INTRODUCTION TO PROGRAMMING

in

TINY BASIC

By Tom Pittman

PART ONE

0 — GETTING STARTED

You have your own personal computer! It will do anything you tell it to do, if it understands you. There is an old story about a farmer who said his mule obeyed him perfectly. A reporter asked him how he did it, and the farmer said, "with a pleasant voice and kind words." The reporter allowed as he had to see this, so the farmer took him over to the mule, picked up an old two–by–four, and gave the mule a sharp blow on the head.

"Wait a minute," said the reporter, "I thought you said you did it with a pleasant voice and kind words!"

The farmer replied, "I do, but I have to get the mule's attention first."

Your ELF II will obey you exactly, but you have to get its attention first. No, not with a two–by–four! We will use a special program which enables your ELF to understand what you say. This program is called a TINY BASIC Interpreter. It is an "interpreter" because it interprets what you are saying to the computer in computer language, and it interprets what the computer is saying in your language. Like the little man who stands beside the President when he visits other counteries, and translates into English and back. It is called "Basic" because it is the fundamental, basic thing you need to get started. Also, the letters stand for "Beginner's All–purpose Symbolic Instruction Code." It is "Tiny" because this interpreter is smaller (that is, it will run on a smaller computer) than most BASIC interpreters.

But enough history. Let's get started. You will need, besides the basic ELF II, the "Giant Board" with the Monitor and Cassette interface, one or more "4K RAM" boards, an ASCII keyboard (this looks like a typewriter keyboard), and an approved Video Modulator to connect the computer output to your TV set. The 4K RAM board should be configured for addresses 0000–0FFF (see the 4K RAM instruction manual for details), and you should leave the two RAM chips that came with the basic ELF II plugged in. Figure 1 will show you where the important parts of your computer are.

Turn your ELF II on (plug it in, or whatever turns it on). You can tell it is on when you can see two red numbers or letters lit above the number keys. There are three toggle or lock–down switches to the right. All three should be in the position towards yourself (switched off or locked up). We will call this condition **Reset**. When I say "Reset the Computer" I mean put the switches in this position.

Put the TINY BASIC Cassette into your cassette player, and start to play it. Adjust the volume so that when the whistle starts it is comfortably loud. This is probably at 1/2 to 3/4 full volume. Rewind it, and plug the "in" cable from the computer into the earplug (or "external speaker") socket on your cassette player. Push the "1" key one more time (the red digits will change to "00"), then start the player. After ten or fifteen seconds the red digits will start changing rapidly. The program is loading into the computer.

After a short while the digits will stop on "FF". It is possible that they might stop on some other pattern, with the little "Q" light to their right also on, or perhaps they never start. If this happens, there was an error in reading the tape, and you need to start over from the first Reset. If the Q light always comes on, or if the digits never start rapidly changing, perhaps you need to adjust the volume control or the tone control on your cassette player, or perhaps the head is dirty (see the instruction manual for the "Giant Board"). Try again from the first Reset.

When the digits stop without the "Q" light coming on, you are ready to run TINY BASIC. You do not need the cassette player any more for a while (you can turn it off). Reset the computer then push the Run switch away from you again. Push the "RETURN" key on the ASCII keyboard. If your TV set is connected and turned on, you should see something like four dots in the upper left corner of the TV screen, and a fifth dot blinking just slightly below and to the right of the lowest one. If you do not see this, perhaps there is a wire out of place connecting the keyboard or TV to the computer, or the TV is not tuned to the same channel as the modulator. Note: you cannot proceed without a correct load from cassette and the blinking dot on the TV screen.

The blinking dot we will call a **cursor**. Whenever you see it, TINY is waiting for you to type something in. Go on, type something. Type your name. But don't push the RETURN key again — yet.

Try out the various keys on the keyboard, so you get used to what each letter looks like on the screen. Notice that sometimes if you are right at the right edge of the screen, and you type another letter, part of it will be there where the cursor was, and part of the letter will be on the next line at the left edge. You will get used to this, but if it bothers you for now, you can always type a space or two, so that the whole letter appears on the next line.

You will also notice that not all the letters are the same width. We did that to make them easier to read, but still not take up too much space on the screen. Some of the letters have descenders, that hang below the line a little. This makes them touch the tops of the letters on the next line. Again, this was to make the characters as readable as possible without sacrificing space. We hope you don't mind too much.

Your keyboard may have a special key labelled "SHIFT LOCK". If so, you want to be sure it is in the unlocked position (push it a few times to see if that releases it).

As you begin to fill the screen, you will notice that TINY rolls the top line off the screen to make room for a new line at the bottom. This is called **scrolling**, because it works something like an ancient scroll.

Now press the "ESC" key. Notice that the screen immediately scrolls up and you get a colon on the left margin of the new line. Try it again. See how it leaves the cursor dot on the end of the previous line. Type something (like your name), then the ESC key again. We use the ESCape key to escape from the line we are typing. TINY BASIC simply throws that line away and does not look at it again. The dot reminds you that this has happened. Notice that it looks different from a period (it is lower).

Now type an ESC so that the cursor is right next to the colon. Then type a DEL. It works something like the ESC, but if your line has several characters in it, you have to type as many DELs as there are characters before you get a new line. The DEL stands for DELete, and it deletes one character at a time from your line, always the last one you typed. We will get a better idea how this works in the next chapter.

Another special key you will be interested in is not even on your keyboard. You have to fake it by pressing two keys at once. Press the CTRL key down, then while holding that down, press the "L" key. We use this method to erase the screen. Notice, however, that the cursor is in the top corner of the screen. Anything you type is completely lost off the top. You can correct this by typing ESC. This is actually perfectly normal, and later you will see how to use this fact.

Now you know how to speak to your computer. In the next chapter you will see how to get your computer to speak to you. In the rest of this book we will always assume that TINY BASIC is loaded and ready to go. That means that any time you turn off your computer you will have to go through the steps in this chapter again. Since you will probably get more out of this book if you spread it out over several days, you can expect to reload TINY BASIC two or three times at least.

1 — ELF TALKS BACK

In this chapter you will learn how to understand what your ELF II computer is telling you. Be sure it is turned on, and TINY BASIC is loaded and running (review Chapter 0 if you do not remember how to do this). Type an ESCape or two to get the cursor on a new line with only a colon.

In Chapter 0 I told you not to type the RETURN key yet. Well, now you are ready: try it. You got a new line with its prompt (we call the colon on the left end of the line a prompt because it prompts you to type something in). This does not appear particularly exciting. Type your name, followed by a RETURN. TINY BASIC answers!

You should see, on a new line under your name, the line

```
!397
```

and another colon prompt line under that. What does it mean? If you were to look 397 up in the list of error messages in the TINY BASIC user's manual, it would say, "Misspelled keyword." This does not mean you misspelled your name. Since your computer does not know how to spell your name, it could not know whether you mispelled it. In fact, it does not even know what to do with your name. Type some other person's name. You see, you get the same answer. Type HELLO. Same result. The computer does not understand these words, and the response, "!397" is to tell you that the computer does not know what you mean. You may get, from time to time, some message I have not yet told about. I will probably get around to discussing the message a little later, so you might want to write down exactly what you typed and what the message was — it does not stay on the TV screen very long after you start typing again. Most often, of course, when you get a message you were not expecting, it is because you did not type exactly what you thought you did.

There are a few words the computer does understand. Type PRINT and a return key. Every time we want to say something to the computer, we must end the message with a Carriage RETURN. TINY BASIC does not look at what you type until you hit the RETURN key. This time there is no error message, only a new line with a prompt. But nothing in particular was PRINTed by the computer. This is because there was nothing else on the line that told it to print. Type

```
PRINT 4
```

You see, if you tell it to print a number, it prints that number on the next line. Try some other numbers: 0, 1, 2, 100, 57, 12345, 87654. On this last number, you will notice that Tiny did not print the same number you typed. The reason is that the number is too big. In fact, any number larger than 32767 is too big. Tell TINY to PRINT 32767, then tell it to PRINT 32768. You see, it puts a minus sign in front of 32768. If you tell TINY to PRINT 32769 it will respond with -32767. As you continue to put in larger numbers, it works backwards, but with a minus sign. See if you can find a number to type in that TINY BASIC will respond

with –1.

Notice that if you type in PRINT 65536 the computer responds with 0. As it turns out, because this BASIC is "tiny", it thinks that 65536 is the same number as zero. Numbers greater than 65536 (but less than 98304) are the same as numbers greater than zero. Numbers less than 65536 (but greater than 32767) are the same as numbers less than zero. The problem has a valid mathematical interpretation, but for now you must assume that you cannot use numbers greater than 32767.

As you noticed, however, TINY BASIC can print negative numbers. Tell TINY to PRINT –2 or some other negative number. Be sure to use the hyphen key (the keytop shows both the hyphen and the equal "=" symbols) and not the underscore key. If you make a typing error, you can ESCape from the line and type it over. Try telling the computer to PRINT a few other negative numbers. What does it show for –32769? Remember, I said that we have to limit our numbers to less than 32768, because strange things happen to larger numbers.

Before we go any farther in working with numbers, I should say some more about correcting mistakes. You will make typing errors. We all make typing errors. I make them all the time. If it is a gross error, we use the ESC key to cancel the line. But if a finger slips on the last digit of a long line, it is a shame to have to type the whole line back in. So we have what is called a **backspace**. It does not really back up the cursor (remember, this is a TINY BASIC), but it will erase the previous character inside the computer. We use the underscore key for this. The underscore key is usually on the right–hand end of the keyboard (don't confuse it with the hyphen!). If you cannot find the underscore key, use the DEL key instead. Type

PRINT 123_45

Notice that the computer responds with 1245. The "3" was deleted. What do you think will happen if you type two underscores instead of one? Try it. You see, two backspaces back up over two characters. You can back up as far as you like this way. Notice that if you make a mistake while typing a correction, you can back up again. And again, if necessary. TINY does not mind. It is there to obey you. Suppose you tell TINY to back up when there is nothing on the line left to back up over; what would you expect? Try it. Type PRINT then six underscores. The new line prompt is to tell you that you backed up over the colon. Perhaps you might have lost count while backing up, and TINY wants to give you a clean line to try again on.

Well, computers should compute, not just repeat back everything you type in. Let's see if your ELF knows how to add. Type in

PRINT 2+3

Notice that the "+" key also has a semicolon (";") on it. You have to hold the SHIFT key down while typing "+". The answer is (what did you expect?) 5. Try 2–3 and maybe some other addition and subtraction. Notice that we use the same hyphen for subtraction as for negative numbers. For multiplication we use a times symbol which shows up on the screen as

a little "x", but on the keyboard it is a star or asterisk "*". Be sure to hold the shift key down for this one too, or it will be a colon instead. Ask your computer to PRINT 2x3 (you type 2*3). The computer does not have a special symbol for divide, so we use the slash "/". Tell the computer to PRINT 24/6 (that is, 24 divided by 6). Since 24 is exactly divisible by 6, we expect (and get) an answer of 4. But suppose we tell the computer to divide two numbers that do not come out exact; what then? Try 8/3. In grammar school you learned that 8/3 is 2.33333. But TINY BASIC is not able to deal with (decimal) fractions so it just throws them away. It told you that 8/3 is 2 and threw away the .33333.

You can tell TINY to do several operations at once, if you like. Try 1+2+3+4+5 and 89*45/15–6. Now tell it to PRINT 3*4–6/2. What should you expect for the answer? In school you learned that you always do the multiplication and division before addition and subtraction, so TINY does the same thing. It multiplied 3 times 4 and got 12, then divided 6 by 2 and got 3, then finally subtracted the three from the 12 to print 9. Suppose you really wanted it to subtract before dividing? Then you must use parentheses to group the operations in the order you desire. For example, to multiply, then subtract, then divide, you type PRINT (3*4–6)/2 and to do the subtraction first you type PRINT 3*(4–6)/2. You see how each one of these gives you a different answer. Any computation inside parentheses is done first. If you like you can have another set of parentheses inside the parentheses. And another. As long as your request does not get more than maybe three lines long on the screen, TINY will accept as many operations and parentheses as you like. Try

```
PRINT ((((( (((((  (((((  15  )))))  )))))  )))))
```

By now you probably have let some typing errors slip through. And TINY (usually) responded with some message like "!73" or "!395" or "!556" or "!560". Each of these numbers identifies a kind of error that TINY knows how to distinguish. For example, error number 73 says that TINY does not know quite what is wrong with your PRINT command. It may actually print some number (probably the wrong one) before saying so. Error number 556 tells you that TINY was expecting to find a number, but but you had typed something else. Error number 560 says you forgot a right parenthesis. If you forget a left parenthesis in a PRINT command you will probably get error number 73. Error number 395 tells you that you typed something between the colon prompt and the command to PRINT.

You now know to tell TINY BASIC to print a number or the result of some computation on numbers. But suppose you want to print

```
THE ANSWER IS 5
```

TINY BASIC can do this also. You tell it to do so by commanding:

```
PRINT "THE ANSWER IS 5"
```

Notice the quotation marks around the message. Any time you want TINY to print anything other than numbers (including the results of computations), you need quotation marks around

what is printed. Actually what you really want is for TINY to show the message, then do the computation, and print the result. Type in

```
PRINT "THE ANSWER IS  " ; 2+3
```

Notice the space before the second quotation mark and the semicolon after it. The semicolon tells TINY that there is more to be printed on the same line. So TINY prints the message you put in quotation marks, then adds 2+3 and prints the result on the same line. Anything inside quotation marks is printed exactly the way you type it, but if there are no quotation marks, TINY BASIC will perform the specified computations and print the result. You could type

```
PRINT "2+3=";2+3
```

and TINY would respond with

```
2+3=5
```

Notice that TINY BASIC prints exactly what you tell it to print, even if what it prints is not true:

```
PRINT "2+3=7"
```

prints

```
2+3=7
```

even though that is not true. Just because something has been printed by a computer does not make it true. The computer only does obediently what people tell it to do. If your electric bill says you owe $5000 for last month's electricity to your home, it is not the computer's fault. Some person told the computer the wrong thing. When your computer prints the wrong answers, you can be sure it is because you told it to do so. Most of the time you and I and everyone else in this business spends on computers is and will be trying to figure out what it is we told the computer to do that we did not really want to tell it to do. And part of the fun is finding the problem and fixing it. It is kind of like a puzzle, only often much harder.

Now you know how to tell the computer to print messages on the screen, and to compute formulas and print the results. In the next chapter we will see how to tell the computer to do these things later.

2 — RUNNING PROGRAMS

Every time you have typed something into your computer, it responded immediately to do what you told it to do. That is gratifying, but it sometimes is not very useful. In this chapter you will learn how to tell your ELF to remember several things to print, and how later to tell it to print those things.

It is actually quite easy. In front of the command to PRINT something, you type a number (say like 10). When TINY sees the number, it saves the line in the computer's memory. Then you can type another line with a different line number, and TINY will save that one also. First clear the computer memory by typing CLEAR (and a carriage RETURN, of course). Then type in:

```
10 PRINT "2+3=";2+3
20 PRINT (3*24-12)/2;" DAYS HATH SEPTEMBER"
```

Notice that the only thing TINY BASIC prints out after you hit the carriage return on each of these lines is a new line prompt. We need one more thing before we can get this to print out. We need to tell TINY that this is all we want it to do. You do this by typing

```
99 END
```

This line is stored in the computer's memory as the last line. You do not need to type it again once it is in the memory in the right place.

Now I can almost hear you ask, "How do I know these lines were saved in the computer's memory?" Well, we can ask the computer for a list of all the lines in its memory, and see what it says. Type

```
LIST
```

You will see, on your screen, something that looks a little bit like this:

```
10 PRINT "2+3="; 2+
•3
20 PRINT ( 3*24-12 )
/2;" DAYS HATH SEP
TEMBER"
99 END
:
```

Of course you have to realize that there are really only three lines in memory, and that they have been broken up into six pieces for display on your screen. You know these lines are in

memory, because you can PRINT out anything else you wish (without line numbers) or hit the RETURN key seven times and then LIST again, and you will get the same display. What you have done is entered a **program** into your computer's memory.

Now suppose you made a typing error in entering your program into memory. After you hit the carriage return key on the keyboard, there is no amount of ESCaping or backspacing that will correct that error. The line has been stored, error and all, into memory. But all is not lost! Just type the line in again with the same line number, the way you really intended it to be, and the new version will replace the old line in memory. If your mistake is in the line number itself, you will have to type the line with the correct line number, then on a separate line, just the line number of the line that is wrong. If you were lucky and made no mistakes in our little example, you might try changing line 10 to print the sum of 3+4 instead of 2+3. After you do this, LIST the program again to be sure you typed what you wanted.

You can run this program just as easily. Just type RUN with a carriage RETURN, and there, underneath the line you just typed (i.e. under the word, "RUN") will be the results of the two commands you typed in earlier. Type RUN again. And again. You see, as long as the program is in the computer's memory, you can RUN it over and over again. This is one of the useful things about computers. Though it may take a long time to get a program into the computer correctly, after it is, you can run it many times and get exactly the same results.

But suppose those results are not exactly what we wanted. Suppose, for example, that we want a blank line on the display between the two output lines. A blank line, as you recall, happens if you type in the command PRINT with nothing else on the line. If we put a line number on a line that says only PRINT, then when the program is RUN, that command will print a blank line. But how do we get it <u>between</u> the other two commands in the program? You could just retype the whole program with the new line in its proper place, but that is much too much work. Instead we will choose a line number between the other two line numbers — say 15:

```
15 PRINT
```

Type it in, then LIST the program again. Aha! It seems to be in the right place, but the first line ran off the screen before you got a good look. Type LIST again, but this time, after the screen has blinked two or three times, reach over and push the "1" key on the corner of your ELF II. As long as you hold that key down, the display stops moving so you can look at it. When you release the key, the LISTing resumes, but you can always hold it again by pressing the "1" key again.

Anyway, now you can clearly see that line number 15 is between lines 10 and 20. It does not matter at all that this was the last line you typed. In fact you can type lines in any order you please, and TINY BASIC will put them in the correct order. This is why we usually do not start our numbering with line 1 or go up by sequential numbers. Often we will think of something later to add to the program, and if there are unused numbers at the point we want to make the change, we can just type it in. If the line numbers are sequential then we have to retype the program with the new lines inserted. I am much too lazy for that.

Bye the way, don't forget to RUN this new version of the program to see what the effect of line 15 is.

Now let's add another line to the poem. What line number should we use? That's right, 30 is a good number. Make it print "APRIL, JUNE,". List the program to be sure it is right, then RUN it. Add another line to print "AND NOVEMBER." Test it (i.e. Run it, and be sure it looks right). Gee, that first line is getting lost off the top of the display when the last version is run. Why don't you just remove that command from the program. Removing a line is easy: you just type the line number, followed by a carriage return. TINY replaces that line with a line that has nothing on it, and the line vanishes from the program. Try it, then LIST the program.

Now, while you are at it, why not pretty up the first line of the poem? Splitting a word in the middle of a letter is not particularly beautiful, so you should figure out how to divide the phrase into two different lines. Notice that long lines split differently when you LIST them than when you RUN them: we are interested in what it looks like when you RUN. One of the new lines you can number 20 again; the other you might number 25. You will have to type in both lines, because TINY does not know how to make two lines out of one. Do that now, and test the resulting program. Are you happy with it? Why don't you show off your skill with the computer to some appreciative friend. You could clear the screen, type ESCape, then RUN (without the carriage RETURN); when you are ready to demonstrate your program, just type the RETURN key.

You should probably practice thinking up things for your computer to print out on the screen, then writing the programs to print them. Each time it is helpful to remove the old program before putting in the new. Retyping all those line numbers is a bit of a chore, but the whole program can be removed at once by typing the command CLEAR. Do not confuse clearing memory (by the CLEAR command) with clearing the TV screen (by typing control-L).

When you put the next program in, don't forget that the last line must be an END command. If you leave it out, you will get an error message something like

```
!75 AT #0
```

If (maybe I should say When) you make other mistakes in typing your program, TINY will give other error messages, mostly like the ones you have already seen, except that if they are noticed after you type RUN, they will come with a line number to help you find which line has the error. Suppose you program has the line in it somewhere,

```
47 PRINT "HELLO. I AM
```

When you try to run this program and it comes to this line, it will print out

```
HELLO. I AM
!61 AT #47
```

Error number 61 tells you that you left off the close quotation mark. the "AT #47" tells you that the error occurred in line number 47. To look at just that line you can request of TINY:

```
LIST 47
```

or you can request that TINY list several lines by typing two line numbers on the LIST command:

```
LIST 30,50
```

lists all the lines in the program between 30 and 50, inclusive.

Now you can write, modify, and run programs in TINY BASIC. But there is more! In the next chapter you will learn how to store numbers in the computer.

3 — VARIABLES: CHANGING THE COMPUTER'S MIND

So far everything that is stored in the computer's memory is exactly what you typed. Often, however, we would like to tell the computer how to do a problem before we tell it what numbers to use. This is something like the riddle where you are asked to think of a number, then add, subtract, multiply, and divide various quantities without disclosing the number. At the end you are supposed to be surprised that the result can be told you, or that the original number can be deduced from the result. We want to give such riddles to the computer, then maybe tell it what number to think of. This chapter tells you how to do it.

First you need to understand how memory is organized in your computer. TINY BASIC has two places to store things. One of these you already know about, and every time you type in a line with a line number, it goes into that storage area of memory. Internally, it is kind of like a file drawer, where each program line is inserted into its respective place in the file. As more program lines are added, you just move the stop at the back of the drawer back some more to accomodate them. When you take lines out, you crimp the backstop up, so that the rest of the lines in the file will not fall over. Of course it is possible to try to put too much into the file. After a while you simply cannot push the stop back any farther, and TINY BASIC refuses to take the next line. When it does this, it will tell you with error message number 8.

Right now, however, we are interested in the other storage area in memory. It is divided into 26 little boxes like a post office. Each box holds exactly one number. Each box has a name, which is one of the letters of the alphabet. Any time we choose to do so, we can take the number out and look at it, or maybe put another number back into the box. We tell TINY to put a new number into a box with the LET command. For example:

```
LET A=123
```

tells your computer to put the number 123 into the box named "A". You can look at the number in the box named "A" with our friend, the PRINT command:

```
PRINT A
```

And because the box contains a number, we can do anything with it that we might want to do with a number. For example, we can ask the computer to take whatever number is in A, add 5 to it, then print half of the sum:

```
PRINT (A+5)/2
```

If you do that now, after typing in the other lines above, the computer will compute (123+5)/2 which is 64. Suppose, however, you type in

```
LET A=27
PRINT (A+5)/2
```

The computer immediately forgets that the box named "A" ever had 123 in it, and remembers instead that it has 27 in it. The answer is now 16.

It is important to realize that the LET command is not an equation like those you studied in algebra. It may be true that A does equal 27 immediately after the command is obeyed (we say "executed"), but what does it mean to tell the computer

```
LET A=A+2
```

Obviously A never equals A+2 in any mathematics you and I are familiar with. But TINY BASIC understands what that means. Type it in, then PRINT A again. You see, what happened is that TINY saw the line start with "LET A=", and sort of mumbled to itself, "oh, yes, I have to put some new number in A." Then it sees the "A+2" while thinking, "gotta have a number, gotta have a number,..." Of course A+2 is a number: it is the sum of the number in A (which, you recall, we had just stored 27 into) plus two, which is 29. Therefore, TINY stores the number 29, which it just computed, into the box named "A". Now it matters not a whit that there used to be 27 in A; all that matters is that A has 29 in it now. That is why we call thes boxes variables; the value of the number in them can vary from time to time.

Now we can begin to write interesting programs. The idea is to write a program in which we tell the computer what to do with the numbers in certan variables, then when we run the program we tell it what numbers are in those variables.

Let us suppose that you want to order another memory board from Netronics, and you cannot find your calculator to compute the sales tax and handling charges. Oh, let's be efficient, and set it up so you can also order books from BYTE and parts from your favorite surplus store.

Variable P will have the price in cents (no decimal point, remember?), R will have the tax rate in percent, and H will have the handling fees, also in cents. Remembering that the largest number TINY can think about is 32767, we observe that our order must not exceed $327.67. No problem! Netronics boards don't cost very much. We will start with a price of $89.95, which means P will be 8995.

The first thing the program needs to do is calculate the tax on the order. The program will put this number into variable T. If we just multiply the price times the tax rate, we might get a very large number (consider $100 x 5%, which is computed as 10000*5 or 50000, which is too large). So we will be tricky. Besides, it is more fun if you can trick the computer into doing something more than it ought to. We will divide the price by 100, which gives a price in dollars, then multiply that times the tax rate to get the tax in cents on the dollar part of the price:

```
100 LET T=P/100*R
```

But we must not forget the 95 cents part of the price, which is likely to have another nickel

or so of tax. Remember that when you divided the price by 100, the cents part was thrown away. We can depend on that fact, so if we divide the price by 100 then multiply it by 100 again, we will get the dollar part of the price, measured in cents. This can be subtracted from the original price, leaving the cents part of the price, which we tell the program to put in the variable C. I know it sounds confusing, but it's really not too bad — think it through one step at a time: divide 8995 by 100 giving 89 (throw away the remainder 95); multiply 89 times 100, which is 8900. Subtract that from the original 8995 (which is still in P) leaving 95, which is the cents part.

```
110 LET  C=P-(P/100)*100
```

Actually the parentheses are not necessary, because TINY does multiplications and divisions in left-to-right order. Now we still have to calculate the tax on this amount. It will be C*R/100 (remember the tax rate is in percent). But since this throws away the fraction, we need to do the rounding before we divide by 100. Rounding is normally done by adding .5 to the result, then throwing away the fractional part. This is mathematically the same as adding 50 before dividing by 100, so the tax on the cents part is (C*R+50)/100. The parentheses are necessary here. We add this to the tax on the dollars part, and put the sum back into T.

```
120 LET  T=T+(C*R+50)/100
```

Notice that this will add 50 to the product of CxR, divide the result by 100, (throw away the fraction), then add the result to T. We know this because of the natural order for doing addition and division, and because of the parentheses.

Now that we have instructed the computer on how to compute the sales tax, the rest is easy. The check should be made out to the total (in cents) of T+P+H Let's tell the computer to put that into variable M:

```
130 LET  M=P+T+H
```

Now we could just tell the computer to print the result, but an amount in cents looks awfully big, so let's print it out in dollars and cents, with a dollar sign and a decimal point. Yes, I know I said that TINY BASIC cannot compute decimal fractions, but that does not mean you cannot print them! You know how to get the dollar part of M; it is M/100. And you know how to get the cents part alone; it is M-M/100*100. We will print the result in two stages:

```
150 PRINT "$";M/100;".";
160 LET  C=M-M/100*100
170 PRINT  C
```

Notice that on line 150 we are printing three different things, each followed by a semicolon (";"). The semicolon tells TINY that we want to print something else on the same line. So first it prints a dollar sign, then it prints the dollars part of M, then it prints a decimal point, then it notices that there is still more to print, but that you have not yet told it what. When

it gets to line 170 it now knows to print the cents part that we put into C (on line 160) on that same line, and since there is not another semicolon, the line is finished.

We need one more thing: an END statement. I think it is nice also to tell yourself what the number means, so why don't you print out a line explaining the result?

```
200 END
140 PRINT "WRITE CHECK FOR"
```

Of course you must try the program out. Say the memory board costs $89.95 with $3 postage and handling. Assume the tax rate is 5%. To run the program, type

```
LET P=8995
LET H=300
LET R=5
RUN
```

Did your computer give you the answer of $97.45?

Now it is a big nuisance to have to remember to type in LET commands for P, R, and H. Wouldn't it be nice if the computer could simply ask for price, handling, and tax rate? Well, it can. This takes a new command which inputs data from you as the program is running. It is the INPUT command. Try it out just now; type:

```
INPUT X
```

Notice that you got a question mark instead of a colon prompt. TINY is now waiting for you to type a number, which it will then put into the variable X. Type some number (say, 456) and a RETURN. Now that you have your colon prompt back, ask TINY to print out X. Repeat the experiment, but this time with some other number. Do you think you understand the INPUT command?

Let's put it into the program. First we will print out what the computer is expecting, then we will request it as input:

```
10 PRINT "PRICE";
20 INPUT P
30 PRINT "HANDLING";
40 INPUT H
50 PRINT "TAX RATE";
60 INPUT R
```

Notice that each PRINT command ends with a semicolon. This tells TINY that more is to come on this line. What is to come is the question mark of the INPUT command, so each request for input comes with a question mark automatically supplied by TINY. Now RUN the program (without any LETs). See how it asks you for a price? Say you want to buy a $2.95

Lexicon from an ad in BYTE; type in 295 (remember, no decimal point or dollar sign!) and RETURN. Now it asks for Handling. $.75 per book, so type 75. There is no tax on books from New Hampshire, so type in 0. The check is to be for $3.70. Try a few others (type RUN again).

You should not assume, because you get the right answer the first two times you run your program, that it is necessarily correct. Here for example, is a very subtle problem in this program. Perhaps you will notice it as you experiment with various numbers. Or you can hurry on to the next chapter, which also tells you how to fix the problem.

4 — IF AN ELF COULD DECIDE...

In Chapter 3 you learned about changing the numbers stored in the computer's memory while the program is running. In this chapter you will learn about changing the sequence of program commands, while the program is running.

In the last episode, you will recall, we left the hero of the story hanging on the edge of a cliff, ready to go crashing down to his doom with the wrong answer. Did you discover the problem? If not, try telling the program that you want to buy a $235 printer with $20 shipping and no tax. You see, one of the zeros is missing from the cents part. We do not know which zero yet, so let's run it again, but assume the shipping is $20.07. Does that help you to discover the problem? What if the price were $236.03 (try both shipping charges)?

The problem is not so much that you are getting the wrong answer, but that if the cents part is less than 10 the leading zero is not printed. We got a little too tricky! We would like to say to TINY something like, "print the cents part with two digits, even if the first digit is zero." Or maybe, "If the cents part is less than 10, then print an extra zero first."

Gee, that last one looks promising. Suppose TINY understood you when you said

```
IF C<10 THEN PRINT "0"
```

Well, did it? What is in C? Type in the following sequence of commands (notice which ones print out responses):

```
LET C=99
IF C<10 THEN PRINT "IT BETTER NOT BE"
LET C=10
IF C<10 THEN PRINT "IT IS STILL NOT <"
LET C=9
IF C<10 THEN PRINT "AHA!"
LET C=-99
IF C<10 THEN PRINT "WELL, WHAT DID YOU EXPECT?"
```

So it looks like this will solve the problem. We need to insert our correction between lines 160 and 170:

```
165 IF C<10 THEN PRINT 0;
```

Notice that we still end with a semicolon, because the cents in C is still coming. Try the program again with that $235 printer. Can you find any more problems (we call them bugs; getting rid of them is called debugging)? Note: just because you cannot find any bugs in your program does not mean that there are none.

Let's play with the IF command some more. CLEAR out your program from memory, so

you can put a new one in.

I used to have trouble with Roman numerals when I was younger. Maybe you did too. We will write a program to convert decimal numbers to Roman Numerals. What it will do is INPUT a number, then slowly convert it to Roman on one printed line. The number we will work with will be in variable N:

```
10 PRINT "NUMBER";
20 INPUT N
```

As I write this, I am not sure how much memory the program will take. If you have only 4K the program probably cannot process very large numbers. I will write the program backwards, then quit when I run out of space. This is OK, because TINY will put the lines in order as we type them in. I will suppose that line number 500 is a good line to end on (the line numbers are quite arbitrary; I am just guessing that I will not need more lines than will fit in less than 500). We assume that all that is left of the number in N is less than 4 (I will explain how N got that way as we go along):

```
500 END
490 PRINT
480 IF N>0 THEN PRINT "I";
470 IF N>1 THEN PRINT "I";
460 IF N>2 THEN PRINT "I";
```

You can try this much out, if you want to RUN it. When it asks for a number, be sure to type in a number less than 4.

We assumed the number would be less than 4 when the program reached this point, so let's make it so:

```
450 LET N=N-N/4*4
```

This, as you will recall, sets N to the value of the remainder of N divided by (in this case) 4. Four is a special case:

```
440 IF N=4 THEN PRINT "IV";
```

If the number is greater than or equal to five (but less than nine) we print a "V":

```
420 IF N>=5 THEN PRINT "V";
430 LET N=N-N/5*5
```

Notice I could have just as easily written,

```
430 IF N>=5 THEN LET N=N-5
```

but you see, this takes more typing and thus more space in memory. Each character (letter,

digit, etc.) of each line takes one byte of memory. Later I will show you some ways to save memory space. Try the program again. It should take numbers up to 8 now.

I don't think you will have any difficulty seeing that Roman Numeral 9 works like 4 and that the tens work like the ones:

```
400 IF N=9 THEN PRINT "IX";
410 LET N=N-N/9*9
390 LET N=N-N/10*10
380 IF N>=10 THEN PRINT "X";
370 IF N>=20 THEN PRINT "X";
360 IF N>=30 THEN PRINT "X";
350 LET N=N-N/40*40
340 IF N/10=4 THEN PRINT "XL";
330 LET N=N-N/50*50
```

Watch out for the lines that contain the sequence, "LET N=N-N/...". It is awfully easy to lose your place and drop one of the Ns out.

Well that is about all the program you can get into 4K. If you have more memory you are welcome to expand it. But first lets think a little more about how it works (with a demonstration, of course). Type RUN, and when it asks you for a number, type in 47.

Here is how it does it (you might like to follow the program listing while reading this paragraph): The first thing the program does is remainder the number by 50. That is, we are not interested in any part of the number greater than fifty. If there had been more room, perhaps the program could have been expanded to rake numbers up to 5000 or some such. However, the remainder of 47 divided by 50 is 47. No problem. Now 47 divided by 10 is equal to 4, so the rest of line 340 is executed, and an "XL" is printed. The number in N is then remaindered by 40, which is seven. The IFs in lines 360, 370 and 380 are all false, so nothing is printed from them. In fact, the next interesting line is 420, since only then is the IF true, and "V" is printed. Finally, dividing by five the remainder is 2, so 460 is false, but 470 and 480 are true, and two "I"s are printed. Line 490 prints a nothing to end the line. Try another number and see if you understand it.

When we look at this program we see a lot of nearly identical lines, such as for example, lines 460, 470, and 480. Wouldn't it be easier to say something like, "if N>0 then print "I", subtract one from N, and try again; but if N=0 then quit"? Well, each line has a line number on it; do you think the computer could "go to" a line that was not the next line? Let's see how that would work:

```
460 IF N=0 THEN GO TO 490
470 PRINT "I";
480 LET N=N-1
485 GOTO 460
```

Now you see, that takes more lines in the program (we had to add line number 485), but they

are much shorter, so we saved a little space. Perhaps we can get some more savings:

```
360 IF N<10 THEN GO TO 400
370 PRINT "X";
380 LET N=N-10
390 GO TO 360
```

I hope you are trying each set of changes out to convince yourself that the program still works the same.

Now let's try a real tricky modification. So tricky, in fact, that if you show it to a Computer Science expert, he will tell you it is "bad programming structure." Never mind him; he will also tell you TINY BASIC is a crummy language. He just has no appreciation. Besides that, he is jealous of your low-cost computer. What we will do is, if N is 4 or 9 (you can tell if it is by remaindering by 5), we will print a single "I", then add one and run it back through the tens routine (a routine is a few lines of program that does one thing). Of course if it was 4, the tens routine will do nothing and eventually line 420 will print a "V". On the other hand if it was 9, line 360 will see a 10, and the program will print another "X" after that "I". Do you understand? If not, you should try to follow the program through just as if you were the computer. First type in the six lines of changes (below), so you will not forget them.

To understand a complicated program (and also to get a feel for how much work the computer is really doing), we do what is sometimes called "a hand simulation." On a piece of paper make separate columns for each variable the program will be using; in this case there is only one variable of interest: N. On another piece of paper, or on another part of this piece, we will hand-print (in nice computer-like block letters) exactly what the computer would print out.

We will start at line 360 of the program with N=29, so write "29" in the column labeled "N". look at line 360 of the program (you can look at a single line by typing LIST 360) and do whatever it says. In this case it says "IF N<10..." but the number under N on your paper is not less than ten (it is 29), so ignore the rest of the line and go to the next. You can look at the next line by typing LIST 360+1. This says to PRINT an "X" on your output line (on the paper). Next line (LIST 370+1) tells you to LET N=N-10. In other words, start with the "=" sign and figure out N (which is 29 on your paper) minus 10, then cross out the 29 and write "19" under it, since this command puts a new value in the variable N. The next line tells you to GO TO 360, so go LIST 360 again. The third time around N will be 9, which is less than 10, so you GO TO line 400. Now at line 400 you need to calculate the value of "N-N/5*5" to see if it is less than 4 (it isn't), but this is not a LET command so don't change anything in the "N" column on your paper. Just go to the next line, print an "I" on your paper next to the two Xs, then add one to N (that makes 10) and go back to line 360 again. It may seem tiresome, but when you reach the END on line 500 you <u>really</u> understand what happened, and why.

Oh, here are the changes:

```
440
450
```

```
400 IF N-N/5*5<4 THEN GO TO 420
405 PRINT "I";
410 LET N=N+1
415 GO TO 360
```

Once you are convinced that you understand what this program does (and more important, how it does it), you should be able to add a line (how about line number 320?) that prints "L" if the starting number is greater than or equal to 50. If you have 8K of memory or more, see if you can extend our program with its modifications to numbers over 100. Over 1000? Note that it might help to write down on a piece of paper what you are trying to do.

There is one more thing you can do to your program to make it easier to use. We assume that you probably want to test several numbers in the program each time you modify it or demonstrate it to a friend. Wouldn't it be nice if you did not have to type RUN each time? Can you guess what to add to the program to make the computer go back to input another number after it finishes the previous? Yes, we will use a GO TO:

```
495 GO TO 10
```

Gee, that is a lot easier to use! Only one problem — how do you get it to stop? If it is already running you are in a pickle (I will tell you how to get out in a minute). But the clean way to prevent an **endless loop** (a program that has a GO TO back to some earlier line with no way out) is to provide an exit. Notice that for all the other GO TO commands in the program that go backwards, there is always something being done to N so that the next time by the decision affecting that GO TO is different. But in this last case N is always zero when it reaches the GO TO on line 495. But wait, what is the Roman Numeral for zero? What does the program print out? That's right, there is no Roman Numeral for zero. So let's have the program notice that you gave it a zero input, and make that the escape hatch:

```
30 IF N=0 THEN GO TO 500
```

Now actually, line 500 is just the end of the program. We could just as easily tell TINY BASIC to end it right here if N=0:

```
30 IF N=0 THEN END
```

I said I would tell you how to get out of the endless loop. Perhaps you already discovered it accidentally.

If the program is waiting for input (you can tell by the question mark prompt), a space will be ignored, and an ESCape or Return will only start a new line with another question mark. Numbers and letters are accepted as input. The way to get out of this condition is to type something illegal, like a period, followed by the RETURN key. TINY will not know what to do with the period, and will respond with error message number 556 and the line number of the INPUT line.

If the program is busy computing (it blanks out the TV screen while it is computing), you can stop it by pressing any key on your keyboard. I prefer an ESCape, but if you do not want to lose a line of output that might be on the top of the screen, a space will also do. This will cause an error message (number 0), and will also tell you the line number that it was about to do. If you change your mind and you wish you had not stopped it, you can type GO TO and the line number it stopped at, and it will continue executing where it left off. For example, if you break out of the Roman Numeral program by hitting the space bar, it might type

```
!0 AT #390
```

If you had only bumped the keyboard and you want it to continue, type

```
GO TO 390
```

Or if you just want to start over, you can do that by typing RUN.

Incidentally, if you suspect there is a bug in the program, you can always break out (by one of the above techniques), then LIST parts of the program, PRINT any variables to see what is in them, or even make changes in the program or variables, then continue at the line you broke out of.

If you know the program is printing things and you just want to take time to look at them without killing the program, press the "!" key on the corner of your ELF. This works the same as when you are listing.

Now you know quite a bit about programming. In the next chapter we will learn some sophisticated memory-saving techniques, some of which you have already seen in this chapter.

5 — CALL FOR SAVINGS

In the last chapter we found that the program we wanted was too big for the memory you are starting with. Alas! This problem does not go away. But all is not lost. In this chapter we will tackle the worst memory gobbler of all: financial programs. No, not a General Ledger program in your computer (you do not have enough memory), but a checkbook balancing program. To make this fit, you will learn two different ways to save program space.

The first way to save space is a feature which came in with some of the earliest computers (30 years ago). It is similar in concept to the programming trick I showed you in the Roman Numeral program in Chapter 4. But it is a well accepted concept that even your computer scientist friend will approve of. The concept is this: Whenever possible, we use the same routines and lines of program over and over.

One way to do this is to use a variable to count the number of times you have been through the routine, and when you reach some preset count, you quit. This is called a **loop**. We used this technique to count the "X"s and "I"s in the Roman Numeral program. Some computer languages have special ways to set up loops in the program; TINY BASIC does not because you can do the same things with a LET and an IF.

Another way to use the same routine over and over is to put it in what we call a subroutine. A **subroutine** is simply a routine which is set apart like a doctor's office or a shoe store, so that any time the program needs to have done what that subroutine is good at, the program **calls** the subroutine to do it. For example, if you need a pair of shoes, you go to the shoe store; if someone else in your family needs shoes, again, another trip to the shoe store. If someone gets sick, or needs a shot, you go to the doctor's office. You do not go to the doctor's office to buy work boots, and the shoe clerk is not allowed to administer vaccinations. When you finish at the shoe store or the doctor's office or the library, or wherever you went, you **return** home, or maybe you go run some more errands. Now you could have sat at home the whole time, and had your own private shoe clerk bring you shoes, and your own private nurse give you shots, and so on, but that is quite expensive.

Now if you have the general idea, let's see what that means in programming a computer. Each subroutine is made up of lines of program, just as you are already familiar with. Perhaps we might use line numbers in the thousands to keep them out of the way, but that is not necessary. Each place in the program where we need that particular function done, we insert a line with the command GOSUB (for GO to SUBroutine) and the line number of the first line of the subroutine, like a GO TO command. When TINY BASIC sees the GOSUB command, it ma§es a mental note of which line the command is on, then does a GO TO to the subroutine. When the subroutine is finished doing its thing, it ends with a RETURN command, which tells TINY, "find the line that last GOSUB was on, and continue after that."

Perhaps a practical example will help. We are going to do a checkbook balancing program, and we need to print out dollars and cents. You already know how to do this, so I need not get into great detail. But my point is that we want to print out starting balance, the check amount, and the ending balance, each with a dollar sign, decimal point, and so on. But

neither you nor I want to write that dollars–and–cents routine three times. So we will make a subroutine out of it.

We will assume that the dollar amount is in variable D, and the cents amount is in variable C. Thus we can handle balances up to $32767.99 (you are obviously not that rich, and if you were, you would know better than to keep that much money in your checking account). The subroutine starts on line 1010:

```
1010 PRINT "$";D;".";
1020 IF C<10 THEN PRINT 0;
1030 PRINT C
1040 RETURN
```

Notice that in line 1040, you type the word, "R","E","T","U","R","N", then hit the RETURN key. We will put the running balance in variable M (for "money") and the cents of that balance in P (for "pennies"). The first thing to do is print out the balance:

```
110 PRINT "BALANCE IS"
120 LET D=M
130 LET C=P
140 GOSUB 1010
```

Now we want to accept the check amount in dollars and cents. As you know, TINY cannot handle a decimal point, so we will pretend that we took our schooling in Europe and use a comma instead of a period for the decimal point. It turns out that TINY is able to accept several variables of input on a single line if the variables are separated by commas. But this only works inside a program. The INPUT statement must have a line number for it to work:

```
160 PRINT "CHECK OR DEPOSIT"
170 PRINT "DOLS,CTS";
180 INPUT D,C
```

When you use this program, you will type a minus sign in front of the check amounts, a plus in front of the deposits. The minus will come through as a negative amount in D. I suppose you are more likely to write a check for less than $1 than you are to make such a small deposit, so we further assume that if the dollar part of an entry is zero, you probably intended it to be –0 (with non–zero cents). If both dollars and cents are zero, the program quits:

```
190 IF D=0 THEN IF C=0 THEN END
200 IF D<=0 THEN GO TO 310
210 PRINT "DEPOSITING"
220 GOSUB 1010
280 GO TO 110
310 PRINT "WITHDRAWING"
320 LET D=-D
```

```
330 GOSUB 1010
370 IF M>=0 THEN GO TO 110
380 PRINT "OVERDRAFT"
390 GO TO 230
```

Notice also that if the balance tries to go negative, we will reject the check and put the money back into the account. I know, most banks have a policy of accepting a certain amount of overdrawn checks and charging you for them. You can fix the program to do that later. Oh, we still need to do the actual adding and subtracting. We define another subroutine which notices when the cents in P have gone over 100, then bumps them back down and adds $1 to M.

```
1110 LET M=M+P/100
1120 LET P=P-P/100*100
1130 RETURN
```

Depositing is easy. Add the dollars; add the cents; then call the subroutine to check. Withdrawing might go negative in the cents, so we borrow a dollar from M, do the subtraction, then call the subroutine to put the dollar back if it was not used:

```
230 LET P=P+C
240 LET M=M+D
250 GOSUB 1110

340 LET P=P+100-C
350 LET M=M-D-1
360 GOSUB 1110
```

Now if you got all that program into your computer without a memory overflow (message number 8) you have more memory than I give you credit for. If not, don't worry about it; I overflowed too. Let's see what can be done about it.

First I should mention that all the tricks which I will tell you about in the next page or so will not make your program run any better or worse; but they will make it harder for you to read the program and figure out what it is doing. This is not a minor point, because most of your time on the computer, as I said, will be spent in trying to find program bugs. If you can write a program and fit it into memory without doing any of these space-saving tricks, do so. It will save you time and aspirin. But if it just will not fit, then go ahead and squeeze out the space.

I never mentioned it to you, but TINY ignores all spaces in program lines, except inside quoted text for printing. The spaces are there to make it easier to read. The following two lines mean exactly the same thing:

```
200 IF D <= 0 THEN GO TO 310
```

```
200IFD<=0THENGOTO310
```

But notice how much easier it is to read the first line. In a PRINT command, however, everything enclosed by quotes is printed, including the spaces. If you take spaces out there, the printed message will not look the same. Do not sacrifice pretty print messages for space. This makes the difference between a "hack" and an elegant piece of software. You might notice in this regard, that TINY BASIC itself is somewhat of a hack: All those error messages are compressed into numbers, which are not pretty at all!

Already you notice that we could save a couple dozen bytes by leaving spaces out. But that is not all. TINY BASIC knows that LET and IF are the only commands with the "=" character. Well, not all IFs. But every LET command has one letter (the variable name) followed by the "=" followed by some computation. So, if there is no "LET" it is still possible for TINY to figure out that this was meant to be a LET command. Since this is one of the more common commands in a typical program, saving three bytes this way is a great help. The following two lines mean the same thing in TINY BASIC:

```
120 LET D=M
120 D=M
```

Similarly, if a line starts with the letters "PR" it is pretty certain that the next three letters will be "INT". You know that, and TINY knows that, so if those three letters are missing, TINY just pretends that they are there. See how much that saves:

```
1030 PRINT C
1030 PRC
```

Be careful! The command "PRI" prints the number in variable "I", so if you misspell the word "PRINT" TINY may print out the value in variable "I" then stop with error message number 73.

What about IF commands? These are also quite common. Do you have any difficulty understanding that these two lines are saying the same thing?

```
200 IF D<=0 THEN GOTO 310
200 IF D<=0 GOTO 310
```

Well, TINY has no problem either. The word "THEN" may be omitted. In some other computer languages the word "THEN" is necessary to tell where the end of the comparison is. This is not true in TINY BASIC since the rest of the line always begins with a command name (or maybe a LET command, but there is still no confusion).

With these compactions in mind, let's look at the check balancing program again:

```
110 PR"BALANCE IS"
120 D=M
130 C=P
```

```
140 GOSUB1010
160 PR"CHECK OR DEPOSIT"
170 PR"DOLS,CTS";
180 INPUTD,C
190 IFD=0IFC=0END
200 IFD<=0GOTO310
210 PR"DEPOSITING"
220 GOSUB1010
230 P=P+C
240 M=M+D
250 GOSUB1110
280 GOTO110
310 PR"WITHDRAWING"
320 D=-D
330 GOSUB1010
340 P=P+100-C
350 M=M-D-1
360 GOSUB1110
370 IFM>=0GOTO110
380 PR"OVERDRAFT"
390 GOTO230
1010 PR"$";D;".";
1020 IFC<10PR0;
1030 PRC
1040 RETURN
1110 M=M+P/100
1120 P=P-P/100*100
1130 RETURN
```

I will let you think up your own sample data for this one.

Two things you might notice in this program. First, I did not bother to remove the spaces after the line numbers. This is because TINY takes those spaces out anyway. You can see this if you type these two lines:

```
 1 2 3 4 5   E N D
L I     S T   1 2 3 4 5
```

After the first letter in line 12345 all the extra spaces go into memory. The other spaces are removed when TINY discovers that the line begins with a number, so the line number is saved with no spaces in it or after it. The LIST command automatically puts one space back in between the line number and the rest of the line.

The other thing you might have noticed, is that there is no END command at the end of the program. Since the last command in the program always goes back to the line after the GOSUB, TINY never gets a chance to see whether there is an END there or not, and you might as well leave it out if you are pressed for memory space. It is usually a good idea to

put it in otherwise, so that when you look at the program next week you know that it is all there.

Now, if you have a single 4K memory board and the 256 bytes that came with your basic ELF II, you should have about 20 bytes of memory left unfilled. How do I know? I have a little program I use to measure the amount of available memory:

```
1 B=B+2
2 GOSUB1
LET B=4
RUN
END
PR B;" BYTES LEFT"
```

After you type RUN, the two-line program will start eating up memory at the rate of two bytes per GOSUB. None of these GOSUBs ever RETURN, so the saved line number just sits there using its two bytes of memory. Pretty soon (or after a long time, if you have a lot of memory), all the memory space is used up, and the program will error off with some message like "!424 AT #1". Since there was no END in the program, you have to type the "END" in by hand (to clear out all those saved GOSUBs), but then you can ask how many there were. The program carefully kept count in variable B, so that represents the number of remaining bytes in memory. How many of these are actually available depends on what your program does (i.e. if it uses a lot of GOSUBs or has very complicated computations, more memory is needed for that, leaving less available). With a single 4K board, you normally have about 400 bytes of program space. Oh, be sure to remove lines 1 and 2 before you try to run the real program.

There is a way to save memory in a different sense which I think you will find useful. That is, it saves the program in memory, so that you can load it back in next week without retyping the whole thing. It is easy to use; just type:

```
SAVE
```

TINY will tell you to "TURN ON RECORD" and "HIT KEY". You must be sure the Cassette Out terminal from your ELF II is plugged into the microphone jack of your cassette recorder, and that it is in the Record mode. Oh, also put a blnak tape in. If you record onto your TINY BASIC tape, you may have to buy a new one. After starting the tape, press the space bar or some other key on the keyboard, and the program will be saved. When it is finished TINY will come back with a colon prompt and you can turn off the recorder.

To load the program back in, just type LOAD and turn on the cassette recorder in the Playback mode, with the earplug jack connected to the Cassette Input terminal of your ELF. If it reads in ok, you will get a colon prompt. If not, it will tell you "TAPE ERROR" and you must try again.

One more thing I should mention, before you write too many unreadable programs. You may have noticed that I did not use line numbers 100, 1000, or 1100 in the check balancing

program. These were being saved for comments to remind us of what their respective parts of the program were supposed to do:

```
100 REMARK: PRINT BALANCE & READ INPUT
205 REMARK: MAKE DEPOSIT
300 REMARK: WITHDRAW CHECK AMOUNT
373 REMARK: NOTICE OVERDRAFT
1000 REM  SUBROUTINE TO PRINT MONEY IN D&C
1100 REM  SUBROUTINE TO VERIFY CENTS<100
```

TINY BASIC knows that you ought to put notes to yourself in your program, so if the command begins with "REM" the rest of the line is ignored by TINY, but left in the program for you to read. Obviously I did not put these in because they did not fit. That is just about the flimsiest of excuses for not putting remarks in your programs.

You now know almost all there is to know about TINY BASIC and how to write programs. In the next chapter we will look as some of the special ways your TINY BASIC can work with numbers (besides computing and printing).

6 — THE PLOT THICKENS

One of the interesting things you can do with TINY BASIC on your ELF II is to control the various input and output devices of the computer. For example, TINY already displays all its PRINT output on the screen of your TV set; with a little cleverness and a special command you can also draw pictures and graphs. In this chapter you will also learn how to interact with the hex keypad and display.

Let us start with the TV. Every dot on the screen which can be either white or black is represented by a bit in memory. TINY BASIC has a special command to allow you to set any point to white or black. The command is normally used to plot graphs on the screen, so it is called the PLOT command. To use it you need to specify which dot on the screen is to be affected, and what its color will be. The position is given in vertical (between 0 and 41) and horizontal (between 0 and 63) coordinates, and the color is either black (value 0) or white (value 1). Unlike the notation you learned in school, position 0,0 is in the top left corner. Try this program out to get a feel for how the PLOT command positions the dot:

```
10 PRINT "COORDINATES";
20 INPUT X,Y
30 REM CLEAR SCREEN
40 PLOT 12
50 REM PLOT THE POINT
60 PLOT X,Y,1
70 PRINT
80 PRINT
90 GO TO 10
```

When you run this program, give it two numbers (separated by a comma) between 0 and 40. There are two things about this program I have not yet explained. Line 40 is a special form of the PLOT command which clears the whole screen (like when you type control–L). The other thing is that no matter where you plot the dot, exactly two lines under it is the next message asking for another coordinate pair. This is because the PLOT command positions the cursor at the coordinates you selected, then the two PRINT commands moved it down two lines. Experiment with the program a little. See where it places the little dot (the one above the word "COORDINATES", not the blinking cursor) as you give it various numbers. What happens if the numbers you give it are negative or greater than 41 (or 63)? What happens if you take the two PRINT commands in lines 70 and 80 out? What happens if you delete line 40? If you do that, you will see another effect: nothing erases the screen (to blacks) except the scroll–up when you get too near the bottom, so it becomes more and more covered with white.

Let's use the PLOT command to draw a graph for some equation. So that the program will

not die if it gets a point out of range, we will write a subroutine to check the range before plotting (P.S. CLEAR out the old program first):

```
800 REM PLOT X,Y IN RANGE
810 IF Y>-21 IF Y<21 PLOT 20-Y,X+32,1
820 PLOT 0,0,10
830 RETURN
```

Notice that we have two IFs on the same line! You saw this before in the checkbook program. If the first is false, the rest of the line is ignored; if it is true, then the command on the rest of the line is executed. This command happens to be another IF, which if it is false the rest of the line is ignored, and so on. The PLOT command has to have coordinates in the range 0–41, 0–63; it would be nicer if they went from –20 to +20 and –30 to +30, so we compute the necessary offset to make it seem so. Also, we normally expect more positive Y-values to plot at the top of the graph, so Y is subtracted from rather than added to the offset. The next line (line 820) is a trick to let us look at the graph for 1/4 second before continuing.

We need to clear the screen, then draw coordinate axes:

```
100 REM FORMULA PLOTTER
110 PLOT 12
120 C=0
130 X=C
140 Y=0
150 GOSUB 800
160 X=-X
170 Y=-Y
180 IF X<0 GOTO 150
190 IF Y<0 GOTO 150
200 Y=X
210 X=0
220 IF Y>0 GOTO 150
230 C=C+2
240 IF C<32 GOTO 130
```

If you want to see the program so far, add a line "400 END" and RUN it; you can watch the axes grow right before your very eyes! Can you figure out how it works?

Now we are ready for the function to plot. The equation will be a subroutine at line 1000. This is the driver for it:

```
300 X=-30
320 GOSUB 1000
330 GOSUB 800
340 X=X+1
350 IF X<31 GOTO 320
```

```
400 END
```

Let's try a simple parabola:

```
1000 Y=X*X/10-10
1010 RETURN
```

When you RUN this, you will see a solid line across the bottom of the parabola. This results from the squares of low values of X (which is then divided by 10). After you RUN that one, you might want to try a trickier formula:

```
1000 IF X<>0 Y=100/X
```

If X is zero you would get an error stop from the division (you can't divide by zero, you know), so the formula is evaluated only if X is not equal to (i.e. "less or greater than") zero. The one time the formula is not evaluated the previous value for Y (which was -100) is used.

Now try some of your own equations. What can you do to shift the coordinate axes, say if you want more of the first quadrant? Hint: do it in the plotting routine (lines 800+). Experiment!

There are two very obvious Input/Output devices on your ELF II: the hexadecimal keypad and the two-digit hexadecimal display. And TINY is able to talk to these. Start with the display. This is output port number 4 in the machine, so the command to output to it is (obviously enough) OUT 4,(data). For data you can use any number or computed value. Try this:

```
OUT 4,99
```

Do you know why the display shows "63", not "99"? The 99 that you typed was in decimal, while the display is hexadecimal. Maybe this program will help you to learn the difference:

```
10 INPUT H
20 OUT 4,H
30 GOTO 10
```

Run this program, and give it various numbers. What happens if you type in 255? 256? 0? You see, only the low eight bits of the number you type in are displayed. This is the same as saying that only the remainder after dividing by 256 is displayed.

Input from the keypad is a little harder. This is because you cannot just tell it to input the keyboard value without telling TINY what to do with the number it gets from the keys. So there is a special phrase you use to read the keypad, but this phrase is used only where TINY might reasonably expect to find a number. For example, the number might be PRINTed:

```
PRINT INP(4)
```

Or you could use it in a calculation and store the result in a variable:

```
10 LET J=INP(4)-INP(4)/16*6
```

This computation, incidentally, converts the input so that TINY can think of it as a decimal number instead of hexadecimal. Try it in a program, the rest of which might be:

```
20 PRINT J
30 GOTO 10
```

Run it and watch the printout as you push various keys on the hex keypad. You might even notice that some funny values turn up just as you push a key. This is because the program looks at the keys twice, expecting both times to give the same result. This is not necessarily so, especially if you are pushing keys at the time. You can remedy this problem by modifying the program to read the keys into a variable (with a LET), then computing with the variable instead of multiple INP references. See if you can make it work.

You may have guessed by now that if INP(4) and OUT4 refer to the hex keyboard and the display, what about INP(1), INP(2), OUT5, OUT6, etc.? Well they do exist. INP(1) turns the TV interface circuit on (but don't expect to see much!). The others relate to various input and output ports you may or may not have attached. This is not really the place to discuss what they can do. Experiment!

There is one other function which you will find interesting. A **function** in TINY BASIC is one of the special forms consisting of a function name followed by an **argument** (some numerical value) in parentheses. INP(4) is a function. The other interesting one is used to create an unpredictable number. We call it a **random number**, but it is not truly random, because it repeats after 32768 times. However, that is seldom enough that you will not notice it. The number that TINY creates is between zero and some upper limit, which you specify in the argument. For example, RND(100) has a value somewhere between 0 and 99, inclusive. Here is a little program I like to leave My ELF doing when it has nothing better to do:

```
10 REM TWINKLE
20 OUT 4,RND(256)
30 LET T=RND(5)/4
40 PLOT RND(42),RND(64),T
50 IF T=0 GOTO 20
60 PRINT
70 GOTO 30
```

After a half hour or so the TV takes on a kind of starry look.

A friend of mine uses TINY BASIC to teach his kids their arithmetic. The program

generates a problem, and asks for the answer. If it is right the kid gets some sort of brownie points. The following is only an outline of how the program might work:

```
100 REM  TUTOR
120 LET  A=RND(9)+1
130 LET  B=RND(9)+1
140 LET  F=RND(3)
150 PRINT
160 PRINT "",A
170 PRINT "              ";
180 GOSUB 500+F*100
200 PRINT
210 PRINT "              ";
220 PLOT 95
230 PLOT 95
240 PRINT
250 PRINT
260 PRINT "RESULT";
300 INPUT  G
310 IF  G=R  GOTO 400
320 PRINT "NO, ";A;
330 GOSUB 500+F*100
350 PRINT "=";R;", NOT ";G
360 E=E+1
370 GOTO 120
400 PRINT "CORRECT!"
410 GOTO 120
500 REM  ADD
510 PRINT "+";B;
520 LET  R=A+B
530 RETURN
600 REM  SUBTRACT
610 PRINT "-";B;
620 LET  R=A-B
630 RETURN
700 REM  MULTIPLY
710 PRINT "*";B;
720 LET  R=A*B
730 RETURN
```

A few lines in this program deserve some discussion. Line number 160 prints nothing, in quotes (open quote followed immediately by close quote), then has a comma before the variable name A. The purpose of the comma is to center the next item in the middle of the TV screen (in this case; if you have already PRINTed past the center, it goes to the left margin of the next line). But the comma cannot be the first thing in the PRINT statement.

Line 180 looks a little strange until you realize it computes a line number. If F is zero, the GOSUB goes to line 500; if F is 1 it goes to 600; if F is 2 it goes to line 700. F can only be 0, 1, or 2 because of line 140.

I think you should be able to figure the rest of this program out yourself. You probably already noticed that it is too big for your 4K memory as it is, but you know how to squeeze it in (and it will fit, with trimming). However, be sure you leave lines 50, 600, and 700 in (you can trim them to just REM) or you will get error number 46.

This program, as it stands, is probably not suitable for kids. It should check to be swre the subtractions do not result in negative numbers, and you may want to grade the difficulty of the problems. Computer Assisted Instruction (called "CAI" for short) turns out to be as big a can of worms as financial problems.

Now that you understand how TINY BASIC works, the next step in learning to write programs is to write programs. What is it you want your computer to do for you? Lay out on paper (in English) step by step, exactly what you want the computer to do. Then rewrite those steps in rules that explain (to you or another human; do not use TINY BASIC yet) exactly how it is to be done. Finally, convert these rules into the TINY BASIC statements to do the job. Type them in and try it out. Programming is like driving: you have to know where you want to go, but the best way to learn is by doing it. Don't worry about making mistakes; TINY doesn't care, and who else will know, if you discover and fix them?

Now that you have a good feel for using TINY BASIC, you will probably want to go back and read the User's Manual. There are some things in it I did not tell you about, and it is a little easier to find things in it than in this tutorial.

If this sounds like the end, it should. The second part of this book deals with esoteric programming techniques which are probably to complicated for you to handle right now. First get some experience writing programs, then come back and read the rest of this book.

PART TWO

GETTING THE MOST OUT OF TINY BASIC

This part of the book is designed for users who have had some experience programming. If you are just getting started, STOP — Read the first part again, do all the exercises, and practice using TINY BASIC for a few weeks. Write your own programs. Get a book of games and try converting them to TINY. Only then will you be able to understand what I am about to say in the rest of this book.

Netronics TINY BASIC was designed to be a small but powerful language for hobbyists. It allows the user to write and debug quite a variety of prorams in a language more "natural" than hexadecimal absolute, and programs written in TINY are reasonably compact. Because the language is small, it is not as convenient for some applications as perhaps a larger BASIC might be, but the enterprising programmer will find that there is very little that cannot be done from TINY with only occasional recourse to machine language. This is, in fact, as it should be: the high level language provides the framework for the whole program, and the individual esoteric functions done in machine language fill in the gaps.

The USR Function

Perhaps the least understood feature of TINY BASIC is the machine language subroutine call facility.

First, how do subroutines work? In 1802 machine language a subroutine may be called with the SEP instruction. This changes the Program Counter to a different address register, while keeping the return address in the original Program Counter register. This is rather limiting in the number of subroutines which may be called, so there are tricks to save the return addresses on a stack so that the address registers may be re-used. These tricks are described in the RCA reference manual for the 1802 (MPM-201A) and are not important to the discussion here.

When the subroutine has finished its operation it executes another SEP instruction to return control to the program that called it. Depending on what function the subroutine is to perform, data may be passed to the subroutine by the calling program in one or more of the CPU registers, or results may be passed back from the subroutine to the main program in the same way. If the subroutine requires more data than will fit in the registers then memory is used, and the registers contain either addresses or more data. In some cases the

subroutine has no need to pass data back and forth, so the contents of the registers may be ignored.

If the main program and the subroutine are both written in TINY BASIC you simply use the GOSUB and RETURN commands to call and return from the subroutine. This is no problem. But suppose the main program is written in TINY and the subroutine is written in machine language? The GOSUB command in TINY is not implemented internally with a SEP instruction, so it cannot be used. This is rather the purpose of the USR function.

The USR function call may be written with up to three arguments. The first of these is always the address of the subroutine to be called. If you refer to USR(12345) it is the same as if you had written a sequence of instructions to load decimal 12345 into address register R3, followed by SEP R3; The computer saves its return address in a register (you may assume it is in R5) and jumps to the subroutine at (decimal) address 12345 with P=3.

So now we can get to the subroutine from a TINY BASIC program. Getting back is easy. The subroutine simply executes a SEP R5 instruction, and TINY BASIC resumes from where it left off. For those of you which worry about such things, TINY BASIC does use the Standard Call and Return Technique described by RCA, so the return address is actually in R6. R5 actually points to another subroutine whose sole function is to copy the address out of R6 into R3, and to pop a new address off the stack into R6. But this need not concern you unless your machine language routine also needs to call other subroutines.

If you want to pass data from TINY to the subroutine in the CPU registers, you may do that also. This is the purpose of the second and third arguments of the USR function call. If you write a second argument in the call, this is evaluated and placed in R8; if you write a third argument it goes into RA; the low byte of the last argument is also placed in the accumulator (the D register). If there are results from the subroutine's operation, they may be returned in RA.1 and the D register, and TINY will use them as the value of the function (D contains the low byte of the 16-bit result). Thus writing the TINY BASIC statement

```
LET P = USR ( 12345, 0, 13 )
```

is approximately equivalent to writing in machine language

```
LDI #30
PHI R3
LDI #39
PLO R3
LDI 00
PHI R8
PLO R8
LDI P
PLO RD
LDI 13
SEP R3
STR RD
```

Now actually there are some discrepancies. As I said, the program does not go back

immediately. Also, TINY only works with 16–bit numbers, though I did not show what happens to RA.1. If you have trouble understanding 1802 machine language, you will probably want to work through A SHORT COURSE IN PROGRAMMING, also available from Netronics.

It is important to realize that the three arguments in the USR function are expressions. That is, any valid combination of (decimal) numbers, variables, or function calls joined together by arithmetic operators can be used in any argument. The following is a perfectly valid statement in TINY BASIC:

```
13 P=P+0*USR(256+24,USR(256+20,47),13)
```

It happens that memory address 0114 is the machine language routine for the PEEK function, and 0118 (decimal 280) is the POKE routine. When this line is executed, the inner USR call occurs first, jumping to the PEEK subroutine to look at the contents of memory location 002F; this byte is returned as its value, which is passed immediately as the second argument of the outer call, which stores a carriage return at the memory location addressed by that byte. We are not interested in any result data from that store operation, so the result is multiplied by 0 (giving zero) and added to some variable (in this case P), which leaves that variable unchanged. We could also have written (in this case)

```
13 POKE PEEK (47), 13
```

What kinds of things can we use the USR function for? As we saw in the example above, we can use it to do the PEEK and POKE operations, though it is hardly worth the trouble. Until you get around to writing your own machine language subroutines, you can use it for certain types of input and output.

Input & Output Via USR

As we saw in the first part of this book, you can directly access the hardware Input and Output ports of your ELF II by the INP function and the OUT command in TINY BASIC. This allows you, for example, to look at the last character keyed in on the ASCII keyboard, if you have one connected. But it gives you you no way to wait for a new keyin, and it is no help at all if you have a serial terminal connected.

On the other hand, you can use the USR function to directly access the character input and output routines that TINY uses, but you need to be careful that the characters do not come faster than your TINY BASIC program can take them. The following program inputs characters, converts lower case letters to capitals, then outputs the results:

```
10 REM READ ONE CHARACTER
20 A=USR(256+6)
30 REMOVE PARITY FOR TESTING
40 A=A-A/128*128
50 REM IF L.C., MAKE CAPS
```

```
60 IF A>96 IF A<123 THEN A=A-32
70 REM OUTPUT IT
80 A=USR(256+9,A,A)
90 GO TO 10
```

Because of the timing limitations of direct character input, it may be preferable to use the buffered line input controlled by the INPUT statement of TINY. Obviously for input of numbers and expressions there is no question, but for arbitrary text input it is also useful, with a little help from the PEEK function. The only requirement is that the first non-blank characters be a number or (capital) letter. Then the command

```
300 INPUT X
```

where we do not care about the value in X, will read a line into the line buffer, affording the operator (that's you) the line editing facilities (backspace and cancel), and put what TINY thinks is the first number of the line into the variable X. Now, remembering that the line buffer is in 0030 to 0078 (approximately; the ending address varies with the length of the line), we can use the PEEK function to examine the characters at our leisure. To read the next line it is essential to convince the line scanner in TINY that it has reached the end of the input line. Location 002F normally contains the current pointer to the input line; if it points to a carriage return the next INPUT statement will read a new line, so all that is needed is to store a carriage return (decimal 13) in the buffer memory location pointed to by this address (see line 13 above).

In a similar fashion we can access the cassette routines to save some part of memory, or to reload it. The cassette save routine is at location 2557 (decimal); the following command saves the contents of the TV display buffer (locations 0DB0-0F08) on cassette:

```
A=A+0*USR(2557,3848,3504)
```

Here the first argument is the address of the routine, the second argument is the ending memory address, and the third is the starting memory address. This will save the actual dots of the display, which can be reloaded into another program (like the sample routine in the Basic ELF II instruction manual) for display. You can also save any data areas you may have set up (more about these later). You should be aware, however, that the machine language routine does not put out the "TURN ON RECORD" message, and the cassette recorder must be turned on immediately when this command is executed (watch the Q light; it will come on when it starts to record the leader).

The cassette load routine was also not designed for this application, but it will work, if you understand what is going on. If the data loads correctly you will get error stop #556 (syntax error), but if there is a tape read error, no message results. I know it's backwards, but as I said, it was not designed for this; I only thought of it while writing this section. The address is 2554, and the following command will read a block from the cassette into a memory buffer whose address is in variable B:

```
IF USR(2554,1,B)*0=0 THEN PRINT "TAPE ERROR"
```

Notice that no ending address is specified in this call. That is because the routine used in TINY reads until an error occurs, or two consecutive bytes of zero are read. Sorry about that! All those pretty pictures on the screen you can save, but you cannot reload into memory in TINY, because they are mostly zeros (all the black space is zero). You can, however, save and reload a data block from some other part of memory (if it has no consecutive zeros; machine language code usually meets this requirement).

Strings

As we have seen, character input is not such a difficult proposition with a little help from the USR and PEEK functions. (Character output was always easy in the PRINT statement). What about storing and manipulating strings of characters?

If we are careful, we can fill up the beginning of the TINY BASIC program with long REM statements and use them to hold character strings (this allows them to be initialized when the program is typed in). For example:

```
2 REMTHIS IS A 50-CHARACTER DATA STRING FOR USE IN TINY
3 REM0        1         2         3         4         5
4 REM12345678901234567890123456789012345678901234567890
5 REM...IT TAKES 56 BYTES IN MEMORY:  2 FOR THE LINE #,
6 REM.....3 FOR THE "REM", AND ONE FOR THE TERMINAL CR.
```

If you insert one line in front to GOTO the first program line, then your program may run just a little bit faster and you do not need the letters REM at the beginning of each line (though you still need the line number and the carriage return). If you are careful, you can POKE the carriage returns out of all but the last line and the line numbers from all but the first line (replace them with data characters), and it will look like a single line to the interpreter. Under no circumstances should you use a carriage return (decimal 13) as a data character; if you do, none of the GOTOs, GOSUBs or RETURNs in your program will work.

Gee, you say, if it weren't for that last caveat, I could use the same technique for storing arrays of numbers.

Arrays

So the question arises, can PEEK and POKE get around the fact that TINY BASIC does not have arrays? The answer is of course, yes. Obviously there is no memory left in your system after TINY has made its memory grab. The possibility that one of the numbers in the array might take on a value of 13 means you cannot use the program space. What else is there? Remember the memory bounds in 0020-0023. Once you have initialized TINY (with the Cold Start), you can put any memory limits you wish in here and restart with a warm start,

and TINY will stay out of the rest of memory. Now you have room for array data, subroutines, or anything else.

You can let the variable A hold the starting address of an array and N the number of elements, and a bubble sort would look like this:

```
500 LET J=1
510 LET K=0
520 IF PEEK(A+J) >= PEEK(A+J-1) GOTO 540
525 K=PEEK(A+J)+256
530 POKE A+J,PEEK(A+J-1)
535 POKE A+J-1,K
540 J=J-1
550 IF J<N THEN GOTO 520
560 IF K<>0 GOTO 500
570 END
```

Of course this is not the most efficient sort routine and it will be veerrry sloooow. But it is probably faster than writing one in machine language, even though the machine language version would execute much faster. There are better sorting algorithms which you can code into TINY; I did not use one of them because they are more complicated.

The Stack

A kind of sneaky place to store data is the GOSUB stack. There are two ways to do this without messing with the Warm Start. But first let us think about the rationale.

When you execute a GOSUB (or, more precisely, when TINY executes one), the line number of the GOSUB is saved on a stack which grows downward from the end of the user space. Each GOSUB makes the stack grow by two bytes, and each RETURN pops off the most recent saved address, to shrink the stack by two bytes. Incidentally, because the line number is saved and not the physical location in memory, you do not need to worry about making changes to your program in case of an error stop within a subroutine. Just don't remove the line that contains an unRETURNed subroutine, unless you are willing to put up with TINY's complaint.

The average program seldom needs to nest subroutines (i.e. calling subroutines from within subroutines) more than five or ten levels deep, and many computer systems are designed with a built-in limitation on the number of subroutines that may be nested. The 8008 CPU was limited to eight levels; the 6502 is limited to about 120. Many BASIC interpreters specify some maximum. I tend to feel that stack space, like most other resources, obeys Parkinson's Law: the requirements will expand to exhaust the available resource. Accordingly, the TINY BASIC subroutine nest capacity is limited only by the amount of available memory. This is an important concept. If my program is small (the program and stack contend for the same space), I can execute hundreds or even thousands of GOSUBs before the stack fills up. If there are no corresponding RETURN statements, all that memory just sits there doing nothing.

If you read your User's Manual carefully, you will recall that memory locations 0026–0027 point to the top of the GOSUB stack. Actually they point to the next byte not yet used. The difference between that address and the end of memory (found in 0022–0023) is exactly the number of bytes in the stack. One greater than the value of the top–of–stack pointer is the address of the first byte in the stack.

If you know how many bytes of data space you need, the first thing you can do is execute half that many GOSUBs:

```
400 REM B IS THE NUMBER OF BYTES NEEDED
410 LET B=B–2
420 IF B> –2 THEN GOSUB 410
430 REM SIMPLE, ISN'T IT?
```

Be careful that you do not try to call this as a subroutine, because the return address will be buried under several hundred "420"s. If you were to add the line,

```
440 RETURN
```

the entire stack would be emptied before you got back to the calling GOSUB. Remember also that if you execute an END command the stack is cleared, but an error stop or a break will not affect it. Before you start this program you should be sure the stack is clear by typing END; otherwise a few times through the GOSUB loop and you will run out of memory.

If you are careful to limit it to the main program, you can grab bytes out of the stack as the need arises. Whether you allocate memory with one big grab, or a little at a time, you may use the PEEK and POKE operations to get at it.

The other way for using the stack for storing data is a little more prodigal of memory, but it runs faster. It also has the advantage of avoiding the POKE command, in case that still scares you. It works by effectively encoding the data in the return address line numbers themselves. The data is accessed in true stack format: last in, first out. I used this technique successfully in implementing a recursive program in TINY BASIC.

This method works best with the computed GOTO techniques described later, but the following example will illustrate the principle: Assume that the variable Q may take on the values (–1, 0, +1), and it is desired to stack Q for later use. Where this requirement occurs, use a GOTO (not a GOSUB!) to jump to the following subroutine:

```
3000 REM SAVE Q ON STACK
3010 IF Q<0 THEN GOTO 3100
3020 IF Q>0 THEN GOTO 3150
3050 REM Q=0. SAVE IT.
3060 GOSUB 3200
3070 REM RECOVER Q
3080 Q=0
3090 GOTO 3220
3100 REM Q<0. SAVE IT.
```

```
3110 GOSUB 3200
3120 REM RECOVER Q
3130 Q=-1
3140 GOTO 3220
3150 REM Q>0. SAVE IT.
3160 GOSUB 3200
3170 REM RECOVER Q
3180 Q=1
3190 GOTO 3220
3200 REM EXIT TO (SAVE) CALLER
3210 GOTO . . .
3220 REM EXIT TO (RECOVER) CALLER
3230 GOTO . . .
```

When the main program wishes to save Q, it jumps to the entry (line 3000), which selects one of the three GOSUBs. These all converge on line 3200, which simply jumps back to the calling routine; the information in Q has been saved on the stack. To recover the saved value of Q it is necessary only to execute a RETURN. Depending on which GOSUB was previously selected, execution returns to the next line, which sets Q to the appropriate value, then jumps back to the calling routine (with a GOTO again!). Q may be resaved as many times as you like (and as you have memory for) without recovering the previous values. When you finally do execute a RETURN you get the most recently saved value of Q.

For larger numbers, the GOSUBs may be nested, each saving one bit (or digit) of the number. The following routine saves arbitrary numbers, but in the worst case requires 36 bytes of stack for each number (for numbers less than -16383):

```
1470 REM SAVE A VALUE FROM V
1480 IF V>=0 THEN GOTO 1490
1482 LET V=-1-V
1484 GOSUB 1490
1486 LET V=-1-V
1488 RETURN
1490 IF V>V/2*2 THEN GOTO 1510
1500 GOSUB 1520
1502 LET V=V+V
1504 RETURN
1510 GOSUB 1520
1512 LET V=V+V+1
1514 RETURN
1520 IF V=0 THEN GOTO 1550
1522 LET V=V/2
1524 GOTO 1490
1550 REM GO ON TO USE V FOR OTHER THINGS
```

Note that this subroutine is designed to be placed in the path between the calling routine and

some subroutine which re-uses the variable V. When the subroutine returns, it returns throught the restoral part of this routine, which eventually returns to the main program with V restored. The subroutine which starts at line 1550 is assumed to be recursive, that is, it may call on itself through this save routine, so that any number of instances of V may be saved on the stack. The only requirement is that to return, it must first set V to 0 so that the restoral routine will function correctly. Alternatively we could change line 1550 to jump to the start of the subroutine with a GOSUB:

```
1550 GOSUB . . .
1552 LET V=0
1554 RETURN
```

This requires another two bytes on the stack, but it removes the restriction on the exit conditions of the recursive subroutine.

If you expect to put a hundred or more numbers on the stack in this way you might consider packing them more tightly. If you use ten GOSUBs and divide by 10 instead of 2, the numbers will take one third the stack space. Divide by 41 and any number will fit in three GOSUBs, but the program gets rather long.

Bigger Numbers

Sixteen bits is only good for integers 0-65535 or (-32768) to (+32767). This is fine for games and control applications, but sometimes we would like to handle fractional numbers (like dollars and cents) or very large range numbers as in scientific notation. Let's face it: regular BASIC has spoiled us. Granted. But if you could balance your checkbook in TINY BASIC you might not cringe so much when someone asks you what your computer is good for.

One common way to handle dollars and cents is to treat it as an integer number of cents. That would be OK if your balance never went over $327.67, but that seems a little unreasonable. Instead you can break it up into separate numbers for dollars and cents as in Chapter 5 in the first part of this book. This allows you balance to go up to $32,767.99, which is good enough for most of us. I will not dwell on this example here, but the idea can be extended. You can handle numbers as large as you like, putting up to four digits in each piece.

A similar technique may be used to do floating point arithmetic. The exponent part is held in one variable, say E, and the fractional part is held in one or more additional variables. In the following example we will use a four-digit fractional part in M, adding to it a number in F and N:

```
1000 REM FLOATING POINT ADD FOR TINY BASIC
1010 IF E-4>F THEN RETURN
1020 IF N=0 THEN RETURN
1030 IF E+4<F THEN LET M=0
1040 IF M=0 THEN LET E=F
1050 IF E=F GOTO 1130
```

```
1060 IF E>F GOTO 1100
1070 E=E+1
1080 M=M/10
1090 GOTO 1040
1100 F=F+1
1110 N=N/10
1120 GOTO 1020
1130 M=M+N
1140 IF M=0 THEN E=0
1150 IF M=0 RETURN
1160 IF M>9999 THEN GOTO 1230
1170 IF M>999 RETURN
1180 IF M<-9999 THEN GOTO 1230
1190 IF M<-999 RETURN
1200 M=M*10
1210 E=E-1
1220 GOTO 1170
1230 E=E+1
1240 M=M/10
1250 RETURN
```

This subroutine uses decimal normalization; by changing the divisors and multipliers appropriately, it can be made into a binary, hexadecimal, or even ternary floating point machine. By using the multiple precision techniques described in the checkbook balancing example, greater precision can be obtained in the fractional part.

Computed GOTO

One of the more powerful features of TINY BASIC is the computed line address for GOTO and GOSUB statements. I once saw a TINY BASIC program which had several large blocks of the program devoted to sequences of IF statements of the form,

```
110 IF A=1 GOTO 1000
120 IF A=2 GOTO 2000
130 IF A=3 GOTO 3000
140 IF A=4 GOTO 4000
150 GOTO 100
```

Now there is nothing wrong with this form of program, but I'm too lazy to type all that, and besides, I could not get his whole program into my memory. Instead of lines 110 to 140 above, the single line

```
125 IF A>0 IF A<5 GOTO A*1000
```

keyword is the first tested, so it becomes the fastest-executing statement, whereas the other form must be tested against all 17 keywords before it is assumed to be an assignment statement.

Another way to speed up a program depends on the fact that constant numbers are converted to binary each time they are used, while variables are fetched and used directly with no conversion. If you use the same constant over and over and you do not otherwise use all the variables, assigning that number to one of the spare variables will make the program both shorter and faster. You can even make the assignment in an unnumbered line; the variables keep their values until explicitly changed.

Finally it should be noted that GOTOs, GOSUBs and RETURNS always search the program from the beginning for their respective line numbers. Put the speed-sensitive part of the program near the front, and the infrequently used routines (setup, error messages, and the like) at the end. This way the GOTOs have fewer line numbers to wade through, so they will run faster.

As we have seen, there is not much that TINY BASIC cannot do (except maybe go fast). Sure, it is somewhat tedious to write all that extra code to get bigger numbers or strings or arrays, but you can always code up subroutines which can be used in several different programs (like the floating point add, lines 1000-1250), then save them off on cassette.

Remember, your computer (with TINY BASIC in it) is limited only by your imagination.