOT		x		not
¥	x			Japanese Yen

Error Messages

TRS-80 PC-1/Sharp PC-1211 Error Messages

Error Number	Problem
1	You wrote the program line wrong; or you have the computer trying to divide by zero which it won't do; or the computer is reaching a math result higher than $1x10^{100}$which it can't handle; or you are trying to use string data in numerical data memory locations; or you are trying to use numerical data in string memory locations.
2	You have tried to send the computer to a line which doesn't exist. Check your GOTO, GOSUB, RUN, DEBUG or LIST statement.

3	You have tried to get the computer to give more than four stages of GOSUB/RETURN or FOR/NEXT loops, which it can't handle; or you have tried to use RETURN without a preceeding GOSUB so the computer doesn't know where to go; or you have tried to do a NEXT with a FOR so the computer doesn't know what to do.
4	You have tried to write a program longer than available program memory; or you have tried to write more reserve program steps than are available in reserve memory; or you don't have enough flexible memory available for what you are trying to do.
5	An error has occurred while loading from or saving to tape.
6	You are using PRINT or PAUSE and the format of your numerical data is wrong.

Casio FX-702P Error Messages

Error Number	Problem
1	You have tried to write a program longer than available program-memory. Reduce the number of lines in the program or clear other programs out of program memory.
2	You wrote the program line, using BASIC improperly. Rewrite the program line correctly.

3 You are doing math and the computer can't handle what you have asked; the number may be larger than 10^{100} which the computer can't handle or some other impossible result is at hand.

4 You have tried to send the computer to a line which doesn't exist. Check GOTO and GOSUB.

5 You haven't dimensioned an array properly.

6 Use DEFM to increase data memory.

7 You have the computer trying a RETURN without a preceeding GOSUB; it doesn't know where to go. Or it has found a NEXT without a preceeding FOR so it doesn't know what loop it is in; or you are trying to exceed 10 levels of subroutines; or you are trying for more than eight levels of nested FOR/NEXT loops.

8 You don't know the password. You are trying to LIST or CLR without giving the correct password. Or you are trying to give the password and not getting it right.

9 You are trying to SAVE to tape or LOAD from tape and you don't have a tape recorder hooked up; or you are trying to PRINT on a paper printer and you don't have the printer hooked up.

Index

Other books from ARCsoft Publishers

For the TRS-80 PC-2, PC-1, Sharp PC-1500, PC-1211, Casio FX-702P:

101 Pocket Computer Programming Tips & Tricks
ISBN 0-86668-004-7 — 128 pages — $7.95

50 Programs in BASIC for Home, School & Office
ISBN 0-86668-502-2 — 96 pages — $9.95

50 MORE Programs in BASIC for Home, School & Office
ISBN 0-86668-003-9 — 96 pages — $9.95

Murder In The Mansion and Other Computer Adventures
ISBN 0-86668-501-4 — 96 pages — $6.95

35 Practical Programs for the Casio Pocket Computer
ISBN 0-86668-014-4 — 96 pages — $8.95

Pocket Computer BASIC Coding Form
programming worksheets — 40-sheet pad — $2.95

For the TRS-80 Color Computer

101 Color Computer Programming Tips & Tricks
ISBN 0-86668-007-1 — 128 pages — $7.95

Color Computer Graphics
ISBN-086668-012-8 — 128 pages — $9.95

The Color Computer Songbook
ISBN 0-86668-011-X — 96 pages — $7.95

55 Color Computer Programs for the Home, School & Office
ISBN 0-86668-005-5 — 128 pages — $9.95

55 MORE Color Computer Programs for the Home, School & Office
ISBN 0-86668-008-X — 112 pages — $9.95

My Buttons Are Blue and Other Love Poems From
The Digital Heart of An Electronic Computer
ISBN 0-86668-013-6 — 96 pages — $4.95

Color Computer BASIC Coding Form
programming worksheets — 40-sheet pad — $2.95

For the APPLE Computer

101 APPLE Computer Programming Tips & Tricks
ISBN 0-86668-015-2 — 128 pages — $8.95

33 New APPLE Computer Programs for Home, School & Office
ISBN 0-86668-016-0 — 96 pages — $8.95

APPLE Computer BASIC Coding Form
programming worksheets — 40-sheet pad — $2.95

Computer Programming Worksheets 40-sheet pads:

IBM Personal Computer BASIC Coding Form — $2.95
APPLE Computer BASIC Coding Form — $2.95
TRS-80 Color Computer BASIC Coding Form — $2.95
Pocket Computer BASIC Coding Form — $2.95
Universal BASIC Coding Form — $2.95

For electronics hobbyists

25 Electronics Projects for Beginners
ISBN 0-86668-017-9 — 96 pages — $4.95

25 Easy-To-Build One-Night & Weekend Electronics Projects
ISBN 0-86668-010-1 — 96 pages — $4.95

ISBN: *International Standard Book Number*

Pocket-BASIC Book V

Pocket Computer Programming Made Easy

by Jim Cole

This book is the ideal companion for all of the exciting new handheld pocket computers. You don't have to know anything about computers or programming to use this handy beginner's guide to BASIC, the most popular program language.

Written especially for the immensely-popular pocket computers—TRS-80 PC-2 and PC-1, Sharp PC-1500 and PC-1211, Casio FX-702P—the BASIC instruction in this book is applicable to any computer, whether pocket-size or desktop. It's the perfect introduction to using BASIC in any computer.

The simple down-to-earth language will help you quickly understand how to talk to your computer and get it to do what you want.

Jim Cole takes you step-by-step through the most frequently used words in BASIC. The language is a lot like English and author Cole points out the familiar look-alikes which have meanings you already know and understand.

Building on what you already know, even before you start, he uses familiar English to show how your computer receives instructions and information from you and how it follows those instructions. Sections in this book explain clearly *What's Inside Your Computer; Writing and Running Programs; Input and Output; The Real Computer Power!; Superpower In Your Pocket; Especially Useful for Math; Storing Information; Pocket Computer BASIC Instructions;* and *Error Messages.*

For the first time anywhere, this book presents a complete list of *all* BASIC words—commands, statements, functions, operators, instructions—used in *all* of the popular pocket computers. Displayed side-by-side in a handy look-up table, this exhaustive source list makes using BASIC in a pocket computer a breeze!

No scientist-talk, no engineering jargon, no Ph.D. lingo here. No computerese over the heads of beginners. This book has been tailored for the layman who wants to get started using a microcomputer. There never has been a learner's guide to BASIC this easy to read and understand and put to use.

ARCsoft Publishers
WOODSBORO, MARYLAND 21798

ISBN 0-86668-009-8