Radio Shack

CAT. NO. 62-2016
FIVE DOLLARS AND NINETY-FIVE CENTS

BASIC COMPUTER LANGUAGE

It's easier than you think!

232 PAGES OF PROGRAMMED INSTRUCTION AND USER PROGRAMS

If you don't know a thing about computers, congratulations! This book was written especially for you. Its fresh approach makes computer programming easy to understand because it actually makes learning fun.

SEE WHAT OTHER USERS SAY:

"I found this book easy to understand and most helpful in using my TRS-80."

Peter Nero
Conductor/Pianist

"Just great! Super! Now that's the way to teach BASIC. The heck with all the esoteric useless info. Computers are FUN and that's the name of the game."

Lawrenceville, Georgia

"At last, a book that assumes the user doesn't know from "Boo" about programming . . . It's in a language I could understand."

Chicago, Illinois

"I have been working with computers since 1960. During that time I've studied scores of assorted books and manuals relating to computers and various languages. This manual is the best I've seen."

Tucson, Arizona

A Personal Note from the Author

This is not a conventional book. There are plenty of good conventional books, and plenty that are not so good.

This book is written specifically for people who don't know anything about computers, and who don't want to be dazzled by fancy footwork from someone who does. It is written to teach you how to use the BASIC language (that's the simplest and most popular microcomputer language) and start you on a fast track to becoming a competent programmer. To that end, every fair and unfair, conventional and unconventional, flamboyant and ridiculous technique I could think of was used. I want you to have fun with a computer! I don't want you to be afraid of them, because there is nothing to fear.

The only restraints put on this book were good taste and a genuine attempt not to insult your intelligence. Beyond that, it contains no "snow jobs", no efforts to impress or intimidate you, and no attempt to sell you anything except the idea that computers are just not all that hard to learn to use.

Sit back, relax, read slowly as though savoring a good novel, and above all, let your imagination wander. I'll supply you with all the routine facts and techniques you need. The real enjoyment begins when your imagination starts the creative juices flowing and a computer becomes a tool in your own hands. You become its master - not the other way around. At that point it evolves from just a box of parts into an extension of your personality.

Dr. David A. Lien
San Diego - 1977

FIRST EDITION
SECOND PRINTING — 1978

All rights reserved. Reproduction or use, without express permission, of editorial or pictorial content, in any manner, is prohibited. No patent liability is assumed with respect to the use of the information contained herein. While every precaution has been taken in the preparation of this book, the publisher assumes no responsibility for errors or omissions. Neither is any liability assumed for damages resulting from the use of the information contained herein.

© Copyright 1977, Radio Shack,
A Division of Tandy Corporation,
Fort Worth, Texas 76102, U.S.A.

Printed in the United States of America

Table of Contents

This User's Manual and You 4

Part A: Chapter 1 Computer Etiquette. PRINT, NEW, and RUN 7
Chapter 2 Expanding a Program. LIST, REM, END, LET, WHAT?, HOW? 11
Chapter 3 Math Operators 17
Chapter 4 Scientific Notation 23
Chapter 5 Order of Operations. Use of Parentheses 25
Chapter 6 Relational Operators. IF-THEN, GOTO 29
Chapter 7 INPUTting Data 33
Chapter 8 Calculator Mode. MEM, SORRY 37
Chapter 9 Using Cassette Tape for Mass Storage. CLOAD, CSAVE 41
Chapter 10 Loops. FOR-NEXT, STEP, CLS, Break Key 45
Chapter 11 Timer Loops. LIST ###, RUN ###, STOP, CONT 53
Chapter 12 Formatting Output with TAB 61
Chapter 13 Nested Loops 65
Chapter 14 INT Function 69
Chapter 15 More Branching Statements. Subroutines. ON-GOTO, GOSUB, ON-GOSUB, RETURN 77
Chapter 16 READ, DATA, RESTORE. String Variables A$ and B$ 85
Chapter 17 ABS Function 93
Chapter 18 Level I Shorthand Dialect. Multiple-Statement Lines 95
Chapter 19 Generating Random Numbers with RND 99
Chapter 20 Video Display Graphics. SET, RESET 105
Chapter 21 Arrays Using A(X) 123
Chapter 22 Advanced Graphics. PRINT AT, POINT 133
Chapter 23 Flowcharting 141
Chapter 24 Logical Operators. * (AND), + (OR) 147
Chapter 25 Advanced Subroutines 155
Chapter 26 Debugging Programs 165

Part B: Sample Answers to Programming Exercises in Chapters 3 through 25 177

Part C: Prepared User's Programs 201

Appendix:
Appendix A: Prepared User Subroutines 216
Appendix B: Cassette Data Files 221
Appendix C: Combined Function and ROM Test 225

SUMMARY OF LEVEL I BASIC 232

This Book and You

This Book has been written for the average person who has no experience with a Computer.

While it was originally written to go with Radio Shack's TRS–80 Microcomputer, it has been met with such enthusiasm, we realized it had a much broader appeal and application. And of course it really is not a book about a product -- it is a book about a computer language ... How to Learn to Use Basic, and have fun doing it. So, when we talk about the TRS–80, just keep in mind that almost all microcomputers which run on BASIC will function in the same manner.

We've deliberately kept our style light and humorous (some may even say it's corny!) . . . we think this will make your learning experience fun.
(and why **shouldn't** learning be fun ... ?)

The Manual is organized in three basic sections:

A. 26 Chapters which introduce you to various capabilities of the Computer; in small enough bites so you won't choke. These Chapters include numerous little check points and examples (as we get deeper into the book the examples get deeper).
At the end of the Chapters we've given some Exercises – to give you a chance to try out your knowledge ON YOUR OWN.

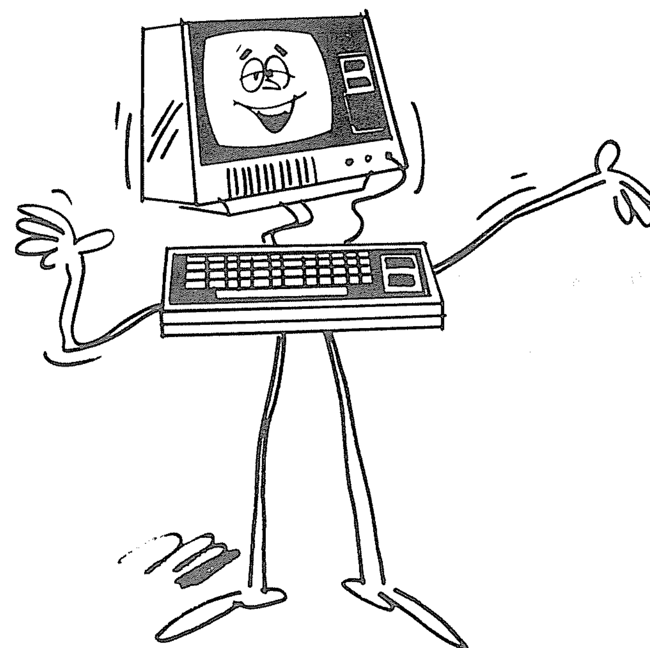
B. A section with sample answers to the Exercises in each Chapter. You can see how you make out with your attempts at programming.

C. A section with some **User's Programs** – some good examples of interesting and practical programs (some for fun, some for business, some for education, etc.).

We've also included some helpful information in an APPENDIX.

The Manual is written in a style where the Computer assists you in learning (educators might like to call it "Computer Assisted Instruction" ... we'll try to avoid trying to impress you with that type of fancy wording).

So, on you go – and we hope you have as much fun with this book as we did preparing it (we had some headaches too ... hope you don't have any of those).



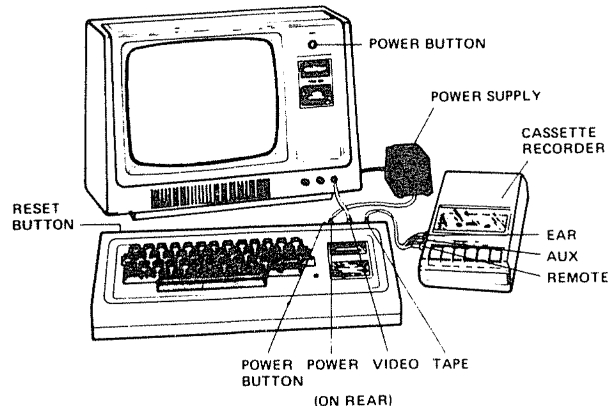
NOTE: Some forms of BASIC may have a few more words or a few less words in their beginning level -- but almost all of what we talk about in this book applies to all BASIC language microcomputers.

SETTING UP A MICROCOMPUTER SYSTEM

Every Microcomputer system needs to have proper connections made. Sometimes it just requires plugging in a power cord; other systems might require more connections. The next two pages list requirements for connecting Radio Shack's TRS-80 Microcomputer. You might want to read through them to get an idea of how a typical system is set up.

Connecting the Video Display and Keyboard:

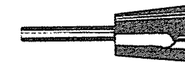
1. Connect the power cord from the Video Display to a source of 120 volts, 60 Hz AC power. Note that one prong of the AC plug is wider than the other — the wide prong should go into the widest slot of the AC socket.
   NOTE: If you use an AC extension cord, you may not be able to plug the Display's power cord in. Do not attempt to force **this wide prong into the extension cord**; use a wall outlet if at all possible.
2. Connect the power cord of the Power Supply to a source of 120 volts, 60 Hz AC power.
3. Connect the gray cable from the front of the Video Monitor to the **VIDEO** jack on the back of the Keyboard Assembly. **Take care to line up the pins correctly** (the plug fits only one way).
   NOTE: Before the next step, be sure the **POWER** switch on the back of the Keyboard is off (button out).
4. Connect the gray cable from the Power Supply to the **POWER** jack on the back of the Keyboard Assembly. **Again, take care to mate the connection correctly**.



Connecting the Cassette Recorder:

NOTE: You do not need to connect the Cassette Recorder unless you plan to record programs or to load taped programs into the TRS-80.

1. Load batteries into the CTR-41 as described in the Manual. Or make connections for 120 volt AC power.
2. Connect the short cable (DIN plug on one end and 3 plugs on the other) to the **TAPE** jack on the back of the Keyboard Assembly. **Be sure you get the plug to mate correctly**.
3. The 3 plugs on the other end of this cable are for connecting to the CTR-41.
   A. Connect the **black** plug into the EAR jack on the side of the CTR-41. This connection provides the output signal from the CTR-41 to the TRS-80 (for loading Tape programs into the TRS-80).
   B. Connect the larger **gray plug** into the AUX jack on the CTR-41. This connection provides the recording signal to record programs from the TRS-80 onto the CTR-41's tape.
   Also, plug the Dummy Plug (provided with the CTR-41) into the MIC jack (this disconnects the built-in Mic so it won't pick-up sounds while you are loading tapes).
   NOTE: Be sure you always use the Dummy Plug when loading programs onto tape (Recording).



Dummy Plug

   C. Connect the smaller gray plug into the REM jack on the CTR-41. This allows the TRS-80 to automatically control the CTR-41's motor (turn tape motion on and off for recording and playing tapes).

Notes On Using The Recorder

There are a number of things you should be aware of as you use the Cassette Tape System: (Some of this will be covered in greater detail in Chapter 9 . . . but some of you can't wait till then . . . *can you!)*

1. To Play a tape (load a taped program into the TRS-80), **you must have the CTR-41's Volume control set to 7 to 8**. Then press the CTR-41's PLAY key and then type CLOAD on the TRS-80 and ENTER this command. This will start the tape motion. An * will appear on the top line of the Monitor; a second * will blink, indicating the program is loading. When loading is done, the TRS-80 will automatically turn the CTR-41 off and flash READY on the screen. You are then ready to RUN the program (type in and hit ENTER).

2. To Record a program from the TRS-80, press the CTR-41's RECORD and PLAY keys simultaneously. Then type CSAVE on the TRS-80 and ENTER this command. When the program has been recorded, the TRS-80 will automatically turn the CTR-41 off and flash READY on the screen. Now you have your program on tape (it still is in the TRS-80 also). Many computer users make a second or even a third recording of the tape, just to be sure they have a good recording.

   NOTE: To load the full 4K of RAM in the TRS-80 takes less than 3 minutes of tape. Short programs will take only a few seconds of tape.

3. Use the CTR-41's Tape Counter to aid you in locating programs on tapes.
4. For best results, use Radio Shack's special 10 minute Computer Tape Cassettes (especially designed for recording computer programs). If you use standard audio tape cassettes, be sure to use top quality, such as Realistic SUPERTAPE. Keep in mind that audio cassettes have lead-ins on both ends (blue non-magnetic mylar material) — **you can not record on the leader portion of the tape.** Advance the tape past the leader before recording a program.
5. When you are not going to use a CTR-41 for loading or recording programs, do not leave RECORD or PLAY keys down (press STOP).
6. To REWIND or FAST-Forward a cassette, you must disconnect the plug from the REM jack (with REM jack connected, the TRS-80 controls tape motion).
7. If you want to save a taped program permanently, break off the erase protect tab on the cassette (see CTR-41 Manual).
8. Do not expose recorded tapes to magnetic fields. Avoid placing your tapes near the Power Supply.
9. To check if a tape has a program recorded on it, you can disconnect the plug from the EAR jack (also disconnect the REM plug so you can control the CTR-41 with the keys) and Play the tape; you'll hear the program material from the speaker.

TURNING THE SYSTEM ON

Turn on the Video Display by pressing the **POWER** button. Turn on the TRS-80 Keyboard by pressing the **POWER** button on the back (next to the POWER jack); the red LED just to the right of the Keyboard should light up and the screen should show READY . Adjust C (contrast) and B (brightness) controls on the front of the Display for the sharpest display. Set Brightness so the background is gray and the words are white. Do not set Brightness too high.

If Display does not show READY , press the Keyboard's **POWER** switch off and on again.

NOTE: There is a Reset button inside a door at the left rear of the Keyboard assembly. This Reset button can be used to unlock a looping program or if the TRS-80 does not turn off a cassette or in other such abnormal program situations.

One More Thought –

You're all ready now, right? Well, maybe. But let's just prepare you for the book with one more thought . . .

How do you "talk" to a Computer? In Binary Numbers? In Electronics (is there such a language . . .)? In English . . .?

Well, we use a simplified form of English — it's called the BASIC Language (Beginners All-purpose Symbolic Instruction Code). (There are lots of other "computer languages", but this is the easiest.) This Manual covers Radio Shack's LEVEL I BASIC. It is similar to most other forms of beginning BASIC.

As you go through this Manual you'll learn the different words of this simple computer language — and how to punctuate *(VERY IMPORTANT)* — and how to apply all of it for fun and practical benefit. It's an easy language to learn — but remember, **you've got to use the language and words that your Microcomputer understands** (we'll be giving you some examples of wrong language use and you'll see what happens).

Chapter 1

Computer Etiquette

From the moment you turn it on, the TRS-80 follows a well-defined set of rules for coping with you, the "master." This makes it an especially easy computer to use. To a large extent, all you have to do is say the right thing (via the keyboard) at the right time. Of course, there are lots of "right things" to say; putting them together for a purpose is called **programming**.

This set of rules is permanently stored in the Computer in two programs, called the **monitor** and the **interpreter**.

In this chapter we're going to start a conversation with the TRS-80 by teaching it a few simple social graces. At the same time, you'll be learning the fundamentals of computer etiquette. You'll even write, wonder of wonders, your first TRS-80 computer program!

Getting READY

1. Connect the keyboard-computer, Video Display and Power Supply as explained in the previous section. Plug Video Display and Power Supply into 12Ø-volt AC outlets.
2. Press POWER button on Video Display and the back of the Keyboard. Give the video tube a few seconds to warm up.
3. READY
   >— should appear in the upper left corner of the screen. Press the **ENTER** key several times to produce a column of READY messages. The Computer is trying to tell you something: "I'm ready — it's your turn to do something!"

ENTER is a monitor command. It tells the Computer to take a look at whatever you've typed on the screen. In step 3, you didn't type anything, so the Computer just comes back with another READY .

To make sure you start off with a clean slate — erasing all traces of prior programs or tests — type NEW and press **ENTER** . The Computer will respond by erasing the screen and printing

```
READY
>—
```

at the top of the screen.

Now type in P.M. and **ENTER** . This is a test to see that the Computer powered up properly. The display should read:

```
P.M.
3583
```

Hit [P] key, [•] key, [M] key and [•] key. Don't use shift key — letters are always capital for TRS-80.

If you have 8K of memory, the number should be 7679. With 16K of memory, it should be 15871.

If the number is not 3583, turn the Computer off, using the pushbutton on the right rear corner of the keyboard. Wait about 1Ø seconds and turn it on again. Repeat the test and verify that the number is 3583.

It's good practice to perform this simple test whenever you turn on the TRS-80. Always type NEW and ENTER before performing the test. As for what the test tests — we'll wait a few chapters for that!

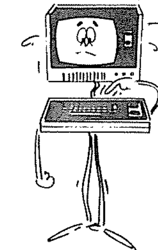
Just What Is a Computer Program?

A program is a sequence of instructions that the Computer stores until we command it to follow (or "execute") those instructions. Programs for the TRS-80 are written in a language called BASIC — and that should give you an idea of how easy it is to learn!

Let's write a simple one-line program to let the TRS-80 introduce itself. First be sure the last line on the screen shows a >, which we call the "prompt". This is the Computer's way of saying, "Go ahead — do something!" Now type the following line, **exactly** as shown:

```
1Ø PRINT "HELLO THERE. I AM YOUR NEW TRS-8Ø MICROCOMPUTER!"
```

Do **not** hit ENTER key yet!



"EXTRA CAREFUL"

You don't have to use the SHIFT key to get a capital letter — that's the only kind of letter the TRS-80 uses. However, some of the keys do have two characters printed on them. Use the SHIFT key to get the upper characters — like the " marks and the exclamation point(!).

If you made a mistake, don't worry — it's much easier to correct typing errors on the TRS-80 than it is on a regular typewriter. No rubber erasers or white paint to fuss with! Just use the backspace key ◄. Each time you press this key, the rightmost character will be erased. If your error was at the beginning of the line, you'll have to erase your way back to that point and then retype the rest of the line.

See the little "dash" (—) that moves across the screen as you type in a letter? This is the "cursor". It lets you know exactly where the next character you type will be printed on the screen. Pushing the space bar moves the cursor along one space, without printing anything.

Now go back and examine **VERY CAREFULLY** what you have typed:

1. Did you enclose everything after the word PRINT in quotation marks?
2. Are there any extra quotation marks?

If everything's okay, you can press ENTER . The > prompt will reappear. The Computer is telling you, "Fine — what's next?"

If you press ENTER a second time, the screen will read:

```
READY
```

This is reassuring, but not necessary — as long as the bottom item on the screen is the >prompt, you know it's "your turn."

If It's Too Late

If you find an error after you've typed a line and pressed ENTER , you cannot use the ◄ backspace key to correct it. Instead, retype the **entire** line correctly. As soon as you ENTER the line, it will replace the incorrect one. This is because both of them share the same starting number (in this case, 1Ø).

"Allow me to introduce myself."

Now we'll tell the Computer to execute our program. The BASIC command for this is simple: RUN. So type RUN and press ENTER . If you made no mistakes, the display will read:

```
HELLO THERE. I AM YOUR NEW TRS-8Ø MICROCOMPUTER!
```

If this isn't what you got, go back and try it again. If RUN still doesn't produce the greeting, there's something wrong in your program. Type NEW to clear it out and type in the one-line program again.

If it did work — let out a yell! "HEY MA, IT WORKS!" This is very important, because now that you have tasted success with a computer, it may be the last you are heard from in some time.

Note that the word PRINT was not displayed, nor were the quotation marks. They are part of the program's instructions and we didn't intend for them to be printed.

Type the word RUN again and hit ENTER .

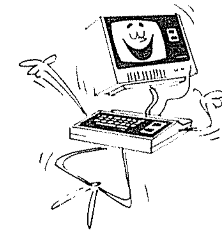
Type RUN to your heart's content, watching the magic machine do as it's told, over and over. When you feel you've really got the hang of all this, get up and stretch, walk around the room, look out the window — the whole act. Because you'll soon get hooked and you won't want to take time for such things later on.

Learned in Chapter 1

Commands	Statements	Miscellaneous
BREAK	PRINT	>prompt
ENTER		— cursor
NEW		← backspace key
RUN		" " quotation marks

We'll put a list like this at the end of each chapter. Use it as a checkpoint to make sure you didn't miss anything.

Maybe you're wondering what's the difference between BASIC commands and BASIC statements. Commands are executed as soon as you type them in and press ENTER . Statements are put in to programs and are only executed after you type the command RUN .



"HEY MA, IT WORKS!"

Whether you're typing in a program, or giving direct commands like RUN, you've got to hit ENTER to tell the Computer to take a look at what you've typed and act accordingly.

Special message for people who can't resist the urge to play around with the computer and skip around in this book. *(There always are a few!)* It's possible to "lose control" of the Computer, so that it won't give you a READY message when you press ENTER. To regain control, just press BREAK , then ENTER . If that doesn't work, find the Reset button inside the left rear corner of the TRS-80 and push it. There!

As you exercise your TRS-80, you'll note that with SHIFT you get some symbol/characters that are not used with LEVEL I (Eg. ^ [ ]) although they can be inside a print statement.

Notes:

Chapter 2

How To Expand A Program

You now have a program in the Computer. (If you turned it off between lessons, fire it up again and type in line 1∅ from Chapter 1.) It's only a one-liner, but let's expand it by adding a second line. In BASIC, every line in the program **must** have a number, and the program is executed in order from the smallest number to the largest. Type:

```
2∅ PRINT "YOU CALLED, MASTER. DO YOU HAVE A COMMAND?"
```

Check it carefully — especially the quote marks, then

```
RUN ENTER
```

If all was correct, the screen will read:

```
HELLO THERE. I AM YOUR NEW TRS-8∅ MICROCOMPUTER!
YOU CALLED, MASTER. DO YOU HAVE A COMMAND?
```

If it ran OK, answer the question by typing

```
YES ENTER
```

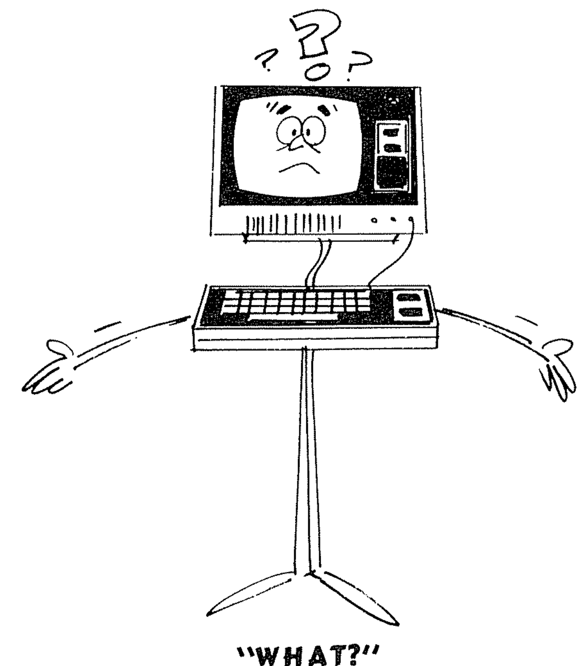
Oh — sorry about that! It "bombed", didn't it? The screen said,

```
WHAT?
```

This error message is the result of a built-in troubleshooter which lets you know when you've said the wrong thing (or the right thing at the wrong time). The WHAT? message on the screen says, "No-no, dummy — the program you wrote doesn't have any way for me to accept an answer just because it asked a question" — or words to that effect.

A later lesson will cover another error message. Meanwhile, if you get a WHAT?, HOW? or SORRY, go back and examine the program for an error. Your "YES" answer here was used purposely to show an error message. Later on, we'll program the Computer to accept a "YES" or "NO" answer and act on it.

> Have you noticed that we use ∅ for the number zero — so you can distinguish between the letter and number. The Video Display does it this way — so we'll do the same throughout the Manual.



"WHAT?"

And the Program Grows

It is customary, traditional (and all that) to space the lines in a program 1Ø numbers apart. Note that your two-line program has the numbers 1Ø and 2Ø. The reason . . . it's much easier to modify a program if you leave room to insert new lines in-between the old ones. There is no benefit to numbering the lines more closely (like 1,2,3,4). **Don't do it.**

Look at the Video Display. Let's decide we'd rather not have the two lines so close together, but would like to have space between them. Type in the new line:

```
15 PRINT ENTER
```

Then

```
RUN ENTER
```

It should now read:

```
HELLO THERE. I AM YOUR NEW TRS-8Ø MICROCOMPUTER!

YOU CALLED, MASTER. DO YOU HAVE A COMMAND?
```

Looks neater, doesn't it? But what about line 15??? It says PRINT. PRINT what???? Well — print **nothing**. That's what followed PRINT , and that's just what it printed. But in the process of printing nothing it automatically activated the carriage return, and inserted a space between the printing ordered in lines 1Ø and 2Ø. So *that's* how we insert a space.

Didn't that room between lines 1Ø and 2Ø come in handy?

Another important statement is REM, which stands for REMARK. It is often convenient to insert REMarks into a program. Why? So you or someone else can refer to them later, to help you remember complicated programming details, or even what the program's for and how to use it. It's like having a scratch-pad or notebook built-in to your program. When you tell the Computer to execute the program by typing RUN and ENTER , it will skip right over any numbered line which begins with the statement REM. The REM statement will have no effect on the program. Insert the following:

```
5 REM *THIS IS MY FIRST COMPUTER PROGRAM* ENTER
```

then

```
RUN ENTER
```

The run should read just like the last run, totally unaffected by the presence of line 5. Did it?

You might be wondering why the asterisks(*) in line #5? The answer is . . . they're just for decoration: *let's give this operation some class!* Remember, anything that is typed on a line following REM is ignored by the Computer.

Well, this programming business is getting complicated and I've already forgotten what is in our "big" program. How can we get a listing of what our program now contains? Easy. A new BASIC command. Type

```
LIST ENTER
```

The screen should read:

```
 5 REM *THIS IS MY FIRST COMPUTER PROGRAM*
10 PRINT "HELLO THERE. I AM YOUR NEW TRS-80 MICROCOMPUTER!"
15 PRINT
20 PRINT "YOU CALLED, MASTER. DO YOU HAVE A COMMAND?"
```

You can call for a LIST any time the prompt appears on the screen.

Where is the END of the program?

The end of a program is, quite naturally, the last statement you want the Computer to execute. Most computers require you to place an END statement after this point, so the computer will know it's finished. But with your TRS-80, an END statement is optional — you can put it in or leave it out. Remember though, if you want to run your BASIC programs on fussier computers, you'll probably need the END statement.

When we get into more complex programs, you'll want to use END statements to force the Computer to stop at specified points — so actually, END comes in very handy even with the TRS-80.

Let's take a close look at END. By the rules governing its use, most dialects of BASIC **which require** END insist that it be the last statement in a program, telling the computer "That's all, folks." By tradition, it is given the number 99, or 999, or 9999 (or larger), depending on the largest number the specific computer will accept. Your RADIO SHACK computer accepts Line numbers up to 32767.

Let's add an END statement to our program.
Type in:

```
99 END ENTER
```

then

```
RUN ENTER
```

The sample run should read:

```
HELLO THERE. I AM YOUR NEW TRS-80 MICROCOMPUTER!

YOU CALLED, MASTER. DO YOU HAVE A COMMAND?
```

"Why didn't the word END print?" **Answer**: Because nothing is printed unless it is the "object" of a PRINT statement. So how could we get the Computer to print THE END at the end of the program execution? Think for a minute before reading on.

```
98 PRINT " THE END"
```

This will work if line #98 is the last PRINT statement in your program.

Erasing Without Replacing

Just for fun, let's move the END statement from line 99 to the largest usable line number, 32767. This requires two steps.

The first is to erase line 99. Note that we're not just making a change or correcting an error in line 99 — we want to completely eliminate it from the program. Easier done than said: Type:

```
99
```

Then ENTER

The line is erased. How can we be sure? Think about this now. Got it??? Sure — "pull" a LIST of the entire program by typing

```
LIST ENTER
```

The screen should show the program with lines 5, 1Ø, 15, 2Ø and 98. .99 should be gone. Any entire line can be erased the same way.

The second step is just as easy. Type

```
32767 END ENTER
```

. . . and the new line is entered. Pull a listing of the program to see if it was. Was it??? Now RUN the program to see if moving the END statement changed anything. Did it??? It shouldn't have.

Other Uses for END

Move END from #32767 to line #17, then RUN. What happened? It ENDed the RUN after printing line 1Ø and a space. RUN it several times.

Now move END to line 13 and RUN . Then to line 8 and RUN. Do you see the effect END has, depending where it is placed (even temporarily) in a program?

Another Error Message

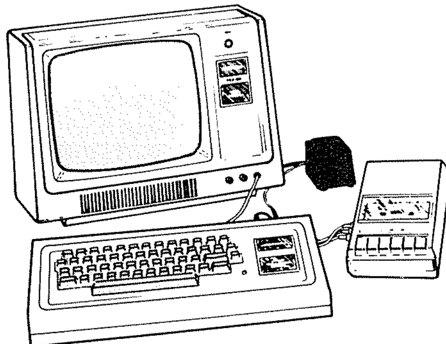
Let's cause a different error message to appear. Move the optional END statement from line 8 to line 5ØØØØ. The Computer should come back with an error message

```
HOW?
```

It is saying "I am very patient with you humans and will obey your every command as long as it is within my ability. Line numbers above 32767 are beyond my ability, so just **HOW** do you expect me to obey?" Pretty smart, this computer.

Learned in Lesson 2

Commands	Statements	Miscellaneous
LIST	PRINT (Space)	Error Messages
	REM	WHAT?
	END	HOW?
		Line Numbering



In general, a HOW? message means, "I understand your instructions, but they're asking me to do something that's impossible." The WHAT? error message, on the other hand, means, "I don't understand your instruction — either the grammar is wrong or you're using words that aren't in my vocabulary."

Notes:

Chapter 3

"But Can It Do Math?"

Yes, it can. Basic arithmetic is a snap for the TRS-80. So are highly complex math calculations — when you write special programs to perform them. (More on this later.)

Of course, we also need that old favorite, the equals sign (=). But wait - - the BASIC language is particular about how we use this sign! Math expressions (like 1 + 2 * 5) can only go on the right-hand side of the equals sign; the left-hand side is reserved for the "variable name". This is the name we give to the result of the math expression. (This all may seem a little strange, but it's really quite simple, as you'll discover in the next few pages.)

LEVEL I BASIC uses the four fundamental arithmetic operations, plus a fifth which is just a modifications of two of the others.

1. Addition, using the symbol **+**
2. Subtraction, using the symbol **—**
   (See — nothing to this. Just like grade school. I wonder whatever happened to old Miss . . . Well, ahem — anyway)
3. Multiplication, using the special symbol *
   (Oh drat, I knew this was too easy to be true!)
4. Division, using the symbol **/**
   (Well, at least it's simpler than the old ÷ symbol)
5. Negation (meaning "multiply-times-minus-one"), using the symbol **—**

Now that wasn't too bad, was it? Be careful. You **cannot** use an "X" for multiplication. Unfortunately, a long time ago a mathematician decided to use "X", which is a letter, to mean multiply. We use letters for other things, so it's much less confusing to use a "*" for multiplication. Confusion is one thing a computer can't tolerate.

So, to computers, "*" is the only symbol which means multiply. After using it a while, you, too, may feel we should do away with X as a symbol for multiplication.

Putting all this together in a program is not difficult, so let's do it. First, we have to erase the "resident program" from the Computer's memory.

"Resident program" is computer talk for "what's already in there".

Type the command

```
NEW ENTER
```

then type

```
LIST ENTER
```

to check that there's nothing left in memory. The Computer should come back with a simple >.

Putting the Beast to Work

We will now use the Computer for some very simple problem-solving. That means using equations — *oh* – *panic*. But then, an equation is just a little statement that says what's on one side of the equals sign amounts to the same as what's on the other side.

That can't get too bad (it says here).

We're going to use that old standby equation,

"Distance traveled equals Rate of travel times Time spent traveling."

If it's been a few years, you might want to sit on the end of a log and contemplate that for awhile.

To shorten the equation, lets choose letters (called variables) to stand for the three quantities. Then we can rewrite the equation as a BASIC statement acceptable to the TRS-80:

```
40 D = R * T
```

What's that 40 doing there? That's the program line number. Remember, every step in a program has to have one. We chose 40, but another number would have done just as well. The extra spaces in the line are there just to make the equation easier for us to read; the TRS-80 ignores them. Later, when you write very long programs, you'll probably want to eliminate extra spaces, because they take up memory space. For now, they may be helpful, so leave them in.

We can use any of the 26 letters from A through Z to identify the values we know as well as those we want to figure out. Whenever you can, it's a good idea to chose letters that remind you of the things they stand for — like the D, R, and T of the Distance, Rate, Time equation.

To further complicate this very simple example, we will point out now that there's an optional way of writing the equation, using the BASIC statement LET:

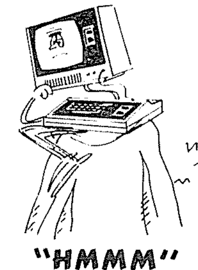
```
40 LET D = R * T
```

This use of LET reminds us that making D equal R times T was **our** choice, rather than an eternal truth like 1 + 1 = 2. Some computers are fussy, and always require the use of LET with programmed equations. Your TRS-80 says, "Have it your way".

Okay — let's complete the program.

Assume:

Distance (in miles) = Rate (in miles per hour) multipled by Time (in hours). How far is it from Boston to San Diego if a jet plane traveling at an average speed of 500 miles per hour makes the trip in 6 hours?



"HMMM"

Remember, we have to use the * for multiplication.

Here's what line 40 means to the Computer: "Take the values of R and T, multiply them together, and assign the resulting value to the variable D. So until further notice, D is equal to the result of R times T."

We could not reverse the equation and write, R*T = D. This would have no meaning for the Computer. Remember, the left hand side of the equation is reserved for variable names (whichever letter we choose). The right hand side is the place to put math expressions involving numbers, operators, and known variables.

(Yes, I know you can do that one in your head but that's not the point!)

Type in the following:

```
10 REM * DISTANCE, RATE, TIME PROBLEM * ENTER

20 R = 500 ENTER

30 T = 6 ENTER

40 D = R * T ENTER
```

Check the program carefully, then

```
RUN ENTER
```

Hum de dum . . . . ho-hum. . . . . (this sure is a slow computer).

```
READY
```

All it says is READY . *The Computer doesn't work!*

Yes it does. **It worked just fine.** The Computer multiplied 500 times 6 just like we told it, and came up with the answer of 3000 miles. But we forgot to tell it to give *us* the answer. Sorry about that.

Yes, yes . . . we know the distance from Boston to San Diego is closer to 2500 miles — but we took a quick detour via Bermuda (besides, 3000 is an easier number to be working with)!

Can you finish this program without help? It only takes one more line. Give it a good try before reading on for the answer. That way, the answer will mean more to you. (Hint: We've already used PRINT to print messages in quotes. What would happen if we said 50 PRINT "D"? . . . No, we want the **value** of D, not "D" itself. Hmmm, what happens when we get rid of the quotes?)

DON'T READ BEYOND THIS POINT UNTIL YOU'VE WORKED ON THE ABOVE EXERCISE!

Look in Part B of this Manual for an answer for this 1st Exercise. Also some notes and ideas.

Well, the answer of 3000 is correct, but its "presentation" was no more inspiring than the printout from a hand calculator. This inevitably leads us back to where we first started this foray into the unknown — the PRINT statement.

Note that we said in line 50 PRINT D. There were no quotes around the letter D like we had used before. The reason is simple but fairly profound. If we want the Computer to print **the exact words** we specify, we enclose them in quotes. If we want it to print the **value** of a variable, in this case D, we leave the quotes off. That simple message is worth serious thought before continuing on.

Did you think seriously about it?! . . . Then on you go!

Now suppose we want to include both the **value** of something *and* some **exact words** on th same line. Pay attention, as you will be doing more and more program design yourself, and PRINT statements give beginners more trouble than any other single part of computer programming. Type in the following:

(REMEMBER: Typing in a statement with a line number that already is in use erases the original statement entirely — and that's what we want to do here.)

```
50 PRINT "THE DISTANCE (IN MILES) IS", D  ENTER
```

Then

```
RUN ENTER
```

The display should appear:

```
THE DISTANCE (IN MILES) IS          3000
```

How about that! The message enclosed in quotes is printed exactly as we specified, and the letter gave us the value of D. The comma told the Computer that we wanted it to print two separate items on the same line. We can tell it to print up to four items on the same line, simply by inserting commas between them.

With this in mind, see if you can change line 50 so the computer finishes the program with the following message:

```
THE DISTANCE IS     3000                                MILES.
```

Break up the quoted message into two parts, and put the variable in between them on the PRINT line.

```
50 PRINT "THE DISTANCE IS", D, "MILES."
```

Now what about all that extra space on the printout line? The reason for it is that the computer divides up the screen width into four zones of 15 characters each. When a PRINT statement contains two or more items separated by commas, the computer automatically prints the items in different print zones. Automatic zoning is a very convenient method of outputting tabular information, and we'll explore the subject further later on.

It's possible to eliminate all that extra space in the output from our Distance, Rate, Time program. Retype the last version of line 50, substituting semi-colons (;) for commas throughout the line.

(Careful — don't replace the period with a semi-colon.)

```
RUN ENTER
```

The display should appear:

```
THE DISTANCE IS 3ØØØ MILES.
```

Look carefully at program line 5Ø. There's no unused space between the S in IS, the D, and the M in MILES. But in the printout on the display, there is a space between IS and 3ØØØ, and another space between 3ØØØ and MILES. How come?

Reason: A semicolon automatically inserts one space between the two items it is separating. As you do more programming, this point will become important.

WHEW!

Well, we have already covered more than enough commands, statements and math operators to solve myriads of problems.

Math operators? — they're the =, +, —, * and / symbols we talked about earlier.

Now let's spend some time actually writing programs to solve problems. There is no better way to learn than by doing, and **everything** covered so far is fundamental to our success in later Chapters. So don't jump over these exercises — it's the best way to get you into the thick of programming. You'll find sample answers in Part B, along with further comments.

EXERCISE 3-2: Write a program which will find the time required to travel by jet plane from San Diego to Boston, if the distance is 3ØØØ miles and the plane travels at 5ØØ MPH.

EXERCISE 3-3: If the circumference of a circle is found by multiplying its diameter times π, (3.14) write a program which will find the circumference of a circle with a diameter of 35 feet.

EXERCISE 3-4: If the area of a circle is found by multiplying π times the square of its radius, write a program to find the area of a circle with a radius of 5 inches.

EXERCISE 3-5: Your checkbook balance was $225. You've written three checks (for $17, $35 and $225) and made two deposits ($4Ø and $2ØØ). Write a program to adjust your old balance based on checks written and deposits made, and print out your new balance.

Learned in Chapter 3

Statements	Math Operators	Miscellaneous
LET	=	,
	+	;
	—	A-Z variables
	*	
	/	

Remember, you can use any of the 26 letters, not just D, R and T (they were just convenient for our problem).

Chapter 4

Are There More Stars or Grains of Sand?

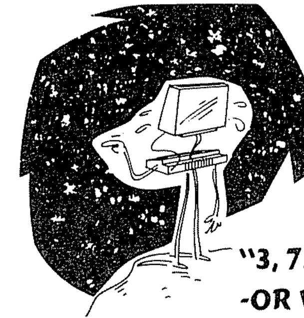
In this mathematical world we are blessed with very large and very small numbers. Millions of these and billionths of those. To cope with all this, your Computer uses "exponential notation", or "standard scientific notation" when the number sizes start to get out of hand. The number 5 million (5,ØØØ,ØØØ), for example, can be written "5E+Ø6". This means, "the number 5 followed by six zeros."

If an answer comes out "5E-Ø6", that means we must shift the decimal point, which is after the 5, six places to the **left**, inserting zeroes as necessary. Technically, it means 5×10^{-6}, or 5 millionths, (.ØØØ,ØØ5). It's really pretty simple once you get the hang of it, and a lot easier to keep track of numbers without losing the decimal point. Since the Computer insists on using it with very large and very small numbers, we can just as well get in the good habit, too.

Type NEW before performing the following exercises.

EXERCISE 4-1: If one million cars drove ten thousand miles in a certain year, how many miles did they drive altogether that year? Write and run a simple program which will give the answer.

EXERCISE 4-2: Changes lines 2Ø and 3Ø in the Car Miles Solution program (from Exercise 4-1) to express the numbers written there in exponential notation, or SSN (Standard Scientific Notation). Then RUN it.



Or technically, $5*10^6$, which is 5 times ten to the sixth power:

5*1Ø*1Ø*1Ø*1Ø*1Ø*1Ø

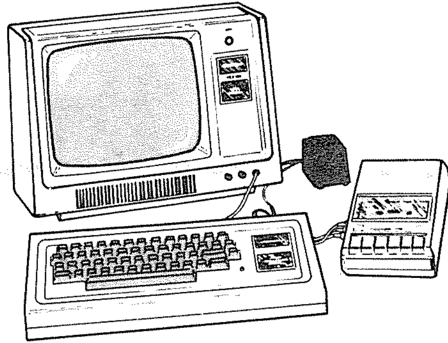
Now you can see the value of scientific notation!

In our BASIC, that's 5/1Ø/1Ø/1Ø/1Ø/1Ø/1Ø

Didn't forget the **ENTER** did you? Up till now we've been reminding you that you have to enter each line or command — but from now on, we'll assume you've got that little routine matter down pat.

Learned in Chapter 4

Miscellaneous
E–notation



(E stands for "exponent" and in our case it refers to the exponent of 1∅ — ie. the number of zeros to the right or left of the main number.)

Chapter 5

Using ( ) and the Order of Operations

Parentheses play an important role in computer programming, just as in ordinary math. They are used here in the same general way, but there are important exceptions.

1. In BASIC, parentheses can enclose operations to be performed. Those operations which are within parentheses are performed before those not in parentheses.
2. Operations buried deepest within parentheses (that is, parentheses inside parentheses) are performed first.
3. When there is a "tie" as to which operations the Computer should perform first after it has removed all parentheses, it works its way along the program line from left to right doing the multiplication and division. It then starts at the left again and performs the addition and subtraction.

NOTE: INT. RND and ABS functions are performed before multiplication and division. (We haven't talked about these yet, but just to be complete . . .)

4. A. problem listed as (X) (Y) will **NOT** tell the Computer to multiply. X * Y is for multiplication.

Example: To convert temperature in Fahrenheit to Celsius (Centigrade), the following relationship is used:

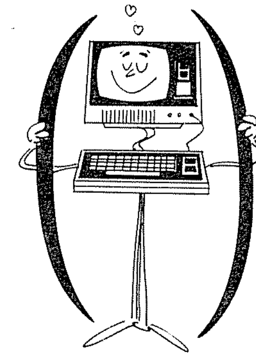
The Fahrenheit temperature equals 32 degrees plus nine-fifths of the Celsius temperature.

Or, maybe you're more used to the simple formula —

$$F^\circ = \frac{9}{5} \text{ X } C^\circ + 32$$

Assume we have a Celsius temperature of 25°. Type in this program and RUN it.

```
10 REM * CELSIUS TO FAHRENHEIT CONVERSION *

20 C = 25

30 F = (9/5)*C + 32

40 PRINT C; "DEGREES CELSIUS =" ;F;"DEGREES FAHRENHEIT."
```



"FRIENDS."

If you want to be sure your problems are calculated correctly, use ( ) around operations you want performed first.

Recall the old memory aid, "My Dear Aunt Sally"? In math you are supposed to do Multiplication and Division first (from left to right), then come back for Addition and Subtraction (left to right). The TRS-80 uses the same sequence.

Sample Run:

```
25 DEGREES CELSIUS = 77 DEGREES FAHRENHEIT.
```

First notice that line 4Ø consists of a PRINT statement followed by four separate expressions — two variables and two groups of words in quotes called "literals" or "strings".

Remember what the semi-colons are for?

Next, note how the parentheses are placed in line 3Ø. With the 9/5 secure inside, we can multiply its quotient times C, then add 32.

Now, remove the parentheses in line 3Ø and RUN again. The answer comes out the same. Why?

1. On the first pass, the Computer started by solving all problems within parentheses, in this case just one (9/5). It came up with (but did not print) 1.8. It then multiplied the 1.8 times the value of C and added 32.
2. On our next try, without the parentheses, the Computer simply moved from left to right performing first the division problem (9 divided by 5), then the multiplication problem (1.8 times C), then the addition problem (adding 32). The parentheses really made no difference in our first example.

Next, change +32 to 32+ and move it to the front of the equation in line 3Ø. Run it again, without parentheses.

Did it make a difference in the answer? Why not?
Answer: Execution proceeds from left to right, multiplication and division first, then returns and performs addition and subtraction. This is why the 32 was **not** added to the 9 before being divided by 5. *Very important!* If they had been added, we would of course have gotten the wrong answer.

EXERCISE 5-1: Write and run a program which converts 65° Fahrenheit to Celsius. The rule tells us that "Celsius temperature is equal to five-ninths times what's left after 32° is subtracted from the Fahrenheit temperature."

$$C^\circ = (F^\circ - 32) \times \left[\frac{5}{9}\right]$$

EXERCISE 5-2: Remove the first set of parentheses in the #5-1 answer and run again.

EXERCISE 5-3: Replace the first set of parentheses in program line 3∅ and remove the second pair of parentheses, then RUN. Note how the answer comes out — correctly!

EXERCISE 5-4: Insert brackets in the following equation to make it correct. Write a program to check it out on the TRS-80.

3∅ - 9 - 8 - 7 - 6 = 28

Learned in Chapter 5

Miscellaneous
( )
Order of Operations

Notes:

Chapter 6

IF you liked Chapters 1 through 5, THEN you're going to love the rest of this book!

Because we're really just getting into the good stuff. Like IF-THEN and GOTO statements that let your Computer make decisions and take . . . er, executive action. But first, a few more operators . . .

Relational operators allow the Computer to compare one value with another. There are only three:

1. Equals, using the symbol =
   (How'd you guess?)
2. Is greater than, using the symbol >
3. Is less than, using the symbol <

Combining these three, we come up with three more operators:

4. Is not equal to, using the symbol < >
5. Is less than or equal to, using the symbol <=
6. Is greater than or equal to, using the symbol >=

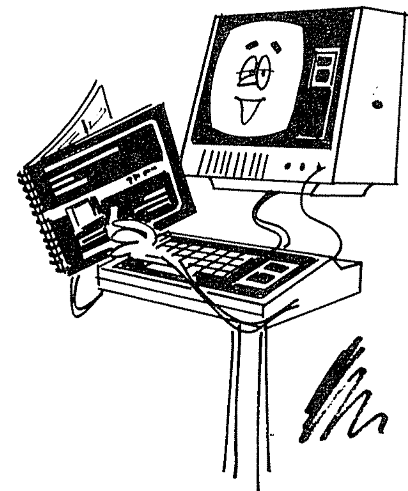
By adding these six relational operators to the four **math** operators we already know, plus new *STATEMENTs*, called IF-THEN, & GOTO, we create a powerful system of comparing and calculating that becomes the central core of everything else that follows.

The IF-THEN statement, combined with the six relational operators above, gives us the **action** part of a system of logic. Enter and RUN this program:

```
10 A = 5

20 IF A = 5 THEN 50

30 PRINT "A DOES NOT EQUAL 5."

40 END

50 PRINT " A EQUALS 5."
```

The screen should display:

```
A EQUALS 5.
```



"THEY GOTTA MAKE IT A MOVIE!"

Example: A< B means A is less than B. To help you distinguish between < and >, just remember that the smaller part of the < symbol points to the smaller of the two quantities being compared.

Now let's examine the program line by line.

Line 1Ø establishes the fact that A has a value of 5.

Line 2Ø is an IF-THEN statement which directs the Computer to **go to line 5Ø IF the value of A is exactly 5**, *skipping over whatever might be inbetween lines 2Ø and 5Ø*. Since A does equal 5, the Computer jumps to line 5Ø and does as it says, printing A EQUALS 5. Line 3Ø and 4Ø are not used at all in this case.

Now, change line 1Ø to read:

```
1Ø A = 6
```

and RUN

The run should say:

```
A DOES NOT EQUAL 5.
```

Taking it a line at a time:

Line 1Ø establishes the value of A to be 6.

Line 2Ø tests the value of A. If A equals 5, THEN the Computer is directed to go to line 5Ø. But "the test fails", that is, A does **NOT** equal 5, so the Computer proceeds as usual to the next line, line 3Ø.

Line 3Ø directs the Computer to print the fact that A DOES NOT EQUAL 5. It does not tell us what the **value** of A is, only that it does not equal 5. The Computer then proceeds on to the next line.

Line 4Ø ENDs the program's execution. Without this statement separating lines 3Ø and 5Ø, the Computer would charge right on to line 5Ø and print its contents, which obviously are in conflict with the contents of line 3Ø. This is an example of using an IF-THEN statement with only the most fundamental relational operator, the equals sign.

Now let's see if you can accomplish the same thing by using the "does not equal" sign:

EXERCISE 6-1: Rewrite the resident program using a "does not equal" sign in line 2Ø instead of the equals sign, changing other lines as necessary, so the same results are achieved with your program as with the one in the Example.

EXERCISE 6-2: Change line 1Ø to give A the value of 6. Leave the other four lines from #6-1 as shown. Add more program lines as necessary so the program will tell us whether A is larger or smaller than 5 and RUN.

EXERCISE 6-3: Change the value of A in line 1Ø at least three more times, running after each change to ensure that your new program works correctly.

No sample answers are given since you are choosing your own values of A. It will be obvious whether or not you are getting the right answer.

The IF-THEN statement is what is known as a *CONDITIONAL branching* statement. The program will "branch" to another part of the program **on the condition that** it passes the test it contains. If it fails the test, the program simply continues to the next line.

A statement called GOTO is known as an *UNCONDITIONAL branching* statement. If we were to replace lines 4Ø and 8Ø with GOTO 99, and add line 99:

```
99 END
```

. . . whenever the Computer hit line 4Ø or 8Ø it would **unconditionally** follow orders and go to 99, ENDing the run. While your Radio Shack Computer is rather broad-minded when it comes to accepting these various BASIC dialects, many computers are not. For practice, change lines 4Ø, 8Ø and 99 as discussed above and

Did the program work OK as changed? Did you try it with several values of A? Be sure you do so! We will find many uses for the GOTO statement in the future.

Note: **Do Not** leave a space between GO and TO (some forms of BASIC use two words – Radio Shack's BASIC wants only one word).

Learned in Chapter 6

Statements	Relational Operators	Miscellaneous
IF-THEN	=	Conditional branching
GOTO	>	Unconditional branching
	<	
	<>	
	<=	
	>=	

Chapter 7

It Also Talks and Listens

Begin this lesson by typing in the sample answer program to Exercise #6-2:

By now you have probably gotten tired of having to retype line 1Ø over each time you wish to change the value of A. The *INPUT* statement is a simple, faster and more convenient way to accomplish the same thing. It's a biggie, so don't miss any points.

Add the following lines to the *resident* program:

```
5 PRINT " THE VALUE I WISH TO GIVE A IS"

1Ø INPUT A
```

Now RUN

The Computer should print:

```
THE VALUE I WISH TO GIVE A IS

?_
```

See the question mark on the screen. It means, "It's your turn — and I'm waiting . . ."

Enter a number and see what happens. It should be identical to what happened when you typed in the same number earlier by **changing** line 1Ø. Run the program several more times to get the feel of the INPUT statement.

Pretty powerful, isn't it?

Let's add a touch of class to the INPUT process by retyping line 5 as follows:

```
5 PRINT " THE VALUE I WISH TO GIVE A IS";
```

Look at that line very carefully. Do you see how it differs from the earlier line 5??? It **is** different . . . . A semicolon has been added at the **end** of the line.

Resident — remember, that's the program that is now residing in the Computer.

Think back a bit now. We used semicolons before in PRINT statements, but only in the middle to hook several of them together so they would print close together on the same line. In this case, we put a semicolon at the **end**, so the **question mark** from the **next** line wil print on the **same** line, rather than down there by itself. After changing line 5 as above, RUN it. It should read:

```
THE VALUE I WISH TO GIVE A IS?_
```

Please note that you cannot use a semicolon indiscriminately at the end of a PRINT statement. It is only meant to hook two lines together, both of which have printing to be done. The INPUT line prints the question mark. We shall see later where two long lines starting with PRINT can be connected together by the trailing semicolon so as to print on the same line.

Your Radio Shack TRS-80 *Interpreter* is, as has been mentioned, able to speak "The King's Basic" as well as a variety of dialects. The first of the many "short-cut" dialects we will be exploring throughout these lessons involves combining PRINT and INPUT into one statement. Change line 5 to read:

```
5 INPUT "THE VALUE I WISH TO GIVE A IS";A
```

then delete line 1Ø by typing

```
1Ø
```

then RUN.

The results come out exactly the same, don't they? Here is what you have changed:

1. PRINT to INPUT
2. Both statements on the same line
3. Eliminated the extra line

In the long programs which you will be writing, running and converting, this shortcut will be valuable.

Up to now, all our programs have been strictly one-shot affairs. You type RUN, the Computer executes the program, prints the results (if any) and comes back with a READY. To repeat the program, you have to type in RUN again. Can you think of another way to get the Computer to execute a program two or more times?

Interpreter — is the internal circuit that allows you to "talk" to the TRS-80 in English (BASIC) and it can talk to you.

Sometimes the word **dialect** is used when talking about the different forms of a computer language. Just as with dialects in "human" languages, there can be slight differences in word uses, etc. in BASIC. (Radio Shack's BASIC is totally compatible with the Dartmouth BASIC — the original BASIC. But we do have some handy short cuts, so we might call them a "dialect".) We'll sometimes refer to this as a shortcut and sometimes as a dialect.

No — don't enlarge the program by repeating its steps over and over again — that's not very creative!

We'll answer that question by upgrading our Celsius-to-Fahrenheit conversion program (Chapter 5). If you think GOTO is a powerful statement in everyday life, wait till you see what it does for a computer program!

Type NEW and the following:

```
10 REM * IMPROVED CELSIUS TO FAHRENHEIT CONVERSION PROGRAM *

20 INPUT "WHAT IS THE TEMPERATURE IN DEGREES CELSIUS";C

30 F = (9/5)*C + 32

40 PRINT C; "DEGREES CELSIUS = ";F; "DEGREES FAHRENHEIT."

50 GOTO 20
```

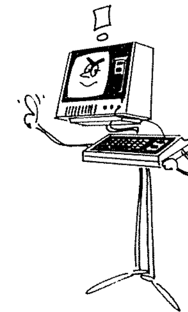
and RUN.

The Computer will keep on asking for more until you get tired or the power goes off (or some other event beyond its control). This is the kind of thing a Computer is best at — doing something over and over again. Modify some of the other programs to make them self-repeating. You'll find they're much more useful that way.

These have been 7 long and "meaty" lessons, so go back and review them all again, repeating those assignments where you feel weak. We are moving out into progressively deeper water, and it is complete mastery of these **fundamentals** that is your life preserver.

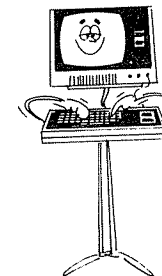
Learned in Chapter 7

Statements	Miscellaneous
INPUT and INPUT with built-in PRINT	; Trailing semicolon



"GO TO!"

You'll have to hit [BREAK] to get out of the program loop.



"I CAN DO THIS ALL DAY"

Notes:

Chapter 8

Two Easy Features

The Calculator Mode

Before continuing our exploration of the nooks and crannies of our Computer — acting as a **computer**, we should be aware that it also works well as a **calculator**. If you **omit** the line number before certain commands, the Computer will execute them, print the answer on the screen, then erase the command you entered. What's more, it will work as a calculator even when a computer program is loaded, **without disturbing that program**. All you need, to be in the calculator mode, is the prompt >.

Example: How much is 3 times 4? Type in

```
PRINT 3 * 4
```

. . . the answer comes back

```
12
```

Example: How much is 345 divided by 123?

```
PRINT 345/123
```

. . . the answer is

```
2.80488
```

Spend a few minutes making up routine arithmetic problems of your own, using the calculator mode to solve them. Any arithmetic expression you might use in a program can also be evaluated in the calculator mode. This includes parentheses and chain calculations like A*B*C.

Try the following problem:

```
PRINT (2/3)*(3/2)
```

The answer comes back:

```
1.0000009
```

What? A number times its reciprocal is supposed to equal 1 exactly. So what gives? You have discovered the Computer's limit of accuracy. Just like a calculator (or a person), a computer can never be perfectly accurate all the time. For short arithmetic expressions, the TRS-80 is accurate to the fifth or sixth decimal place. In longer, more complex expressions, a minute error in the sixth place can be magnified to where it becomes significant. All programmers have to cope with this kind of built-in error. We'll discuss one way of handling it in a later chapter.

Calculator Mode for Troubleshooting

Suppose a program isn't giving you the answers you expect. How can you troubleshoot the program? One way is to ask the Computer to tell you what it knows about the variables used in the resident program.

Example: PRINT X. The Computer will tell you what the present value of X is.

Keep this handy tip in mind as you get into more complex programs.

Another thought: *Something* is stored in every memory cell (even if *YOU* have not put anything there). Enter and RUN this instruction in the calculator mode:

```
PRINT A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S,T,U,V,W,X,Y,Z
```

The answers depend on the values last given those variables — even from much earlier programs. If you turned off the Computer since last using some of the variables, the numbers stored in those locations will be completely arbitrary and meaningless.

The Memory Command

Since the programs you write do occupy space in the Computer's memory, and program size is limited to how much memory you have purchased, it may be important to know how much memory you are using for a given program. That's what the Memory Command is for.

This manual is meant to be for the computer operator and programmer, so we are studiously avoiding computer electronics theory — when possible. Owners interested in that phase of this interesting subject are referred to their local Radio Shack store for books which specifically address it.

The least amount of memory available in the TRS-80 is 4K. This means there are about 4,ØØØ different memory locations to store and process your programs. (Actually, 4Ø96.) (If you have 8K of memory, the number is 8192; for 16K it is 16384.)

To get some idea what this means, type:

```
NEW

PRINT MEM
```

. . . and the answer will be

```
3583
```

With no program loaded, there are 3583 memory locations available for use. The difference in memory space between 3583 and 4∅96 is set aside for processing programs and overall management and "monitoring" of what the Computer is doing.

NOTE: If you don't get this answer (for example a very large number or one with a — in it), turn the Computer's POWER off for at least 1∅ seconds and then turn it on again. Try PRINT MEM once more. [If your TRS-80 has more RAM, you can expect a larger number, as follows:
8K RAM: 7679
16K RAM: 15871.]

Type in this simple program:

```
10 A = 25
```

then measure the memory remaining by typing

```
PRINT MEM
```

. . . the answer will be

```
3573
```

The program you entered took 3583 — 3573 = 1∅ *bytes* of space. Here is how you can account for it:

Byte — is the basic unit of storage for most computers; normally it is considered as a string of eight binary digits (bits). Thus a byte = 8 bits.

1. Each line number and the space following it (regardless of how small or large that line **number** is) occupies 3 memory cells. The "carriage return" at the end of the line takes 1 more byte, even though it does not print on the screen. The memory "overhead" for each line, short or long, is 4 bytes.

2. Each letter, number and space takes 1 byte.

In the above program 4 + 6 = 1∅ bytes.

Enter this additional line, leaving in line 1∅, and calculate the amount of space remaining in memory. Then check it with the PRINT MEM command.

```
20 PRINT "THIS EXAMPLE IS TO MEASURE MEMORY USAGE."
```

How much space is left in memory???? ______________________.

Answer: Line 1∅ took up 1∅ bytes. Line 2∅ takes 4 bytes for "overhead" + 48 characters = 52 bytes. 52 + 1∅ = 62 bytes. 3583 — 62 = 3521 bytes. Type

```
PRINT MEM
```

. . . to see if you agree.

Obviously, the short learning programs we have been writing so far are not taking a lot of memory space. This changes quickly, however, as we move to more sophisticated programming. Make a habit of typing PRINT MEM when completing a program to develop a sense of its size and memory requirements.

The third and final error message is

```
SORRY
```

It means "Sorry — you have run out of memory locations and must either cut down the program size or purchase additional memory." With some practice you will be able to predict how much memory a given program will need. All lessons and programs furnished with Radio Shack's LEVEL I system will run in the "4K" of memory you have available.

Remember the others?
WHAT? — "I don't understand *WHAT* you want."
HOW? — "I understand what you are telling me, but I don't know *HOW* to do it."

Learned in Chapter 8

Commands	Miscellaneous
PRINT MEM	Calculator Mode
	Memory
	Byte
	SORRY

Chapter 9

Using Cassette Tape

We will soon write and run long and powerful programs. It becomes tedious to type them in accurately just once, let alone each time we want to use them. Impressing your friends with this new super-whazzoo Computer is somewhat more difficult if they sit watching old TV reruns of Star-Trek while you take an hour or so to type in a program. There has to be a better way.

Back at the beginning of this Manual we gave a procedure for connecting and using the Tape Recorder *(remember . . . that was for all the impatient ones!)*. Let's take out time with this Chapter to be sure we've covered EVERYONE.

The TRS-80 has a built-in "Cassette Tape Interface" which allows you to record and store any program on high quality cassette tape. A full "4K" of memory can be *dumped* onto tape, or *loaded* from tape, in about 3 minutes. Most programs are shorter and take even less time. That isn't even enough time to get through the deodorant ads. Besides building up your own tape library of computer programs, you can exchange favorite programs with other TRS-80 owners by exchanging tapes.

Dumped and loaded are everyday terms used by computer people for storing and "playing back" computer programs (onto tape and from tape).

Recording

Only a little practice is required. Follow the yellow brick road:

1. Locate the Recorder, Interconnecting Cable and Radio Shack Computer Recording Tape cassette.
2. Connect the short cable between the **TAPE** jack on the back of the TRS-80 and your Cassette Tape Recorder:
   A. The small gray plug goes into the REM jack on the Recorder.
   B. The large gray plug goes into the AUX jack.
   C. The black plug goes into the EAR jack.
3. Plug the Recorder into the wall outlet (or install batteries).
4. Type any program into your Computer, preferably one that is at least several lines long. RUN it to be sure it is entered correctly.
5. Load the cassette tape and press the PLAY and RECORD buttons **at the same time** until they lock.
6. "Dump" the program onto tape by typing the command:

NOTE: A "dummy plug" is provided with Radio Shack's CTR-41 — **you must plug this into the MIC jack on the CTR-41** (this prevents sound pick-up from the built-in Mic when "dumping" your TRS-80 programs onto tape).

```
CSAVE
```

CSAVE stands for "Save on Cassette."

The motor on the Recorder will start and you'll be recording the Computer's program onto tape.

Watch the Video screen. When

```
READY

>-
```

returns and the motor stops, your program is recorded on tape. It is also still in the Computer's memory. It has only been "copied" out.

7. Disconnect the small plug from the Recorder's REM jack and Rewind the tape. Disconnect the black plug from the EAR jack and Play the tape so you hear what digital data sounds like. Sounds terrible, doesn't it? *You were expecting maybe Lawrence Welk?*

Loading

Reversing the process and loading (copying) the program from tape into the Computer is just as easy.

1. Be sure the tape is fully rewound and the plugs are all in place.
2. Push down the PLAY button until it locks. Set the Volume control to about 7-8.
3. Type NEW (to clear out any existing program).
4. Type the command

```
CLOAD
```

The Tape Recorder's motor will start and data will flow from the tape into the Computer's memory at the rate of about 1200 bytes per minute.

As soon as the Computer senses the data, it will flash a * on the screen; then as it accepts each line of data, a second * will flash on and off.

Watch the Video Display. The program is entered when

```
READY

>-
```

returns and the recorder motor stops.

4. RUN the program to see that the data transfer was successful. In the rare event that it was not, repeat the above steps, being sure that all cables are properly connected, **the Volume is set to** 7-8 and the tape recorder heads are clean. (Listen to the tape to be sure there was a program on it.)

> IMPORTANT: Too little volume will cause a bad "data dump": too much volume may result in distortion in the Tape Recorder and also goof-up the "dump".

> CLOAD stands for "Load from Cassette"

> The * * display (second one flashing) is a fool-proof indication that data is being loaded onto the Computer.

> NOTE: If the recorder does not stop, reach around the back of the Computer, open the door at the left rear and press the Reset button inside. This will take the Computer out of the CLOAD or CSAVE mode and return control to the keyboard.

Miscellaneous Tape Palaver

To minimize the chance of hitting a "soft spot" on a tape, where the oxide may be thin or have flaked off, experienced operators routinely do a "double dump" when copying from

computer to tape. This simply means copying the program twice on the same tape — one recording right after the other. On long dumps, one is made in one direction and the other one in the other direction. For extra safety, especially important programs are recorded on more than one tape. Failures are rare, and your own experience should be your guide.

You may have noticed that specially wound Radio Shack Computer Tape has no plastic leader on the ends. This is because when you begin "dumping" data from memory onto tape there must be real live tape there to record it.

Normal audio tape has lead-ins on both ends (typically blue non-magnetic mylar material) — **you can not record on the leader portion of tapes.** Advance the tape past the leader before recording a program.

Radio Shack's Computer Tape is of high uniform quality, selected especially for its low "drop out" characteristics. If one little bit of data is lost the entire program can be lost. This Tape is wound in shorter than usual lengths, with the C-10 being standard. It will record 5 minutes in each direction — far more than enough for the majority of programs. Your entire 4K of memory can be recorded on one side of the C-10, with the other side quickly available for another program without need for a long rewind.

Experienced "computerists" have found from experience that it is better to use a separate cassette (or at least a separate side) for each program rather than try to search through long tapes for a desired program. Since computer data on tape is not readable by the human ear, separate cassettes solve the problem. Computer Tape, tape racks and other recording accessories are available at your local Radio Shack store.

If you record programs on long, standard audio cassettes, use the Tape Recorder's Counter to aid you in locating programs.

When you are not using the Recorder for loading or recording, do not leave RECORD or PLAY keys down (press STOP).

Do not expose recorded tapes to magnetic fields. Avoid placing your tapes near the Power Supply.

Do not attempt to re-record on a pre-recorded Computer data tape. Even though the new recording process erases the old recording, just enough information may be left to confuse the new recording. If you want to use the same tape a second or third time, use a high-quality bulk tape eraser to be sure you erase old data.

If you want to save a taped program permanently, break off the Erase Protect tab on the Cassette (see the Tape Recorder's Manual). When the tab(s) has been broken off, you can not press down the RECORD key on your Recorder (this will keep you from accidentally erasing that tape).

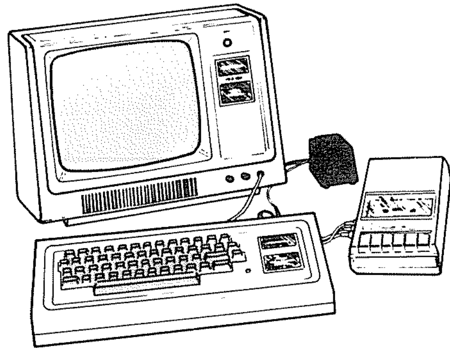
Ground Loops

With some recorders, if you leave the Earphone **and** Aux jack connected at the same time, when you make a recording, you'll end up with a hum added to the program (you can hear it between double-dumps). This is caused by a *ground loop* in the Recorder and cables. To avoid ground loop problems, keep only the Earphone or the Aux plug connected **BUT NOT BOTH**.

Ground loop is an electronic term which means there are two separate ground connections, each being slightly different — the result typically is hum (and you don't want or need that).

Learned in Chapter 9

Miscellaneous
CLOAD
CSAVE
"loading"
"dumping"
**(program loading indicator)



Chapter 10

From > to FOR-NEXT . . . or *SMART* Loops

A major difference between the computer and a calculator is the computer's ability to do the same thing over and over an outrageous number of times, faster than a speeding bullet (to coin a phrase)! This one capability more than any other, separates the two.

The FOR-NEXT loop is of such overwhelming importance in putting our Computer to work, that few of the programming areas we will explore from this point on will exclude it. Its simplicity and variations are the heart of its effectiveness, but its power is truly staggering.

Type in the following program, and RUN :

```
10 PRINT "HELP --- MY COMPUTER HAS GONE BERSERK!"

2Ø GOTO 1Ø
```

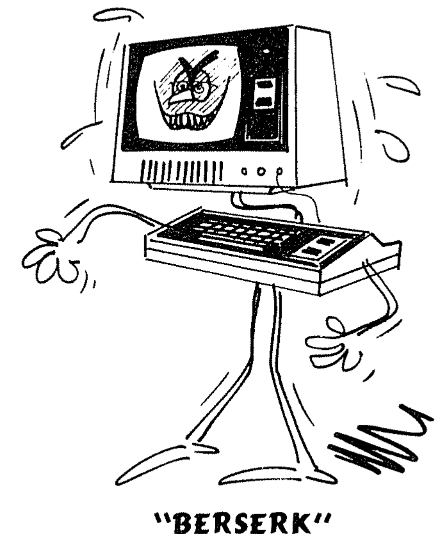
You have noticed by now that the Computer is continuously writing the line HELP --- MY COMPUTER HAS GONE BERSERK! It will continue to do so indefinitely until you tell it to stop. When you have seen enough, hit the [BREAK] key.

What we created is called an "endless loop". (Remember our earlier programs which kept coming back for more INPUT?) Line 2Ø is an unconditional GOTO statement which causes the Computer to cycle back and forth ("loop") between lines 1Ø and 2Ø forever if not halted. This idea has great potential if we can harness it.

Let's modify the program to read:

```
8 FOR N = 1 TO 5

1Ø PRINT "HELP --- MY COMPUTER HAS GONE BERSERK!"

2Ø NEXT N

3Ø PRINT "NO --- IT'S UNDER CONTROL."
```

and it.



"BERSERK"

The line HELP --- MY COMPUTER HAS GONE BERSERK! was printed 5 times, then NO --- IT'S UNDER CONTROL. The FOR-NEXT loop created in lines 8 and 2Ø caused the Computer to cycle through lines 8, 1Ø and 2Ø exactly 5 times, then continue through the rest of the program. Each time the Computer hit line 2Ø it saw "NEXT N." The word NEXT caused the value of N to be increased (or STEPped) by exactly 1, and the Computer unconditionally sent back to the FOR N = statement that began the loop. The NEXT statement is conditional on N being less than 5, because line 8 says FOR N = 1 **TO** 5. After the 5th pass through the loop, the built-in test fails, the loop is broken and the program execution moves on. The FOR-NEXT statement harnessed the endless loop!

The STEP function

There are times when it is desirable to increment the FOR-NEXT loop by some value other than one. The STEP function allows that. Change line 8 to read

```
8 FOR N = 1 TO 5 STEP 2
```

. . . and RUN.

Line 1Ø was printed only 3 times (when N=1, N=3 and N=5). On the first pass through the program, when NEXT N was hit, it incremented (or STEPped) the value of N by 2 instead of 1. On the second pass through the loop N equalled 3. On the third pass through N equalled 5.

FOR-NEXT loops can be stepped by any whole number, even negative numbers. Why one would want to step with negative numbers might seem rather vague at this time, but that too will be understood with time. In the meantime, change the following line

```
8 FOR N = 5 TO 1 STEP -1
```

. . . and RUN.

Five passes through the loop stepping **down** from 5 to 1 is exactly the same as stepping **up** from 1 to 5. Line 1Ø still got printed 5 times.

Modifying the FOR-NEXT loop

Suppose we wanted to print both lines 1Ø and 3Ø five times, alternating between them. How would you change the program to accomplish it? Go ahead and make the change.

HINT: If you can't figure it out, try moving the NEXT N line to some other position.

Right — you moved line 2∅ to line 4∅ and the screen reads:

```
HELP --- MY COMPUTER HAS GONE BERSERK!

NO --- IT'S UNDER CONTROL.

HELP --- MY COMPUTER HAS GONE BERSERK!

NO --- IT.S UNDER CONTROL
```

. . . etc. — 3 more times.

How would you modify the program so line 1∅ is printed 5 times, then line 3∅ is printed 3 times? Make the changes and RUN.

The new program might read:

```
8 FOR N = 1 TO 5

1Ø PRINT "HELP --- MY COMPUTER HAS GONE BERSERK!"

2Ø NEXT N

25 FOR M = 1 TO 3

3Ø PRINT "NO --- IT'S UNDER CONTROL."

4Ø NEXT M
```

We now have a program with **two** controlled loops, sometimes called *DO* loops. The first do-loop *DOES* something five times; the second one does something three times. We used the letter N for the first loop and M for the second, but any letters can be used. In fact, since the two loops are totally separate we could have used the letter N for both of them — not an uncommon practice in large programs where most of the letters are needed as variables.

RUN the program, being sure you understand the fundamental principles and the variations we have introduced.

From > to Incrementing

There is nothing magic about the FOR-NEXT loop, in fact, you may have already thought of another (longer) way to accomplish the same thing by using features we learned earlier. Stop now, and see if you can figure out a way to construct a workable do-loop substituting something else in place of FOR and NEXT.

Answer:

```
8 N = 1

10 PRINT "HELP --- MY COMPUTER HAS GONE BERSERK!"

15 N = N + 1

20 IF N<6 THEN 10

30 PRINT "NO --- IT'S UNDER CONTROL."
```

We say that line 8 *initializes* the value of N, giving it an initial or beginning value of 1. Before initializing to the value we want, N could have been any number left over from a previous program.

Line 15 then *increments* it by 1, making N one more than whatever it was before. Line 10 uses one of our relational operators,<, to see that the new value of N is within the bounds we have established. If not, the test fails and the program continues.

Initializes — initially, or at the beginning, sets the value of one of our variables (or starts a program back at the beginning).

Increments — steps (increases or decreases values in specific steps: by 1's, 3's, 5's, or whatever).

Note that in this system of *incrementing* and testing we do not send the program back to line 8 as was the case with FOR-NEXT. What would happen if we did?

Answer: We would keep re-initializing the value of N to equal 1, and would again form an endless loop.

The opposite of *incrementing* is *decrementing*. Change the program so line 15 reads

```
15 N = N - 1
```

. . . then make other changes as needed to make the program work.

Answer: The changed lines read:

```
8 N = 6

15 N = N - 1

2Ø IF N>1 THEN 1Ø
```

To decrement is to make smaller.

Putting FOR-NEXT to work

It isn't very exciting just seeing or doing the same thing over and over, so there has to be a more noble purpose for the FOR-NEXT loop. There are — many of them, and we will be learning new uses for a long, long time.

Let's suppose we want to print out a chart showing how the time it takes to fly from Boston to San Diego varies with the speed at which we fly. Remember, the formula is D = R*T. Let's print out the flight time required for each speed between 2ØØ mph and 1ØØØ mph, in increments of 2ØØ mph. The program might look like this:

```
1Ø REM * TIME VS RATE FLIGHT CHART *

2Ø CLS
```

```
30 D = 3000
40 PRINT "    B O S T O N  T O  S A N  D I E G O    "
50 PRINT
60 PRINT "RATE (MPH)", "TIME (HOURS)", "DISTANCE (MILES)"
70 PRINT
80   FOR R=200 TO 1000 STEP 100
90    T = D/R
100   PRINT R,T,D
110  NEXT R
```

Enter the program and RUN.

It is really solving the problem from Chapter 3 nine times in a row, for different values, and printing out the result. Your screen should look like this:

```
          B O S T O N  T O  S A N  D I E G O

RATE (MPH)          TIME (HOURS)      DISTANCE (MILES)

  200                 15                3000
  300                 10                3000
  400                 7.5               3000
  500                 6                 3000
  600                 5                 3000
  700                 4.28571           3000
  800                 3.75              3000
  900                 3.33333           3000
 1000                 3                 3000
```

How about that . . . ? Try doing that on the old slide rule or hand calculator!

Analyzing the Program

Look through the program and observe these many features before we do some exercises to change it:

1. The REM statement identifies the program for future use.
2. Line 2∅ uses the CLS (Clear Screen) statement to erase the screen so we have a nice clean place to write on. It allows us to write in a *top down* manner. Run the program later leaving out this line and see what is meant by the *scroll* mode. CLS is a very unfussy statement which you will want to use often just to make your printouts neat and impressive.
3. Line 3∅ *initializes* the value of D. D will remain at its initialized value.
4. Line 4∅ prints a chart heading which is indented and double spaced for appearance.
5. Lines 5∅ and 7∅ use blank PRINTs to insert spaces in the chart.
6. Line 6∅ prints the chart column headings, and uses *automatic zone spacing* to place those headings (the comma).
7. Line 8∅ establishes the FOR-NEXT loop complete with a STEP. It says — initialize the rate (R) at 2∅∅ mph, and make passes through the "do-loop" with values of R incremented by values of 1∅∅ mph until a final value of 1∅∅∅ mph is reached. Line 11∅ is the other half of the loop.
8. Line 9∅ contains the actual formula which calculates the answer.
9. Line 1∅∅ prints the three values. They are positioned under their headings by automatic zone spacing (the commas).
10. Lines 9∅ and 1∅∅ are indented from the rest of the program text. This is a simple programming technique highlighting a do-loop which makes reading and troubleshooting easier. You will see it used increasingly as we move on. **Try to adopt good programming practices like this** as you do the exercises. Indenting does take up a little memory space, and on long programs in their final form it is often omitted.

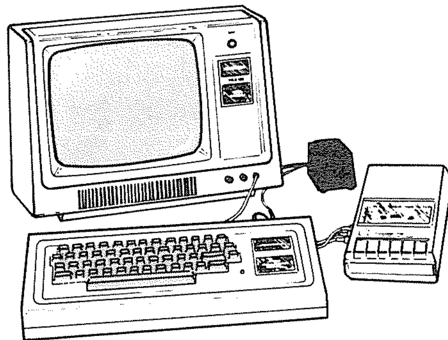
Take a deep breath and go back over any points you might have missed in this lesson. Copy the program onto your Computer Cassette Tape because we will use it in the next Chapter continuing our study of FOR-NEXT loops.

CLS in a program does the same thing as the [CLEAR] Key on the keyboard (but you can't use the [CLEAR] Key as part of a program).

Remember zone spacing? . . . The comma (,) in a PRINT statement automatically starts the printing in the next 16-space print zone.

Learned in Chapter 10

Commands	Statements	Miscellaneous
	FOR-NEXT	Increment
	CLS	Decrement
	STEP	Initialize
		[BREAK] key
		[CLEAR] Key
		"Top down" Display
		"Scroll" Display
		"Do-Loop"



Chapter 11

Son of FOR-NEXT

This is heady stuff. If you turned off the Computer between Chapters, load the program which you taped from Chapter 1Ø into the Computer.

Modify the program so the rate and time are calculated and printed for every 5Ø mph increment instead of the 1ØØ mph increment presently in the program. RUN.

____

____

____

Answer: `8Ø FOR R = 2ØØ TO 1ØØØ STEP 5Ø`

Trouble in the Old Corral

What a revolting development! The printout goes so fast we can't read it, and by the time it stops, the top part is cut off. *Aught'a known you can't trust these computers!*

Solutions For Sale

Several solutions are available:

1. Pressing nearly any key will stop program execution. Try RUNning a number of times, pressing different keys (and the space bar) during the run, to see what happens.

   RUN again, this time using only the ↑ (up-arrow), to freeze the display. Nifty — huh? Clean stop — clean restart. This is the key to use for temporary freezes.

2. If you want a classy display you can build a "pause" **into the program**. The screen will fill, halt a moment, and automatically go on if you don't interrupt the program.

As you can see, pressing a key not only stops execution but inserts its own letter or number. Messy!

There's another one you can try — but it's not a very useful one; press [BREAK] key. That's even messier than the first one. (To restart after a [BREAK], either enter RUN to start program all over again or CONT to continue execution at the "break-point.")

The Timing Loop

In order to learn about the *timer loop,* let's employ another sly trick. We're going to leave our "Flight time" program in the Computer, and put in a second program.

Start by typing

```
9 END
```

We are going to use the space in lines 1 through 8 to write and experiment with a second little program, and want it to END without plowing ahead into the "Flight" program.

The Egg Timer

It takes time to do everything. Even this foxy box takes time to do its thing, though you may be awed by its speed. Type this:

```
1 PRINT "DON'T GO AWAY"

2 FOR X = 1 TO 5ØØØ

3 NEXT X

5 PRINT "TIMER PROGRAM ENDED."
```

. . . and RUN

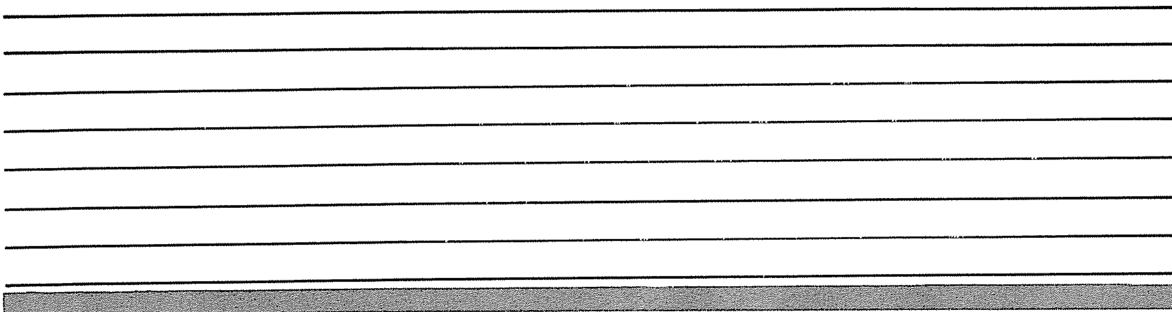
Remember back when we told you not to do this (number lines directly in sequence)? *Well . . . if we hadn't* followed that rule we wouldn't have this nice space to demonstrate the point

How long did it take? Well, it did take time, didn't it? About 1Ø seconds? The Computer can do approximately 5ØØ FOR-NEXT loops per second. That means, by specifying the number of loops, you can build in as long a time-delay as you wish.

Change the program to create a 3Ø-second delay. Time it against your watch or clock to see how accurate it is.

Answer: `2 FOR X = 1 TO 15ØØØ`

EXERCISE 11-1: Using the space in lines 1 through 8, design a program which asks you how many seconds delay you wish, allows you to enter a number, then executes the delay and reports back at the end that the delay is over, and how many seconds it took. A sample answer is in Part B.



How to Handle Long Program Listings

We've got two programs in the Computer now. Let's pull a LIST to look at them. My, my — they are so long we can't see the end. *Now what do we do?*

Again, more than one solution. The easiest way to see the rest of the listing is to use the ↑(up-arrow) key. Each time you press it (go ahead), the listing moves up one line. Pretty exciting, huh? Keep pushing it until you get a prompt (>).

As a matter of fact, you HAVE to keep pushing it, or do something else to get the prompt, since without the prompt it just isn't your turn.

The other solution is to use a slightly more sophisticated version of LIST. It's called LIST### (however you pronounce that!). Type:

```
LIST 5Ø
```

A little scrutiny immediately discloses that the Computer gave a listing starting with line 5Ø and either 1) filled the screenwith 16 lines, or 2) went from line 5Ø to the end of the program, whichever came first.

The ###'s represent the number of the line you want the LISTing to start from.

LIST### and ↑ (up-arrow) can be used to find any part of a very long program you wish.

Again, you must have a prompt in order to continue on and do anything else. Aside from using one (or both) of the above techniques to get to the end of the LIST and find a prompt, there is a quicker way, once you've found what you want in the list. Simply hit the **ENTER** key once or twice to get a prompt.

Is There No End to This Magic?

We now have 2 separate programs resident in the computer. We know how to run the first one — we just type RUN. To run the second one we have a foxy variation on RUN called

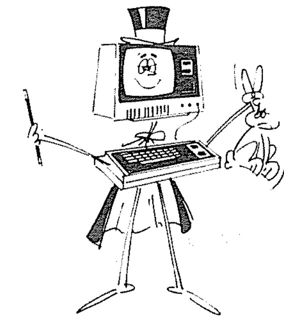
```
RUN ###
```

. . . and, as you might suspect, it is similar to LIST###. To RUN the program starting with line 1Ø, type

```
RUN 1Ø
```

. . . and that's just what happens.

Again, the ###'s represent the number of the line you want the RUN to start with.



"MAGICIAN"

Will wonders never cease? If you have 2Ø or 3Ø programs in the computer at the same time, you can RUN just the one you want, provided you know its starting line number. What's more, you can start any program in the middle (or elsewhere) for purposes of trouble-shooting — a matter we will become more involved in as our programs get longer and more complicated.

Meanwhile, Back at the Ranch

We got into this whole messy business trying to find a way to slow down our run on the flight times from Boston to San Diego. In the process we found out a lot more about the Computer and learned to build a timer loop. Now let's see if we can build a timer loop into our big program. First, let's erase the test program using lines 2, 3, 4, 5, 6, 7, & 9 by typing each of those numbers followed with **ENTER** .

One way to **stop** the fast parade of information in our chart is to put in a STOP. Type in

```
85 IF R = 6ØØ STOP
```

. . . and RUN.

We know R is going to increment to 6ØØ, and that's about half way through the chart, so 6ØØ is a good choice. See how the chart ran out to 55Ø mph then hit the stop at 6ØØ came racing down to line 85. Your screen should read the first part of the chart and

```
BREAK AT 85
```

This means the program is stopped, or broken at **line 85**. You can now gaze at the top half of the chart to your heart's content. To restart the program merely type

```
CONT
```

. . . and it will automatically pick up and print the rest of the chart, or until it hits another stop you may have placed.

At Last

Our ultimate plan is to build a timer into the program so as not to completely STOP execution, but merely delay it so we can study the display.

Type

```
85 IF R<> 600 THEN 90

87 FOR X = 1 TO 500

88 NEXT X
```

. . . and RUN

Hey! It really works! As long as R does **not** equal 600 the program skips over the delay loop in lines 87 and 88. when R **does** equal 600, the test "falls through" and lines 87 and 88 "play catch" 500 times, delaying the program's execution for about one second.

It's been a long and tortuous route with numerous scenic side trips, but we finally made it. Now that you have picked up so many smarts in these two lessons on FOR-NEXT, it's your turn to put them to work.

EXERCISE 11-2: Modify the resident program so that (MPH) appears **below** RATE, (HOURS) appears **below** TIME and (MILES) appears **below** DISTANCE . This one should be a breeze for you.

EXERCISE 11-3: Design, write and run a program which will calculate and print income at a yearly, monthly, weekly and daily rate, based on a 40-hour week, a 1/12th-year month, and a 52-week year. Do this for yearly incomes between $5,000 and $25,000 in $1,000 increments. Document your program with REM statements as necessary to explain the equations you create.

Some of our programs are becoming a little too long for us to leave space in the manual for you to write in your ideas. From now on, use the pad of Programming paper for working up your answers.

EXERCISE 11-4: Here's an old chestnut that the Computer really eats up: Design, write and run a program which tells how many days you have to work, starting at a penny a day, so if your salary doubles each day you know which day you earn at least a million dollars. Include columns which show each day number, its daily rate, and the total income to-date. Make the program stop after printing the first day your daily rate is a million dollars or more.

The "Brute Force" Method (Subtitled: Get a Bigger Hammer)

Much to the consternation of some teachers, a great value of the Computer is its ability to do the tedious work involved in the "cut and try", "hunt and peck" or other less respectable methods of finding an answer (or attempting to prove the correctness of a theory, theorem or principle). This method involves trying a mess of possible solutions to see if one fits, or find the closest one, or establish a trend. Beyond that, it can be a powerful learning tool by providing gobs of data in chart or graph form (later) which would simply take too long to generate by hand.

EXERCISE 11-5: You have a 1ØØØ foot roll of fencing wire and want to create a rectangular pasture.

Using all of the wire, determine what length and width dimensions will allow you to enclose the maximum number of square feet? Use the brute force method; let the Computer try different values for L and W and print out the Area fenced by each pair of L and W.

The formula for area is **Area = Length times Width**
or A = L * W

EXERCISE 11-6: EXTRA CREDIT PROBLEM FOR "ELECTRONICS TYPES"

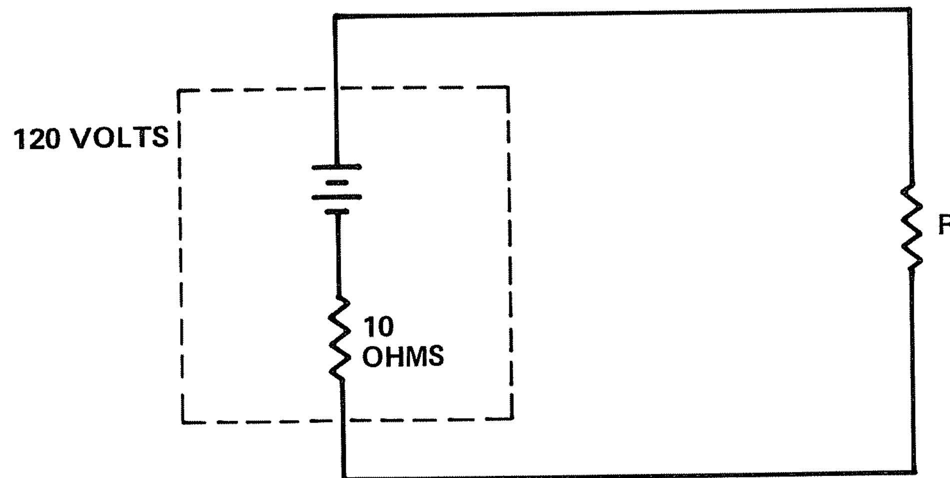
As a further example (more complex and tends to prove the point better) try this final (optional) assignment in this lesson. It involves a problem confronted by every electricity student who has studied **sources** (batteries, generators) and **loads** (lights, resistors). It is the MAXIMUM D.C. POWER TRANSFER THEOREM which states, "Maximum DC power is delivered to an electrical load when the resistance of that load is equal in value to the internal resistance of the source." And then the arguments begin . . . "Use a high resistance load because it will drop more voltage and accept more power." "No, use a low resistance load so it will draw more current and accept more power". "Use a load which is somewhere in between."

Don't necessarily shy away from this one if electricity doesn't happen to be your bag. Enough information is given to write the program, and the principle, the optimizing of a value, is applicable to many fields of endeavor and is little short of profound.

With the values given in the schematic, design, write and run a program which will try out values of load resistance ranging from 1 to 2Ø ohms, in 1 ohm increments, and print the answers to the following:

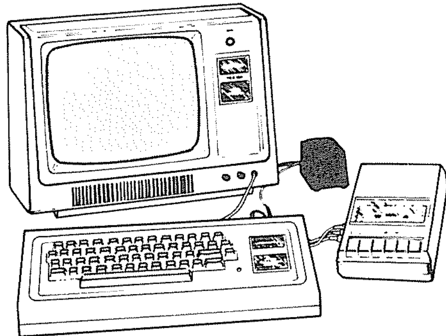
1. Value of Load Resistance (from 1 to 2Ø ohms)
2. Total circuit power (circuit current squared, times source voltage) I^2 * 1Ø
3. Power lost in source (circuit current squared, times source resistance) I^2 * 1Ø
4. Power delivered to load (circuit current squared, times load resistance) I^2 * R
   Note: Circuit current is found by dividing source voltage (12Ø volts) by total circuit resistance (load resistance + 1Ø ohms source resistance). Everything follows Ohms Law (V = I*R) and Watts Law (P = I*V)

GOOD LUCK!!!! Don't look at the answer until you've got it whipped.

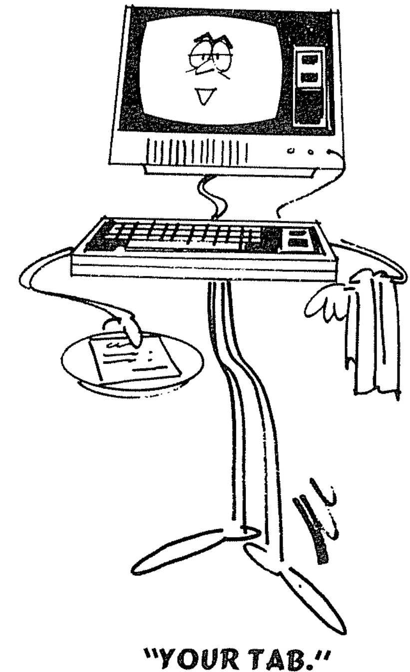


Learned in Chapter 11

Commands	Statements	Miscellaneous
LIST###	STOP	
RUN###		Timer Loop
CONT		↑ Up-Arrow
		"Brute force" or optimizing method



Chapter 12


"YOUR TAB."

From > to TAB

After those last lessons let's take an easy one.

We already know 3 ways to set up our output PRINT format.

We can:

1. Enclose what we want to say in quotes, inserting blank spaces as necessary.
2. Separate the objects of the PRINT statement with semicolons so as to print them tightly together on the same line.
3. Separate the objects of the PRINT statement with commas to print them on the same line in the four different print "zones."

A fourth way is to use the TAB function, which is similar to the TAB on a regular typewriter. It is especially useful when the output is columns of numbers with headings. Type in the following program and RUN:

```
10 PRINT TAB (5);"THE";TAB(20);"TOTAL";TAB(35);"SPENT"

20 PRINT TAB(5);"BUDGET";TAB(20);"YEAR'S";TAB(35);"THIS"

30 PRINT TAB(5);"CATEGORY";TAB(20);"BUDGET";TAB(35);"MONTH"
```

The RUN should appear:

```
THE            TOTAL          SPENT
BUDGET         YEAR'S         THIS
CATEGORY       BUDGET         MONTH
```

EXERCISE 12-1: Write a program using the three PRINT forms below:

```
1. PRINT"      ","      ","      "
2. PRINT"                                    "
   and
3. PRINT TAB( );"      ";TAB( );"      ";TAB( );"      "
```

to set up the headings given in the prior example. Use form 1 for the first line of the heading, form 2 for the second line and form 3 (the TAB form) for the third line.

Hint: Since form #1 uses automatic zone formatting and is not adjustable, the other forms have to be keyed to it.

Whether you follow TAB(##) with a semicolon or comma makes no difference. In either case, the Computer will start printing ## spaces to the right of the left margin. However, it is important to remember that whenever numbers or number variables are printed out, the Computer inserts one space to the left of the number to allow for the — or + sign. Type and RUN the following:

```
10 A = 3

20 B = 5

30 C = A + B

40 PRINT TAB(10);"A";TAB(20);"B";TAB(30);"C"

50 PRINT TAB(10);A;TAB(20);B;TAB(30);C
```

It should appear:

```
          A                   B                   C
           3                   5                   8
```

Note that the numbers are indented one space beyond the TAB(#). Keep it in mind when lining up (or indenting) headings and answers.

Change line 20 to read

```
20 B = -5
```

. . . and RUN. See why the indenting is necessary?

The Long Lines Division

Have you ever wondered what would happen if you wanted to PRINT a great number of headings or answers on the same line — but didn't have enough room on the program line to get in all the TAB statements? You have? Really? You're in luck because it's easy. Type and RUN the following program:

```
10 A = 1
20 B = 2
30 C=3
40 D=4
50 E=5
60 F=6
70 G=7
80 H=8
90 I=9
100 J=10
200 PRINT "A";TAB(50);"B";TAB(10);"C";TAB(15);"D";
210 PRINT TAB (20);"E";TAB(25);"F";TAB(30);"G";
220 PRINT TAB(35);"H";TAB(40);"I";TAB(45);"J"
300 PRINT A;TAB(5);B;TAB(10);C;TAB(15);D;TAB(20);
310 PRINT E;TAB(25);F;TAB(30);G;TAB(35);H;TAB(40);
320 PRINT I;TAB(45);J
```

It's the trailing semicolon (;) that does the trick. It makes the end of one PRINT line continue right on to the next PRINT line without activating a carriage return. The combination of TAB and trailing semicolon allows you almost infinite flexibility in formatting the output.

Multiple Statement Lines

As our parting shot in this easy lesson, we're going to have a sneak preview of what's ahead. Replace lines 10 through 100 with the following:

```
10 A=1:B=2:C=3:D=4:E=5:F=6:G=7:H=8:I=9:J=10
```

Egad, Igor — we've created a monster! *Will it work?* RUN and find out.

Worked just the same, didn't it?

Now don't get all carried away, but this is one of many cases where you can put a number of statements on the same line, separating them with the **COLON** (:). Be careful, **not** the semi-colon. The Computer reads them from left to right, as though each were a separate line number.

Don't try this trick with IF-THEN statements since there are some special considerations — but with virtually everything else we've learned so far, including FOR-NEXT loops, it works fine, saves space, shortens programs and has a lot going for it.

EXERCISE 12-2: Rework the answer to Exercise 11-3 to include the **Hourly** rate of pay in the printout. Use the TAB function to have the chart display all 5 columns side by side.

EXERCISE 12-3: (Optional) Rework the special problem 11-6 answer using the TAB function so the printout includes the internal resistance in a fifth column.

Learned in Chapter 12

Print Modifiers	Miscellaneous
TAB	Trailing semicolon
	Multiple statement lines

Chapter 13

Grandson of FOR-NEXT

The FOR-NEXT loop didn't go away for long. It returns more powerful than ever. Enter this program:

```
10 FOR A = 1 TO 3
20   PRINT " A LOOP"
30    FOR B = 1 TO 2
40     PRINT " ", " B LOOP"
50    NEXT B
60 NEXT A
```

. . . and RUN.

The result is:

```
A LOOP
                B LOOP
                B LOOP
A LOOP
                B LOOP
                B LOOP
A LOOP
                B LOOP
                B LOOP
```

For legibility, add two blank spaces in line 20 before PRINT; three in line 30 before FOR; four in 40 before PRINT; and three in 50 before NEXT.

This display vividly demonstrates operation of the *nested* FOR-NEXT loop. "Nesting" is used in the same sense that drinking glasses are "nested" when stored to save space. Certain types of portable chairs, empty cardboard boxes, etc. can be nested. They fit one inside the other for easy stacking.

Let's analyze the program a line at a time:

Line 1Ø establishes the first FOR-NEXT loop, called A, and directs that it be executed 3 times.
Line 2Ø prints "A Loop" so we will know where it came from in the program. See how this program line is indented several spaces to make it stand out as being nested in the "A" loop?
Line 3Ø establishes the second loop, called B, and directs that it be executed twice. It is indented even more so you can instantly see that it is buried even deeper in the "A" loop.
Line 4Ø prints two items: first the blank shown between the two quote marks, then the comma kicks us into the next print zone where "B Loop" is printed. Makes for clear distinction on the screen between the A loop and B loop, eh?
Line 5Ø completes the "B" loop and returns control to line 3Ø for as many executions of the "B" loop as line 3Ø directs. So far we have printed one "A" and one "B".
Line 6Ø ends the first pass through the "A" loop and sends control back to line 1Ø, the beginning of the A loop. The A loop has to be executed 3 times before the program run is complete, printing "A" 3 times and "B" 6 times (3 times 2).

Study the program and the explanation until you completely understand it. It's simple but powerful magic.

Okay, to get a better "feel" for this nested loop (or loop within a loop) business, let's play with the program. Change line 1Ø to read:

```
1Ø FOR A = 1 TO 5
```

. . . and RUN.

Right! A was printed 5 times, meaning the "A" loop was executed 5 times, and B was printed 1Ø times — twice for each pass of the "A" loop. Now change line 3Ø to read

```
3Ø  FOR B = 1 TO 4
```

. . . and RUN.

When you write programs, be sure to indent lines to highlight nesting (or other lines you want to emphasize). This helps when reading programs — and is a great aid when debugging (troubleshooting) program problems.

Nothing to it! A was printed 5 times and B printed 2Ø times. If you are having trouble counting A's and B's as they whiz by, you remember what to do. Just press the (↑), (↓) or (→) key to stop execution and temporarily freeze the display. The BREAK key and typing CONT do the same thing, allowing hands-off freezing, but inserts a BREAK note and otherwise messes up the display.

How to goof-up nested FOR-NEXT loops

The most common error beginning programmers make with nested loops is improper nesting. Change these lines:

```
50    NEXT A
60 NEXT B
```

. . . and RUN.

The Computer says:

```
WHAT?
60 NEXT B?
```

Looking at the program we quickly see that the B loop is **not** nested within the A loop. We have the FOR part of the B loop inside the A loop, but the NEXT part is outside it. This does not work. A later chapter deals with something called "flow charting", a means of helping us plan programs and avoid this type of problem. Meanwhile we just have to be careful.

Breaking out of Loops

Improper nesting is illegal, but breaking out of a loop when a desired condition has been met is OK. Add these lines:

```
50  NEXT B
55 IF A = 2 GOTO 100
60 NEXT A
99 END
100 PRINT "A EQUALLED 2. RUN ENDED."
```

. . . and RUN.

As the screen shows, we "bailed out" of the A loop when A equalled 2 and hit the test line at 55. The END in line 99 is just a precautionary roadblock set up to stop the Computer from running into line 1ØØ unless specifically directed to go there. That would never happen in this simple program, but we will use protective ENDs from time to time to remind us that lines which should be reached only by specific GOTO or IF-THEN statements must be protected against accidental "hits".

We'll be seeing a lot of the *nested* FOR-NEXT loop now that we know what it is and can put it to use.

EXERCISE 13-1: Enter the original program found at the beginning of this Chapter. It contains a B loop nested within the A loop. Make the necessary additions to this program so a new loop called "C" will be nested within the B loop, and will print "C LOOP" 4 times for each pass of the B loop.

EXERCISE 13-2: Alter the resident program so that it is the same as that found in the answer to Exercise 13-1.

Make the necessary additions to this program so a new loop called "D" will be nested within the C loop, and will print "D LOOP" 5 times for each pass of the C loop.

Learned in Chapter 13

Miscellaneous
Nested FOR-NEXT loops
Protective END blocks

Chapter 14

The INTEGER function

Integer??? "I can't even pronounce it, let alone understand it." Oh, come, come. Don't let old nightmares of being trapped in Algebra class stop you now. It's pronounced (**in**-teh-jur) and simply means a whole number like 1, or 2 or 3, etc. How difficult can that be? Come to think of it, some folks make a whole career of complicating simple ideas. We're here to do just the opposite.

The INTEGER function, INT(X), allows us to "round off" any number, large or small, positive or negative, into an integer, or whole number.

Type `NEW` to clear out any old programs, then enter:

```
30 X = 3.14159

40 Y = INT(X)

50 PRINT "Y = " ;Y
```

. . . and RUN.

The display reads

```
Y = 3
```

Oh — success is so sweet! It rounded 3.14159 off to 3. Change line 30 to read:

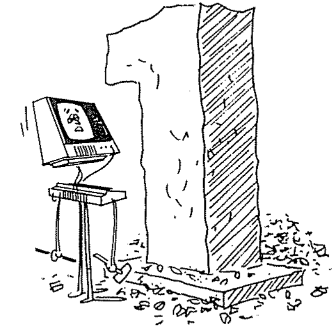
```
30 X = -3.14159
```

. . . and RUN.

Good Grief! It rounded the answer **down** to read

```
Y = -4
```

What kind of rounding is this? Easy. The INT function **always** rounds **DOWN** to the next **lowest** WHOLE number. Pretty hard to get that confused! It makes a positive number less positive, and makes a negative number more negative (same thing as less positive). At least it's consistent.



"CALL IT? INTEGER OF COURSE!"

Careful — we're not talking about ordinary rounding (what could be ordinary about your Computer?!). Ordinary rounding gives us the closest whole number, whether it's larger or smaller than X. INT(X), on the other hand, gives us the **largest** whole number which is less than or equal to X. As you'll see in this chapter, this is a very versatile form of rounding — in fact, you can use it to produce the other, "ordinary" kind of rounding.

NOTE: LEVEL 1 BASIC allows the INT(X) function to work only with numbers larger than −32767 and smaller than +32767. LEVEL 2 BASIC removes this restriction. Use of a value of X outside this range causes the Computer to halt execution and ask HOW?

Taking it a line at a time:

Line 3∅ set the value of X (or any of our 26 alphabet-soup variables) equal to the value we selected, in this case π.
Line 4∅ finds the INTEGER value of the above number and assigns it a variable name. We chose Y.
Line 5∅ prints a little identification (Y=) followed by the value of Y.

Not Content to Leave Well Enough Alone

We can do some foxy things you probably never thought of by combining a FOR-NEXT loop with the INTEGER function.

Change the program to read:

```
30 X = 3.14159

40 Y = INT(X)

50 Z = X - Y

60 PRINT " X = " ;X

70 PRINT " Y = " ;Y

80 PRINT " Z = " ;Z
```

. . . and RUN.

AHA! I don't know what we've discovered but it must be good for something. It reads:

```
X = 3.14159

Y = 3

Z = .141589
```

We've split the value of X into its Integer (whole number) value and called it Y, and its decimal value and called it Z.

Line 6∅, 7∅ and 8∅ merely printed the results.

Hold the phone!!!

Oh — oh! Why doesn't Z equal the **exact** difference between X and Y? Where did that "8" come from in the decimal value?

It has nothing to do with the INT function. Back in Chapter 8 we talked a little about the Computer's accuracy (you always have to watch the accuracy of the last decimal place or two). TRS-80 users who have LEVEL II Basic will not notice this routine "rounding error". If we solved **all** the world's problems with the bottom-of-the-line machine you might not want to upgrade to the higher power models, and one doesn't stay in business long that way, does one?

There **is** a way to control the accuracy of your results in LEVEL I BASIC. It involves artificially rounding your fraction to the desired number of decimal places, and then forcing the Computer to print out only those digits which are "properly rounded".

For example, suppose you only need π to three places. (Of course, you can enter it as 3.142, but that's not the point.) Type NEW , then enter and RUN the following program:

```
10 X=3.14159
20 X=X+.0005
30 X=INT(X*1000)/1000
40 PRINT X
```

Adding .0005 gives our fraction a "push in the right direction". If this fraction has a digit greater than 4 in its 10-thousandths-place, then adding .0005 will effectively increase the thousandths-place digit by 1. Otherwise, the added .0005 will have no effect on the final result. This results in what's called "4/5 rounding."

Try using other values for X (just make sure X*1000 isn't too large for the INT function to handle).

It's easy to change the program to accomplish rounding at a different point. For example, to round X off at the hundredths-place (2 digits to the right of the decimal point), change lines 20 and 30 to read:

```
20 X=X+.005

30 X=INT(X*100)/100
```

and RUN, using several values for X.

This is useful when you're printing out dollars-and-cents — it prevents $39.995-type prices.

Hmmmm!!!

Do you suppose there is any way to separate each of the digits in 3.14159, or in any other number? Do you suppose we would have brought it up if there wasn't? After all . . . (mumble, mumble . . .).

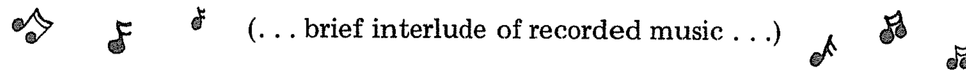
It's really your turn to do some creative thinking, but we'll get you started and see if you can finish this idea. First, wipe out the resident program and retype the program that splits X into an integer and fractional part (the first program in this Chapter).

We clearly can't just go on taking the INT value of X over and over to try and split down decimal value. Let's try it with Z.

```
90 L = INT (Z)

100 PRINT "L=";L
```

. . . and RUN.

Nope — that's a sure loser. We got Ø. The integer value of .141589 was that value rounded down to the next number, and the next number down was zero. Hmmm! Erase out lines 90 and 100 and let's try again. Got any better ideas? No? Well, think some more.



Time out for creative thinking!

Right! If we multiply the value of Z by 1Ø then Z will become a whole number plus a decimal part: 1.41589. We can then take its integer value and strip off the decimal part, leaving the left hand digit standing alone. Let's label the left-hand digit L and see what happens. Enter:

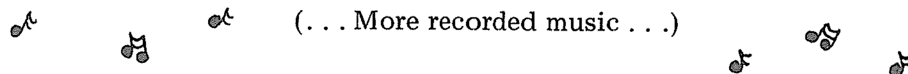
```
90 M = Z * 10

100 L = INT (M)

110 PRINT "L=";L
```

. . . and RUN.

Now, that's more like it. It reads:

```
X = 3.14159

Y = 3

Z = .141589

L = 1
```

We peeled off the leftmost digit in the decimal. Can you think of any way we might use a FOR-NEXT loop in order to strip off some more?



After all, these digits might not be just a more accurate value of pi, but a coded message from a cereal box. If you don't have the decoder ring it's tough luck, Charlie — unless you have a computer!

Enough thinking there on company time! Enter these lines:

```
95 FOR A = 1 TO 6

12Ø M = M - L

13Ø M = M * 1Ø

14Ø NEXT A
```

. . . and RUN.

Voila! (I never did figure out what that meant, but I think it's positive.) The "printout" reads:

```
X = 3.14159
Y = 3
Z = .141589
L = 1
L = 4
L = 1
L = 5
L = 8
L = 9
```

It's all there. Every digit, including the "squirrely" ones from the land of little numbers, is there. Analyzing the program additions (after doing a LIST):

Line 95 began a FOR-NEXT loop with 6 passes, one for each of the 6 digits right of the decimal.

Line 12Ø creates a new decimal value of M (just a temporary storage location) by stripping off the integer part. (Plugging in the values, M = 1.41589 — 1 = .41589)

Line 13Ø does the same as line 9Ø did, multiplies the new decimal times 1Ø so as to make the left-hand digit an integer and vulnerable to being snatched away by the INT function. (M = .41589 * 1Ø = 4.1589)

Line 14Ø moves the control back to line 95 for another pass through the clipping program . . . and the rest is history.

Is this too hard to follow?

No — it isn't hard to follow, and you could go through and indicate every value just like I did and it would be perfectly clear (to coin a phrase). Let's instead learn a way to let the Computer help us understand what it is doing.

We can insert temporary print lines anywhere in any program so we follow every step in its execution. The Computer can actually overwhelm us with data, but by carefully indicating what we want to know, we can observe the inner details of the calculations. Start by adding this line:

```
92 PRINT " #92 M = ";M
```

. . . and RUN

The essentials of this "test" or "debugging" or "flag" line are:

1. It PRINTs something.
2. The print tells the line number, for analysis and easy location for later erasure.
3. It tells the name of the variable you are watching at that point in the program.
4. It gives the **value** of that variable at **that point**.

It is most helpful of all when inserted in FOR-NEXT loops — so:

```
97 PRINT " #97 A = ";A
```

. . . and RUN.

Wow! The data really comes thick and fast! Hard to keep track of so much information, and we've barely begun. This tells what is happening during each pass of the loop. Is there some way to make it more readable? Sure. Can you think of a way?

Yes, there are lots of ways. Indenting is just one simple way to keep the answers separated from the trouble shooting data. Retype lines 92 and 97 as follows:

```
92 PRINT "   "," #92 M = ";M

97 PRINT "   "," #97 A = ";A
```

. . . and RUN.

Ahhh. How sweet it is. That is so easy to read, let's monitor some more points in the program. Type in:

```
125 PRINT "   ","   "," #125 M = ";M

135 PRINT "   ","   "," #135 M = ";M
```

. . . and RUN.

This "flagging" is such a wonderful troubleshooting tool in stubborn programs that you will want to make a habit of never forgetting to use it when the going gets tough.

There it is. All the data you can handle (and then some). By using the up or down arrow key to temporarily halt execution, you can study the data at every step to understand how the program works (or doesn't work). Do it. Understand this program and all its little lessons completely. When you are satisfied, go back and erase out the "flags". You have learned quite enough for this Chapter.

EXERCISE 14-1: Enter this straightforward little program for finding the area of a circle.

(First type NEW.)

```
10 P=3.14159

20 PRINT "RADIUS", "AREA"

30 PRINT

40 FOR R=1 TO 10

50  A = P * R * R

60  PRINT R,A

70 NEXT R
```

Area equals π times the radius squared (that is, the radius times itself). Then RUN it to make sure it works.

Pretty routine stuff — huh? Problem is, who needs all those little numbers to the far right of the decimal point. *Oh, you do?* Well, there's one in every crowd. The rest of us can do without them. Without giving any big hints, modify the resident program to suppress all the numbers to the right of the decimal point.

EXERCISE 14-2: Now, knowing just enough to be dangerous, and in need of a shot of humility, change line 55 so that each value of AREA is rounded (down) to be accurate to one decimal place. For example:

```
RADIUS               AREA
 1                    3.1
```

etc.

Ummm — yaas.

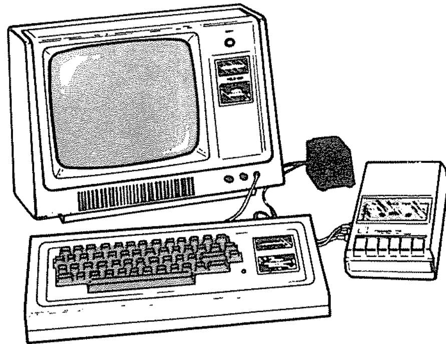
Hang in there. It's super-simple.

EXERCISE 14-3: Carrying the above assignment one step further, modify the program line 55 to round (down) the value of area to be accurate to 2 decimal places.

EXERCISE 14-4: At the risk of inducing complete boredom (yet teaching an unexpectedly important lesson) it's all-together-now: Revise line 55 to introduce 3-place accuracy in the AREA calculated by the resident program.

Learned in Chapter 14

Functions	Miscellaneous
INT(X)	Flags



Chapter 15

It Went That-A-Way

Enter this program:

```
10 INPUT "TYPE A NUMBER BETWEEN 1 AND 5";N
20 IF N = 1 GOTO 110
30 IF N = 2 GOTO 130
40 IF N = 3 GOTO 150
50 IF N = 4 GOTO 170
60 IF N = 5 GOTO 190
70 PRINT " THE NUMBER YOU TYPED WAS NOT BETWEEN 1 AND 5 --- DUMMY
!"
99 END
110 PRINT "N= 1"
120 END
130 PRINT "N= 2"
140 END
150 PRINT "N=3"
160 END
170 PRINT "N=4"
180 END
190 PRINT "N=5"
```

Notice anything funny about line 70? It takes up two lines on the Display! That's because it contains more than 64 characters (including line number and blank spaces). This is perfectly all right, as you may already have discovered in your own programming efforts. In fact, a program line can contain up to 72 characters (including line number and spaces). To enter or LIST such a long line takes up two Display lines; but it's still just one program line!

RUN it a few times to feel comfortable with it and be sure it is "debugged".

Anyway, this program works fine for examining the value of a variable, N, and sending the Computer off to a certain line number to do what it says there. If there are lots of possible directions in which to branch, however, we will want to use a greatly improved test called ON-GOTO which cuts out lots of lines of programming. Let's examine an ON-GOTO after you do the following:

Erase lines 2Ø, 3Ø, 4Ø, 5Ø and 6Ø

Enter this new line:

```
20 ON N GOTO 110,130,150,170,190
```

. . . and RUN the program a few times, as before.

Works just the same, doesn't it?

The ON-GOTO statement is really pretty simple, though it looks hard. Line 2Ø says,

if the INTEGER value of N is 1 then GOTO line 11Ø.
if the INTEGER value of N is 2 then GOTO line 13Ø.
if the INTEGER value of N is 3 then GOTO line 15Ø.
if the INTEGER value of N is 4 then GOTO line 17Ø.
if the INTEGER value of N is 5 then GOTO line 19Ø.
if the INTEGER value of N is not one of the numbers listed above, then move on to the next line.

The ON-GOTO statement has its own built-in INT statement. It really acts like this:

2Ø ON INT(N) GOTO . . . ETC.

Type in the following values of N to prove the point:

```
1.5

3.99999

0.999

5.999

6.0001
```

Get the picture?

Debugged is an old Latin word which, freely translated, means "getting all the errors out of your your computer program."

Remember, an *integer* is just a whole number.

Variations on a Theme

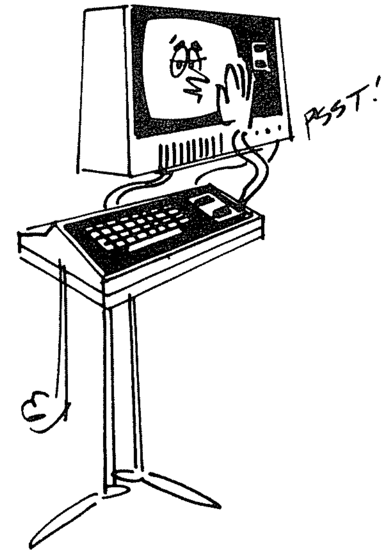
There are lots of tricks that can be played to milk the most from ON-GOTO. For example, if you want to branch out to 15 different locations but obviously cannot type that many different numbers on an ON-GOTO line, you can use several lines, like this:

```
2Ø ON N GOTO 11Ø,13Ø,15Ø,17Ø,19Ø

25 ON N-5 GOTO 21Ø,23Ø,25Ø,27Ø,29Ø

3Ø ON N-10 GOTO 31Ø,33Ø,35Ø,37Ø,39Ø
```

. . . and fill in the proper responses at those line numbers.

In line 25, it was necessary to subtract 5 from the number being input as N, since each new ON-GOTO line starts counting again from the number 1. In line 3Ø, since we had already provided for inputs between 1 and 1Ø, we subtract 1Ø from the input N to cover the range from 11 through 15. By using the ON-GOTO statement, we have programmed into 3 lines what would otherwise have taken 15 lines. By packing more branching options into each ON-GOTO line, we could have done it in 2 lines or less, depending on the number of digits in the line numbers of the branch locations.

As in most of our examples, we could have used any letter after "ON", not just N. As we just saw, N can be the value of a letter variable, or a complete expression, either calculated in place (as here) or in a previous line.



Trade Secret

Due to the vagaries of rounding error and the chance the error might just round a number like "N" a tad below the integer value expected, it is common to see something like this:

```
5Ø ON N+.2 GOTO 1ØØ,2ØØ,3ØØ ETC.
```

The effect of this shifty move is to add just a "pinch" to the incoming value of N, knowing full well that the ON-GOTO statement contains its own INT function. If N happens to have been rounded down to say 1.98 (instead of the 2.ØØØ expected), Ø.2 will be added to it making N 1.98 + .2 = 2.18 which the built-in INT will round down to the desired 2. Pretty sneaky. Values between .1 and .5 are often added to the N for this purpose in well-written programs.

Give Me a SGN(X)

Using the ON-GOTO along with a new function called SGN (it's pronounced sign), plus a modest amount of imagination, produces a most useful little routine. But first, let's learn about SGN.

The SGN function examines any number to see whether it is negative, zero, or positive. It tells us the number is negative by giving us a (−1). If the number is zero it gives us a (Ø). If positive, we get a (+1). It's a very simple function.

First, the BAD News

Unfortunately, LEVEL I BASIC does not have the SGN function built-in.

You can only get so many people in a telephone booth. (There is supposed to be an analogy of sorts there.)

Then, the GOOD News

Fortunately, through the use of a computer (yours) it is possible to create or simulate functions we don't have. That's why Appendix A is full of good things called SUBROUTINES.

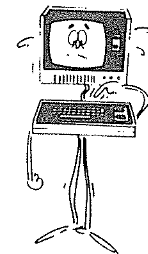
So What Is a Subroutine?

Funny you should ask. A sub-routine is a short but very specialized program (or routine) which you build into a large program to meet a specialized need. LEVEL II BASIC stores many of them in a special place in memory and they can be called up by a simple set of letters. (We have several at LEVEL I, like INT.)

We don't have enough memory to spare here at LEVEL I to hold all the routines in memory, so we are going to use a five-line subroutine instead of the "SGN" function to accomplish the same thing. Even if you have LEVEL II BASIC in your computer, you should complete this Chapter to be sure you learn about subroutines. We don't want to turn out dummies, you know.

Turn to Appendix A. Find the subroutine marked SGN. "Scratch" the program now in the computer by typing NEW, then — very carefully, so you don't make any mistakes, type in the SGN subroutine:

```
30000 END
30800 REM * SGN(X) * INPUT X, OUTPUT T = -1,Ø, OR +1
30810 IF X<Ø THEN T = -1
30820 IF X = Ø THEN T = Ø
30830 IF X >Ø THEN T = +1
30840 RETURN
```



"CAREFULLY"

"Calling" a Subroutine — (Sort of like calling hogs.)

When you want to use a subroutine, use the GOSUB##### statement.

This directs the Computer to go to that line number, execute what it says there and in the lines following, and when done RETURN back to the line containing the GOSUB statement. We will use line 2Ø here.

```
20 GOSUB 30800
```

A RETURN is always built into a subroutine, and you'll find it at line 3Ø84Ø. We have reserved line number 3ØØØØ to hold a protective END block for all of our subroutines, so the Computer doesn't come crashing into them when it is done with the main program.

"#####" represents the line number. All of our Appendix A subroutines use line numbers above 3ØØØØ.

Getting Down to Business

Okay, now let's combine GOSUB and SGN (using a subroutine) to see what all this fuss is about. Type:

```
10 INPUT "TYPE ANY NUMBER";X

20 GOSUB 30800

30 ON T+2 GOTO 50,60,70

45 END

50 PRINT "THE NUMBER IS NEGATIVE."

55 END

60 PRINT "THE NUMBER IS ZERO."

65 END

70 PRINT "THE NUMBER IS POSITIVE."
```

. . . etc. (the subroutine is already typed in) . . . and RUN.

Try entering negative, zero and positive numbers to be sure it works. Most of the program is already obvious to you, but here is an analysis:

Line 1Ø inputs any number.
Line 2Ø sends the Computer to line 3Ø8ØØ by a GOSUB statement. This is different from an ordinary GOTO, since a GOSUB will return control to the originating line like a

boomerang when the Computer hits a RETURN. The GOSUB is not completed and will not move onto the next program line until a RETURN is found.

Lines 3Ø8ØØ through 3Ø84Ø contain this rather simple subroutine.

Line 3Ø84Ø contains the RETURN which sends control back to line 2Ø, which silently acknowledges the return and allows movement to the next line.

Line 3Ø is an ordinary ON-GOTO statement, but adds 2 to the value of its variable, in this case "T". Line 3Ø is really saying, "If T is −1 then GOTO line 5Ø. If it is zero then GOTO line 6Ø, and if it is +1 GOTO line 7Ø. By adding 2 to each of those values we have "matched" them up with the 1, 2, and 3 which are built into the ON-GOTO.

Lines 45, 55, and 65 are routine protective blocks.

By the way, most subroutines are not this simple — as a matter of fact, they get into rather hairy mathematical derivations. We won't bother trying to explain any of them — if you're one of those Math nuts, you go right ahead and play with the numbers . . .

Preview of Coming Attractions?

Like so much of what we are learning, this is just the tip of the iceberg. The ON-GOTO and SGN functions have many more clever applications, and they will evolve as we need them. As a hint for restless minds, note that the **value** of X which we input was not used, but it didn't go away. All we did was find its SGN. Hmmm . . .

Routines vs SUBroutines

We studied a special-purpose routine used as a subroutine. It is one of the few that we can both use and really understand. All the routines, understandable or not, can be built directly into any program instead of being set aside and "called" as subroutines. Their main value as subroutines is that they can be "called" repeatedly from different parts of a program, which is often desirable. As ordinary routines they are usually only used once, and lines containing GOSUB and RETURN are not needed.

One value of using special routines as SUBroutines is that some are exceedingly complex to type without error, and if each is typed once and saved on cassette tape, it can be quickly and accurately loaded into the Computer as the first step in creating a new program. Another good idea is to type all the subroutines at one time, then record on one tape. You can later load that tape and erase out of the Computer those subroutines that are not needed for the program you are creating.

We'll have more to say in a later Chapter. When you see just how powerful subroutines are, you'll feel like your TRS-80 is even smarter than it thinks it is (blush, blush)!

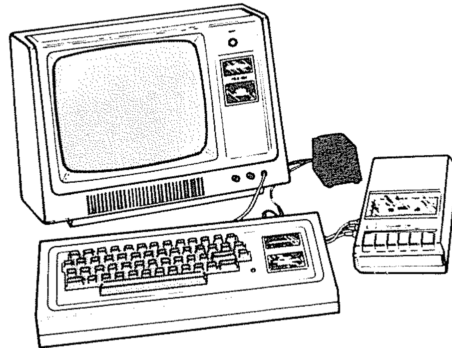
Now it's your turn.

EXERCISE 15-1: Delete lines 3Ø8ØØ-3Ø84Ø from the resident program. Build the SGN routine **into the program** so it works just as well as if we were calling it as a SUBroutine.

This problem will probably take you quite a few lines — so rather than give you a lot of blank space here, why not take advantage of the pad of Program sheets we've given you and write your program answer there. Then check Part B for our suggested answer.

Learned in Chapter 15

Functions	Statements	Miscellaneous
SGN(X)	ON-GOTO	Debugging
	GOSUB	Calling a subroutine
	ON-GOSUB	Routines
	RETURN	



Notes:

Chapter 16

READing DATA

So far, we have learned how to enter numbers into our programs by two different methods. The first is by building the value into the program:

```
10 A = 5
```

The second is by using an INPUT statement to enter a number through the keyboard:

```
10 INPUT A
```

The third principal way is through the DATA statement.

Enter this program:

```
10 DATA 1,2,3,4,5

20 READ A,B,C,D,E

30 PRINT A;B;C;D;E
```

. . . and RUN.

The DATA statement is in some ways similar to the first method in that a DATA line is part of the program. It's different, however, since each DATA line can contain many numbers, or pieces of data, each separated by a comma. Each piece of DATA must be read by a READ statement. Each READ statement can read a number of pieces of DATA if each variable letter is separated by a comma.

The display shows that all 5 pieces of data in line 1Ø, the numbers 1, 2, 3, 4 and 5 were READ by line 2Ø, assigned the letters A through E, and printed by line 3Ø.

Keep in mind this important distinction: DATA lines can be read **only** by READ statements. If more than one piece of data is placed on a DATA line, they must be separated by commas. Keyboard data can be entered **only** via INPUT statements.

DATA lines are always read from left to right by READ statements; the first DATA line first (when there is more than one), and IT DOES NOT MATTER WHERE THEY ARE IN

THE PROGRAM. This may seem startling, but do the following and you will see:

1. Move the DATA line from line 1Ø to line 25 and run. No change in the printout, right?
2. Move the DATA line from line 25 to line 1ØØØØ. Same thing — no change in the printout.

Data line(s) can be placed anywhere in the program.

This fact leads different programmers to use different styles. Some place all DATA lines at the beginning of a program so they can be read first in a LIST and found quickly so data may be changed.

Others place all DATA lines at a program's end where they are out of the way and there are more line numbers available to keep adding DATA lines as the need arises. Still others scatter the DATA lines throughout the program next to the READ lines which bring that data into use. The style you use is of little consequence — **but consistency is comfortable.**

The Plot Thickens

Since you now know all about FOR-NEXT loops, let us see what happens when a DATA line is placed in the middle of a loop. Erase the old program with NEW and type in this program:

```
1Ø DATA 1,2,3,4,5
2Ø  FOR N = 1 TO 5
3Ø   READ A
4Ø   PRINT A;
5Ø  NEXT N
```

. . . then RUN.

That DATA line started outside the loop. Now move it to line 25 and RUN. What happened?

Nothing different! It is important to note this fact or we wouldn't have gone to the trouble to do it. Note that as we went through the N loop 5 times, we read the letter A, and the PRINT statement only printed A, but A's value was different each time. Its value was the same as the value it last READ in the DATA line. The reason — each piece of data in a DATA line can only be read **once** each time the program is run. The next time a READ statement requests a piece of data, it will read the **next** piece of data in the DATA line, or, if that line is all used up, go on to the next DATA line and start reading it.

Change line 2Ø in the program to read:

```
2Ø FOR N = 1 TO 6
```

. . . and RUN.

We, of course, told the READ statement to read a total of 6 pieces of DATA but there were only 5. An error statement caught us, as the screen shows.

```
1 2 3 4 5 HOW?

3Ø READ A?
```

Now change line 2Ø so the number of READs is less than the DATA available

```
2Ø FOR N = 1 TO 4
```

. . . and RUN.

The program ran just fine as long as we didn't use all the available data. The point is, each piece of data in a DATA statement can only be read once during each RUN.

Exceptions, Exceptions!

Because it is sometimes necessary to read the same DATA more than once without having to RUN the complete program over, a statement called RESTORE is available. Whenever the program comes across a RESTORE, all DATA lines are restored to their original "unread" condition, both those that have been read and those that have not, and all are available for reading again, starting with the first piece in the first DATA line. Change line 20 of the program back to

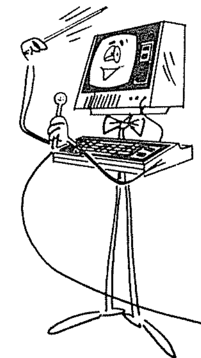
```
2Ø FOR N = 1 TO 5
```

and insert

```
35 RESTORE
```

and RUN.

Oh-oh! The screen prints five 1's instead of 1 2 3 4 5. Can you figure out why?



"ONE MORE TIME?"

Line 3Ø READ A as 1, but line 35 immediately RESTOREd the **DATA LINE TO ITS ORIGINAL UNREAD CONDITION**. When the FOR-NEXT loop brought the READ line around for the next pass it again read the first piece of data, which was that same 1. Same thing with all successive passes.

READ and DATA statements are extremely common. The RESTORE statement is used less often.

String Variables

Who knows where some of these seemingly unrelated words come from? If they weren't so important we could ignore them. We have been using the letters A through Z to indicate numbers. They are called NUMERIC VARIABLES. In LEVEL I BASIC we have set aside 2 additional symbols to indicate STRING VARIABLES. They are A$, and B$, pronounced "A String" and "B String". String variables can be assigned to indicate **Letters, Words** and/or **Combinations** of letters, numbers and spaces of up to 16 characters. Type NEW , then type in:

```
1Ø INPUT "WHAT IS YOUR NAME";A$

2Ø PRINT "HELLO THERE, ";A$
```

. . . and RUN.

Hey-hey! How's that for a grabber? If that, along with what you have learned in earlier chapters doesn't make the creative juices flow, nothing will.

That's Two . . . . .

Two ways we now know to print words. The first, learned long ago, is to imbed words in PRINT statements (and is called "printing a string"). The second is to bring in a word(s) through an INPUT statement (called "inputting a string"). If you can't think of the third way, go back and check the title heading at the first of this chapter.

Ah yes, brilliant student! Ahem . . . *(Reading a string.)*

Change the program to read:

```
1Ø READ A$

2Ø DATA RADIO SHACK TRS-80

3Ø PRINT "SEE MY FOXY";A$
```

. . . and RUN.

See! I told you a string variable would only hold 16 characters. Count them. Any suggestions??

But of course. Level I BASIC has two string variables available. Let's rework the program to print the entire name of the computer.

```
10 READ A$

15 READ B$

20 DATA RADIO SHACK, TRS-80

30 PRINT "SEE MY FOXY"; A$; " ";B$
```

That's more like it. Analyzing the program.

Line 20 contains two Data items, separated by a comma.
Line 10 READs the first one.
Line 15 READs the second one.
Line 30 contains 4 print expressions. The first one prints SEE MY FOXY, leaving a space behind the "Y" since string variables always run letters together, allowing you the option of inserting your own space. The second print is A$, RADIO SHACK. The third print is the space enclosed in quotes. The last print is TRS-80.

In other words, a semi-colon between string variables does NOT cause a space to be PRINTed between them. So you have to insert a space using " " marks.

EXERCISE 16-1: Okay, now it's your turn. Design a program to produce exactly the same results, but using only A$, not B$.

Stuck? Hint: Try a FOR-NEXT loop.

A lot of your learning to date is tied up in this little program, so be sure you completely understand it before you move on.

A Voice from the Past

Remember how in a very early chapter we checked the contents of each Numeric Variable address, A through Z, by using the calculator mode?

```
PRINT A;B;C;D;E; etc.
```

We can do the same thing with our two new Strings. Type:

```
PRINT A$;B$
```

Why does it display

TRS–80TRS–80 ?

The first TRS-80 is simple — it was the last string read by A$. Same thing with B$. Even though B$ was not used in solving the last problem, it was used in the earlier example, and if the Computer was not turned off since then, it was held in memory. This fact is more than a laboratory curiosity. It can get you into "unexplainable" programming problems if you're not aware of it.

Oh, by the way . . .

There isn't room in LEVEL I BASIC to do everything, obviously, and we promised earlier that you would learn how to answer "YES" and "NO" to the Computer. LEVEL II allows you to do it in a straight-forward manner. Here in LEVEL I we have to be sneakier. Enter this program:

```
1Ø Y = 1
2Ø N = Ø
3Ø INPUT "ARE YOU OLD ENOUGH TO VOTE (Y/N)";A
4Ø IF A = 1 THEN 6Ø
5Ø PRINT "DON'T FRET. THE TIME WILL PASS FAST ENOUGH."
59 END
6Ø PRINT "SWELL. DON'T FORGET TO REGISTER!"
```

. . . and RUN.

Analysis:

Line 1∅ sets the value of Y equal to 1.

Line 2∅ sets the value of N equal to ∅. These two values can be quickly and easily checked by typing PRINT Y,N in the calculator mode.

Line 3∅ inputs the answer to the question as either Y or N, (it will also accept YES or NO and any other words starting with Y and N). "A" takes on the value of Y or N as defined in lines 1∅ and 2∅.

Line 4∅ tests the value of A, and if it is 1, sends control to line 6∅. If it is not 1 (but not necessarily ∅), the line 4∅ test defaults and falls through to line 5∅. The appropriate message is printed.

Line 59 is a protective END so if line 5∅ is printed, line 6∅ will not also be printed.

More Analysis:

We have carefully given Y and N values of 1 and ∅. This does not mean that other letters might not have those same values. That does not matter as long as only Y and N are hit. The present program relies on line 4∅ defaulting to the next line if a 1 is NOT found. As a partial precaution against a user hitting the wrong letter accidentally and coming up with the wrong answer, we can "backstop" our program with these additional lines:

```
45 IF A = Ø THEN 5Ø

47 PRINT "PLEASE ANSWER WITH EITHER A Y OR N!"

49 END
```

Analysis:

Line 45 insists that to get the response to line 5∅, zero must be entered. We know that an N will do that for sure. Other letters might also have the value of zero.

Line 47 gives the "default" answer, cautioning the operator that he gave other than a YES or NO.

Line 49 protects against printing the default answer and running into the line 5∅ answer.

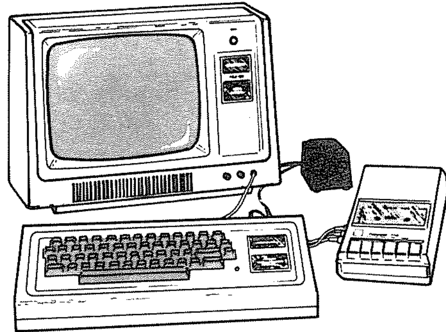
Try a PRINT A;B;C;D;E;F;G;[etc.] routine to see if any other letter variable is storing a value of 1 or 0.

EXERCISE 16-2: Design and write a simple program that asks the user at least 5 questions, and in the process carries on a little conversation with him.

You're on your own with this one.

Learned in Chapter 16

Statements	Miscellaneous
READ	String Variables A$, B$
DATA	Numeric Variables
RESTORE	(Y/N) — Teaching TRS-80 to respond to YES or NO



Chapter 17

Coming Up for Air

We've had a number of heavy chapters, and more are coming. Exciting as all this is, we need a break. How about a super-short (but important) chapter? Tho't you'd agree.

Absolute Value

The ABSOLUTE VALUE of any number is that number without any plus or minus sign. Just the number. Easy enough?

Type:

```
10 INPUT "TYPE ANY POSITIVE OR NEGATIVE NUMBER";X

20 Y = ABS(X)

30 PRINT "X","Y"

40 PRINT X,Y
```

. . . and RUN , inputting different number values, both positive and negative.

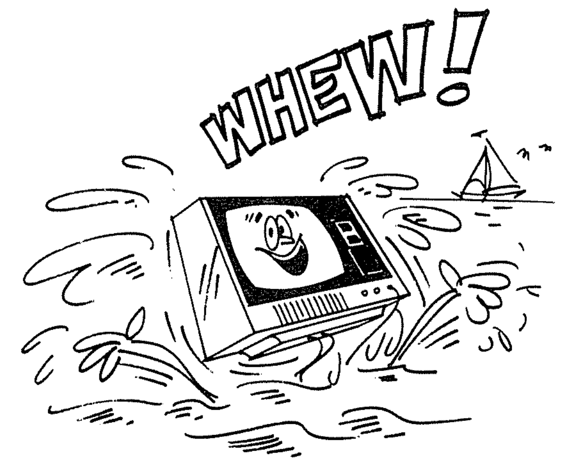
Regardless of what number you input (as X), its absolute value "Y" is that same number without the sign. That's "sign" like in SGN from an earlier chapter.

When you're done playing with this one and understand it, the chapter is over.

Learned in Chapter 17

Function

ABS(X) = absolute value of X



Here's a more technical definition:
If X is less than Ø, then ABS(X)= -1*X
If X = Ø, then ABS(X) = X = Ø
If X is greater than Ø, then ABS(X) = X

Told you it'd be a shorty didn't I?

Notes:

Chapter 18

Now He Tells Me!

This may blow your mind even though you have suspected it all along. The Radio Shack LEVEL I BASIC interpreter has a "shorthand." It is not some wild variation of the language, or a "regional dialect", but a genuine shorthand. We have deliberately not used it until you were well into the language so you would learn how to communicate with all those other folks out there who don't have this shorthand provision.

Hang Onto Your Chair

Nearly every COMMAND, STATEMENT and FUNCTION has a shorthand notation which is much shorter and easier to type, and does exactly the same thing. **The complete list is inside the back cover.** Here is a list covering those you have learned to date:

NEW = N.

LIST = L.

RUN = R.

PRINT = P.

MEM = M.

STOP = ST.

CONT = C.

THEN = T.

END = E.

ABS = A.

RESTORE = REST.

GOTO = G.

INPUT = IN.

FOR = F.

NEXT = N.

CSAVE = CS.

CLOAD = CL.

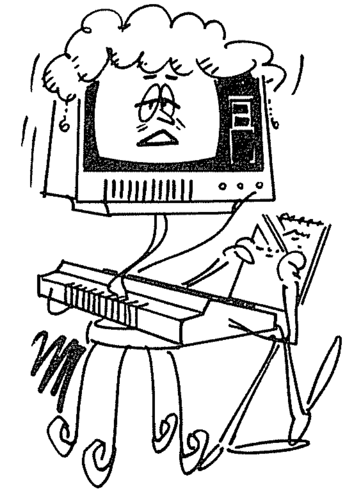
STEP = S. (when used with FOR/NEXT)

TAB = T. (When used with PRINT)

INT = I.

DATA = D.

READ = REA.



"N-C-ST-GO ON, GO ON-"

Remember our power-up test back in Chapter 1? Now you can figure out what the test tests.
P.M. *Let's see, that's . . .?*

There's More?

In addition, your Radio Shack Interpreter allows you to put more than one statement on each numbered line, separating them by a colon (:). For example, a timer loop such as:

```
100 FOR N = 1 TO 500

110 NEXT N
```

becomes . . .

```
100 FOR N = 1 TO 500:NEXT N
```

. . . shorter still . . .

```
100 F.N =1TO500:N.N
```

. . . or even less space . . .

```
100F.N =1TO500:N.N
```

The last listing can be entered into the Computer as shown, with no space between the line number and the first letter, but the Computer will automatically insert a space. Entering it this way does conserve the one extra space that you have been inserting to this time, however.

Caveat Emptor (Don't buy a used chariot from a stranger.)

Control yourself! It's easy to get carried away. While we will be using both Radio Shack Shorthand and multiple statement lines often from here on, you will quickly see that it's possible to pack the information so tightly that it becomes hard to read, and also very hard to modify. For most of this Manual we will avoid multiple statement lines, but when we want to really pack it tight — this is the only way to go!

More Caveat (or is it more Emptor?)

Radio Shack Level I Shorthand is nearly foolproof. (Knowing some of our customers, we can't give an unconditional guarantee; or, to broaden our coverage, let's say we know *ourselves* well enough, too!), but multiple statement lines require careful understanding. Especially critical are statements of the IF-THEN variety.

Enter the following program:

Yet another spacesaver involves the IF-THEN statement. The use of THEN is optional. That is, instead of

IF (condition) THEN (statement)

it's perfectly okay to write

IF (condition) (statement)

An example of this would be:

10 IF A=20 STOP

Note that you can't say

IF (condition) (line number).

You have to add either THEN or GOTO:

IF (condition) G. (line number)

or

IF (condition) T. (line number)

```
10 IN."TYPE IN A NUMBER";X

20 IF X = 3 THEN 50 : G.70

30 P."HOW DID YOU GET HERE?"

40 END

50 P."X =3"

60 END

70 P."CAN'T GET FROM THERE TO HERE."
```

. . . and RUN it a number of times with different input values.

Line 20 is illegal. If the test in the first statement in the line passes, control branches off to line 50. That's OK. If the test fails, however, control drops to the next line in the program — line 30. There is no way the second statement in line 20 (G.70) can ever be executed.

THE MESSAGE — if you put an IF-THEN (or ON-GOTO) type-test in a multiple statement line, it must be the **last** statement in that line. Other invalid procedures will be called to your attention as they are studied in future chapters.

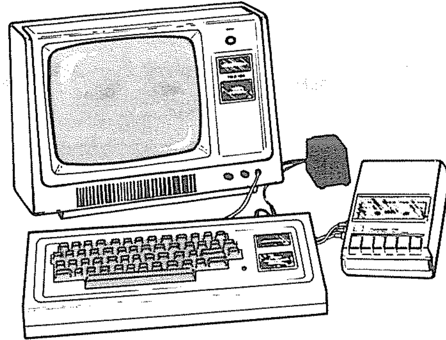
NEXT MESSAGE — you cannot send control to any point in a multiple statement line except to its first statement. Look at Line 20 in the resident program. Even if the G.70 was legal in that line, there is no way to address it. It shares the same line number as the first statement in the same line. Only the first statement is addressable by a GOTO or IF-THEN.

NEXT MESSAGE — DATA lines cannot exist on lines with other statements.

EXERCISE 18-1: Rewrite any one of the programs found in Part C, using every Radio Shack Shorthand feature possible, and multiple statement lines. Use the P.M. (PRINT MEMORY) test to see how much memory is being used. Rework the program to cut it to the smallest memory figure possible.

Learned in Chapter 18

Statements	Miscellaneous
IF (condition) (statement) with THEN omitted	LEVEL I shorthand dialect
	Multiple statement lines



Chapter 19

A RANDOM number is one with a value that is unpredictable. A Random Number Generator pulls random numbers out of a hat. We have a Random Number Generator and you set it up this way:

```
N = RND(X)
```

Where N is the random **number**
RND is the abbreviation and symbol for **random**
X is a control number which can be either typed between the parentheses or brought in as a variable from elsewhere in the program.

Type:

```
5Ø PRINT RND(Ø)
```

. . . and RUN.

RUN it again — at least 1Ø times.

Did you observe:
1. A different number appeared each time?
2. All numbers were between Ø and 1?
3. Very small numbers were expressed in exponential notation. RUN some more until you are satisfied that these statements are true.

Wait a minute — all this RUNning is dumb. You have a Computer! "Build" a FOR-NEXT loop around this Random Number Generator and let it run itself 1Ø times.

What program did you write? I wrote:

```
4Ø FOR N = 1 TO 1Ø

5Ø PRINT RND (Ø)

6Ø NEXT N
```

"RUNNING IS IN- OR IS IT?"

RUN it a few times to get the idea.

Let's put a semi-colon behind our PRINT statement so we can get more numbers on the screen at one time, and increase our FOR-NEXT loop to 9∅ passes. Change your program accordingly:

```
4Ø FOR N = 1 TO 9Ø

5Ø PRINT RND(Ø);

6Ø NEXT N
```

. . . and RUN.

You get the idea.

Are you using R. now instead of RUN? Sure is easier, isn't it?

This is fairly exciting!

Well, maybe so, but you ain't seen nothing yet! Virtually all computer games are based on the RND(X) function, and you'll soon be playing some and designing your own.

RND(X) with racing stripes.

The RND(∅) we just experimented with is the traditional Random Number Generator. In other BASIC dialects you may see it written as just plain RND. With a little mathematical chicanery and use of the INT function it is possible to turn those numbers between ∅ and 1 into something useful. Rather than study that technique, however, let's look at the Radio Shack upgrade which does it all so much easier. Change line 5∅ to read:

```
5Ø PRINT RND(15);
```

. . . and RUN.

We're just trying to tell you that "ours" is better than "theirs". With Radio Shack's Random Number Generator we can come up with all the numbers without all that fancy footwork.

Wow! That's more like it — real live random integers. And they all are values that fall between 1 and 15. Figured it out already? Pretty simple, isn't it?

1. If the number in parentheses (or its INT value) is ∅, the numbers generated are between 0 and 1.
2. If the number in parentheses is 1 or larger, the numbers generated are from 1 to the INT value of that number (inclusive).
3. In LEVEL I BASIC, the largest permissible value of X is 32767.

   Skeptical? You don't believe the numbers are really random? You want proof? A natural reaction. OK — how about pretending to repeatedly flip a coin and see how many heads come up compared to the number of tails?

The Old Coin Toss Gambit.

Remember now, you could toss a thousand heads in a row and the odds on the next toss are **exactly** 5Ø/5Ø that a head will come up again. Every toss is totally independent of what happened before it. *IT IS TOO!!!!!* In the **long run** however, the number of heads and tails should be exactly the same. (Casinos live off people who go broke waiting for their particular scheme to pay off . . . "in the long run"). Your Computer will give you a complete education in "odds" and various games of chance, and allow you to prove or disprove many ideas involving probability. This is known as computer "modeling" or "simulation."

We're going to write this coin toss simulation program in Radio Shack Shorthand and use multiple statement lines just for the practice. Type it in very carefully to avoid errors:

```
1Ø  H=Ø:T=Ø:P.

2Ø  IN."HOW MANY TIMES SHALL WE FLIP THE COIN";F:CLS

3Ø  P."YOU STAND BY WHILE I DO THE FLIPPING - - - - -"

4Ø  F.N=1TOF:X=RND(2):ONXG. 6Ø,7Ø

5Ø  P."IT BOMBED! WAS NEITHER A 1 NOR A 2.":END

6Ø  H=H+1:G.8Ø

7Ø  T=T+1

8Ø  N.N:P.:P.:P.:P.

9Ø  P."HEADS","TAILS","TOTAL FLIPS":P.:P.H,T,F

1ØØ P.1ØØ*H/F;"%",1ØØ*T/F;"%":P.:P.:P.
```

. . . and RUN. "Flip the coin" 1ØØ times on the first RUN to get a feel for the program and the run time. RUN as many times as it takes to convince you that the random number generator produces really random numbers. When it's time for lunch or you can wait quite awhile for the answer, try 25,ØØØ flips or more.

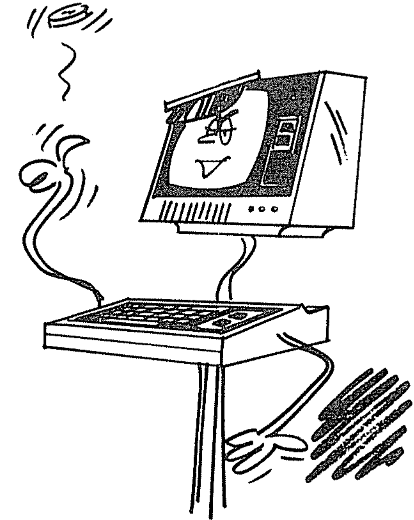
Program analysis:

Line 1Ø contains 3 statements. The first initializes H (for Heads) at zero. The second sets T (for Tails) equal to zero. The third inserts a space in the printout with a PRINT(P.).

Line 2Ø has 2 statements. 1 — Inputs the number of flips desired. 2 — A clear screen (CLS) to start the next print line at the top of the screen.

Line 3Ø Prints a "Standby" statement.

Line 4Ø has 3 statements. 1 — Begins a FOR-NEXT loop that runs "F" times. 2 — Is



"YOU LOSE!"

the RND(X) generator. We have told it to generate integers between 1 and 2, and of course that restricts it to just the numbers 1 and 2. Heads is "1" and Tails is "2". 3 — An ON-GOTO test sends X=1 to line 6Ø where the "Heads" are counted, and X=2 to line 7Ø where the "tails" are counted. **Note that this test is the last statement in the multiple statement line.**

Line 5Ø is a default line. If X = other than 1 or 2, the error message will be printed and execution will END. It will never happen, but you are insisting on proof.

Line 6Ø sets up H as a counter. H was initialized as zero in line 1Ø, and each time the ON-GOTO test sends control to this line because X=1, H is incremented by one and keeps count of the "Heads". The second statement sends control to line 8Ø where only the first statement, NEXT N, (N.N), is executed. When the N Loop has gone through all "F" number of passes, control in line 8Ø will move to the 4 blank PRINT(P.) statements. Until then, the N.N. sends it back to line 4Ø.

Line 4Ø generates another random number (1 or 2). If the next X=2 the ON-GOTO sends control to line 7Ø.

Line 7Ø keeps track of the tails, then passes to Line 8Ø and the NEXT N. When the last "N" is "used up", it inserts 4 blank print lines and falls to . . .

Line 9Ø where the Headings are printed, then the blank line, then the values of H,T and F.

Line 1ØØ calculates and prints the percentage of heads, and percentage of tails, and then prints 3 blank lines at the end to make the display look less cluttered.

More Than One Generator at a Time

It is possible to generate more than one random number by using more than one generator in a program. This has special value when the ranges of the generators are different, but is helpful even if their ranges are the same.

It can also be done with a single generator, but that wouldn't make our point . . . would it!

To make the point, we are going to get you started creating a computer game of "Craps" — where 2 dice are "rolled". Each "die" has six sides, each side having 1, 2, 3, 4, 5 or 6 dots, respectively. When the 2 dice are rolled, the number of dots showing on their top sides are added. That sum is important to the game. Obviously, the lowest number that can be rolled is 2, and the highest number is 12. We will set up a separate Random Number Generator for each die, give each a range from 1 to 6, and call them die "A" and die "B".

Type NEW , then the following:

```
5Ø  A=RND(6):B=RND(6):N=A+B

6Ø  P.N
```

. . . RUN a few times to get the idea.

As you can see, each number printed falls between 2 and 12. We are able to put both of our generators and the adder on the same line since the dice are always both thrown at the same time, and only the total is of interest here.

Why would the following be wrong?

```
5Ø P.RND(11) + 1
```

5Ø

Answer: Adding random numbers created by two generators, each picking numbers between 1 and 6 will create many more sums which equal 3, 4, 5, 6, 7, 8, 9, 1Ø and 11 than a single generator which picks an equal amount of numbers 1 through 11 (to which we add 1, to make the range 2 through 12).

Rules of the Game

In its simplest form, the game goes like this:

1. The player rolls the two dice. If he rolls a sum of 2 (called "snake eyes"), a 3 (called "cock-eyes") or a 12 (called "boxcars") on the first roll, he loses and the game is over. That's "craps".
2. If the player rolls 7 or 11 on the first throw, (called "a natural"), he wins and the game is over.
3. If any other number is rolled, it becomes the player's "point". He must keep rolling until he either "makes his point" by getting the same number again to win, or rolls a 7, and loses.

EXERCISE 19-1: You already know far more than enough to complete this program. Do it. Put in all the tests, print lines, etc. to meet the rules of the game and tell the player what is going on. It will take you awhile to finish, but give it your best before you turn over to Part C (User's Programs) under Craps for a sample solution. Good luck!

Use some of your blank Program Sheets for writing up this program.

Random numbers are unpredictable; properly functioning computers are not. So how do we get random numbers out of our Computer? We don't: we get **pseudo-random** numbers. Each time you use the RND function, the Computer uses an internal "seed number" to produce the desired random number.

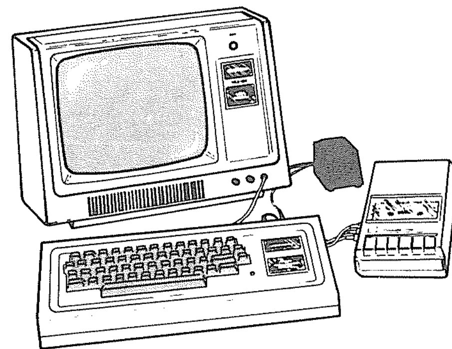
This is neither the time nor the place to get technical, so we'll give the following tip without further explanation:

When you're running game programs using RND, it's a good idea to set the seed to an unpredictable value. This will ensure that you don't get the same pseudo-random number sequence each time you turn on the Computer and play the game. Put the following lines at the beginning of your program where they will be executed only **once**:

```
1 IN. "ENTER A NUMBER BETWEEN 1 AND 1ØØ";N

2 F. I=1 TO N : J=RND(32767) : N.I
```

Learned in Chapter 19

Functions	Miscellaneous
RND(Ø) for random numbers greater than Ø and less than 1	Random vs. Pseudo-random
RND(N) for random numbers from 1 to N	Seed numbers



Chapter 20

And It Draws Pictures Too!

Your TRS-80 can draw an endless variety of pictures on the Video Display screen. You will learn some of the basic procedures and capabilities in this Chapter. Later on there's even more pictures! After that, what you create is limited only by your own imagination. Who knows . . . you may write a graphics program artistically equivalent to the Mona Lisa.

Now, on to 2 of the 4 graphic commands:

SET turns on (or lights up if you will) a particular section, block or "light" on the screen.

RESET turns off (or blackens) a particular "light".

For graphics, the screen is divided into a large number of sections. See the Video Display Worksheet on the next page. Each "light" is a rectangular block 2 dots wide by 8 dots high; and each has its own "address".

For example:

```
SET (55,32)
```

means — "turn on the light" at the junction of 55th "X" Street and 32nd "Y" Avenue.

X is the horizontal address counting across from the left-hand side of the screen. Y is the vertical address, counting down from the top of the screen. So everything starts from the upper left-hand corner.

Type in

```
5Ø SET (55, 32)
```

Clear the screen and RUN.

There it is! The light came on. Check the Video Display Worksheet carefully to find the address of that light. Did it show up in about the right place??

TRS-80 Video Display Worksheet

TITLE ________________ PROGRAMMER ________________ COMMENTS ________________________________ PAGE _____ OF _____

TAB→ 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63

X→ 0 1 2 3 4 5 6 7 8 9 10 11 … 126 127

PRINT AT ↓	Y	Y	PRINT AT
0	0, 1, 2	0, 1, 2	63
64	3, 4, 5	3, 4, 5	127
128	6, 7, 8	6, 7, 8	191
192	9, 10, 11	9, 10, 11	255
256	12, 13, 14	12, 13, 14	319
320	15, 16, 17	15, 16, 17	383
384	18, 19, 20	18, 19, 20	447
448	21, 22, 23	21, 22, 23	511
512	24, 25, 26	24, 25, 26	575
576	27, 28, 29	27, 28, 29	639
640	30, 31, 32	30, 31, 32	703
704	33, 34, 35	33, 34, 35	767
768	36, 37, 38	36, 37, 38	831
832	39, 40, 41	39, 40, 41	895
896	42, 43, 44	42, 43, 44	959
960	45, 46, 47	45, 46, 47	1023

↑ Y

X: 0 1 2 3 4 5 6 7 8 9 10 11 … 126 127

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63

Careful now, don't mess up the screen. Type

```
50 RESET (55,32)
```

and RUN.

How about that. You found the ON-OFF switch!

Want to really press your luck? Try turning the light back on. That's right, type

```
50 SET (55, 32)
```

and RUN 5 times in a row. Then 50 RESET (55,32) and RUN.

Oh well, can't win 'em all. Why didn't it work? It has to work. It **did** work! Then why didn't the fool light go OFF? **Answer**: The carriage return keeps moving it up away from its original address, and only what's at a specific address gets turned ON and OFF. **The screen addresses never move.**

The point of all this obviously is that we can control whether each block on the screen is white or dark (on or off) by "talking" to it at its individual address with SET and RESET statements.

Blinking Lights in the Sky – Flying Saucers or Lightning Bugs?

If one has an ON-OFF switch, what does one do with it? Is that what's called a rhetorical question? With a little imagination one could create blocks that don't just go ON and OFF, but do so to attract attention . . . *by blinking*. This simple program illustrates how to set up a "blinker".

```
10 CLS

20 X = 60

30 Y = 25

40 SET(X,Y)

50 RESET(X,Y)

60 GOTO 40
```

Back For More . . .

In the horizontal direction, there are 128 light-block addresses, numbered from 0 to 127. 0 is at the far left, 64 is near the middle and 127 is at the far right.

In the vertical direction, there are 48 light-block addresses numbered from Ø to 47. Ø is at the top and 47 is at the bottom.

The statement "SET (X,Y)" whitens the block which is the Xth block from the left in the horizontal direction and the Yth one down from the top in the vertical direction. And, you've figured out that RESET works the same way except that it "turns the light off".

Let's try it out. This program will lighten any one block of your choosing. Type:

```
1Ø INPUT "HORIZONTAL ADDRESS (Ø TO 127) IS";X

2Ø INPUT "VERTICAL ADDRESS (Ø TO 47) IS";Y

3Ø CLS

4Ø SET (X, Y)
```

and RUN many times using various values of X and Y.

What happens if X = 15Ø? Try it. How about if Y = 6Ø? Try it, too. The block just moves off the end (or bottom) of the screen and starts over with the address count. We call this "wrap around".

You may have noticed that if a block is lit in the upper left-hand corner, the READY and the prompt (>) destroy it. Try X = 6 and Y = 6. Then X = 5 and Y = 5. We can avoid this problem by not returning control to the prompt — by adding

```
99 GOTO 99
```

at the end of the program. After running the program, this line locks the Computer in an endless loop. To break the loop, press BREAK key. You should put an endless loop at the end of every graphics program. Do it here, then try X = 5 and Y = 5. Now try X = Ø and Y = Ø. Remember the BREAK key to stop a properly "locked out" graphics program, before starting another.

While we have a key that RESETs every block on the screen to "OFF" in one operation (the CLEAR key), we don't have a similar key to turn them all "ON".

However, we can easily write a program that "lights", "whitens" or "paints" the entire screen. It uses one Clear (not really a must, but always a good habit to use one for graphics programs), two FOR-NEXT loops and one endless "locking loop". Type this:

```
1Ø CLS
```

```
20 FOR X = Ø TO 127
30  FOR Y = Ø TO 47
40   SET (X, Y)
50  NEXT Y
60 NEXT X
99 GOTO 99
```

and RUN.

The resident program fills the screen from left to right. Redesign it so it starts at the top and fills to the bottom.

Answer:

```
10 CLS
20 FOR Y = Ø TO 47
30  FOR X = Ø TO 127
40   SET (X, Y)
50  NEXT X
60 NEXT Y
99 GOTO 99
```

Next, rewrite it so it starts painting at the bottom and fills to the top.

NOTE: When running graphics, you'll probably want to turn up both the Contrast and Brighteness slightly.

Don't forget . . . first you have to use the BREAK key to stop the endless loop.

Answer:

```
10 CLS
20 FOR Y = 47 TO 0 STEP-1
30  FOR X = 0 TO 127
40   SET (X, Y)
50  NEXT X
60 NEXT Y
99 GOTO 99
```

Didn't know you could STEP it backwards, eh? Try a few of those and see how it works. Try different increments (—2, —8, etc.).

OK, now rewrite it so it starts painting at the right-hand side and fills to the left-hand side.

Answer:

```
10 CLS
20 FOR X = 127 TO 0 STEP-1
30  FOR Y = 0 TO 47
40   SET (X, Y)
50  NEXT Y
60 NEXT X
99 GOTO 99
```

Try some other step increments too . . .

Fantastic — now you can paint the old barn at least 4 ways!

EXERCISE 20-1: Write a program which will allow painting only a small part of the screen (you determine which part). Allow keyboard INPUT of the starting and ending block numbers in both the horizontal and vertical directions.

Getting the hang of it?? Great. Enough playing with blocks . . . let's draw some lines (really moving on, eh?!). Erase the resident program.

We'll start with a straight line. This program gives us a straight horizontal line across the entire screen. Type:

```
10 INPUT " VERTICAL ADDRESS (0 TO 47)" ;Y

20 CLS

30 FOR X = 0 TO 127

40  SET (X,Y)

50 NEXT X

99 GOTO 99
```

and RUN several times.

We can just as easily create a straight vertical line. Try this.

```
10 INPUT "HORIZONTAL ADDRESS (0 TO 127)";X

20 CLS

30 FOR Y = 0 TO 47

40 SET (X, Y)

50 NEXT Y

99 GOTO 99
```

And RUN this a number of times.

Now, let's see if you can modify this last program so we can INPUT both the starting vertical address and the length (in blocks) of the line.

Of course you haven't forgotten how to do that have you! Type ERASE . . . no, no, no! Type NEW or just N. (Don't forget the period when you use abbreviations.)

```
12 IN. "THE STARTING VERTICAL ADDRESS #(Ø TO 47) IS";V

14 IN. "HOW MANY VERTICAL BLOCKS DO YOU WISH TO FILL"; A

16 IF V+1<48 GOTO 2Ø

18 PRINT "TOO MANY VERTICAL BLOCKS. WOULD WRAP-AROUND!"

19 END

3Ø FOR Y = V TO V+A
```

Now that we can draw straight lines, we can form figures — like squares and rectangles. This program forms a rectangle. After NEW , type:

```
1Ø INPUT "HORIZONTAL STARTING POINT (Ø TO 127)";X

2Ø INPUT "VERTICAL STARTING POINT (Ø TO 47)";Y

3Ø INPUT "LENGTH OF EACH SIDE (IN BLOCKS) -- Ø TO 47)";K

4Ø CLS

5Ø FOR L = X TO X+K

6Ø  SET (L,Y)

7Ø  SET (L,Y+K)

8Ø NEXT L

9Ø FOR M = Y TO Y+K

1ØØ SET (X,M)

11Ø  SET (X+K,M)

12Ø NEXT M

999 GOTO 999
```

and RUN.

Since our building blocks are not square, but 2 by 8 rectangles, we always get a rectangle. How can we change the program to always form a square?

EXERCISE 20-2: Modify the resident program so it always draws a square (on the inside).

Press on . . .

A Little Diversion (Is there no end to all the tricks we can do?!)

All our graphics work to this time has been done by drawing white lines on a darkened screen. We can do just the reverse by painting the screen white first, then darkening the desired areas with RESET . This program for example, draws a black horizontal line on a white background. Type:

```
10 IN. "VERTICAL POSITION (0 TO 47)";Y
20 CLS
30 FOR X = 0 TO 127
40  FOR J = 0 TO 47
50  SET(X,J)
60  NEXT J
70 NEXT X
80 FOR X = 0 TO 127
90   RESET(X,Y)
100 NEXT X
999 G.999
```

and RUN.

If you're interested, go back and try similar easy modifications to other demonstration programs and have some fun with these reverse (or "negative") displays

We can draw other straight (more or less) lines by just changing both X and Y addresses of SET in the FOR-NEXT loop. Try this next program to draw a diagonal line.

```
10 INPUT "HORIZONTAL STARTING POINT (0 TO 127)";X
20 INPUT "VERTICAL STARTING POINT (0 TO 47)";Y
```

Whew! That's not really very easy, but with some careful study (remembering that we have to account for the width of the blocks) it falls into line. Try your own approach a few times before going back to Part B to look at our suggestion. Don't cheat now!

You may want to come back later for some heavier study.

```
30 INPUT "DIAGONAL LENGTH";K
40 CLS
50 FOR L = 0 TO K
60  SET (X+L,Y+L)
70 NEXT L
99 GOTO 99
```

Once we have the diagonal line, we can form a right triangle by adding:

```
70 SET(X,Y+L)
80 SET(X+L,Y+K)
90 NEXT L
```

or

```
70 SET(X+K,Y+L)
80 SET(X+L,Y)
90 NEXT L
```

Try them both. What is the difference in the displays?

Answer: They are inverted, mirror-images of each other.

Broken Lines

In every graphics program we have used, we could have made the lines "broken" by introducing a STEP other than "1" in the FOR-NEXT loops. For example, we can get a broken horizontal line with:

```
10 INPUT "VERTICAL ADDRESS (0 TO 47)";Y

20 INPUT "STEP SIZE";S

30 CLS

40 FOR X = 0 TO 127 STEP S

50  SET (X,Y)

60 NEXT X

99 GOTO 99
```

RUN this program with various values of S. Note that as you increase S, the line is drawn much faster (since the Computer has less work to do). In fact, for S=10 or more, you can hardly see the line being drawn. This is how a TV picture is created — since it too is drawn one unit at a time (but so fast you don't notice the "drawing time").

Change the program as follows:

```
55 RESET (X,Y-1)

70 Y = Y+1

80 GOTO 40
```

If S is small, you can see the lines being formed and cleared. But if S is fairly large (try 10), the line seems to move in somewhat "old-time movie" fashion. This is the way the illusion of motion is created on a TV set and in some of the popular video games.

Now try this next program (clear out the old one). It paints a dot on the screen and moves it up and down.

```
10 INPUT "HORIZONTAL STARTING POINT (0 TO 127)";X

20 INPUT "VERTICAL STARTING POINT (0 TO 47)";Y

30 CLS

40 RESET (X,Y-1)

50 SET (X,Y)
```

```
60  Y=Y+1

70  GOTO 40

99  GOTO 99
```

The RESET command simply follows along behind and erases the dot from the last SET. What happens if you omit RESET? When you try this, remember to change line 7∅ to GOTO 5∅.

Details . . . Details

One minor problem . . . RESET and SET don't work with negative coordinates. Take a look at line 4∅ —

```
40 RESET (X,Y-1)
```

— if you INPUT Y equal to ∅, then the Y address really becomes Y—1 . . . —1. A no-no!

We can get around this pesky little detail by changing line 4∅ to:

```
40  RESET (X,Y+47)
```

Why does this work?

Hint: Wrap-around?

Back to the Good Stuff

We can just as easily move a point to the right with:

```
10  INPUT " HORIZONTAL  STARTING  POINT (90 TO 127)" ;X

20  INPUT "VERTICAL STARTING POINT (0 TO 47)";Y

30  CLS

40  RESET (X+127,Y)

50  SET (X,Y)

60  X = X+1

70  GOTO 40

99  GOTO 99
```

Note that we have used the same trick as above (under Details) to avoid negative values of Y.

What happens if you change line 4∅ to

```
4Ø RESET (X+126,Y)
```

and RUN?

Then;

```
4Ø RESET(X+125,Y)
```

And RUN.

Don't just sit there — try them! See what you almost missed?

EXERCISE 20-3: Change the last two programs so that they move the dot up and to the left respectively.

Now, let's have the dot move down until it strikes a barrier. The program is:

```
1Ø INPUT "HORIZONTAL STARTING POINT (Ø TO 127)";X
2Ø INPUT "VERTICAL STARTING POINT (Ø TO 47)";Y
3Ø INPUT " LOWER BARRIER" ;K
4Ø CLS
5Ø FOR M=Ø TO 127
6Ø  SET (M,K)
7Ø NEXT M
8Ø RESET (X,Y+47)
9Ø SET(X,Y)
1ØØ Y=Y+1
```

```
110  IF  Y<48  THEN  130

120  Y=Y-48

130  IF  Y<>KTHEN  80

999  GOTO 999
```

The dot appears to strike the barrier and stick to it.

Now let's have the dot start in the middle and ricochet from both the top and the bottom:

```
10 CLS

20 FOR M=0 TO 127

30  SET (M,0)

40  SET (M,47)

50 NEXT M

60 Y=14

70 D=1

80 RESET (64,Y+48-D)

90 SET(64,Y)

100 Y=Y+D

110 IF Y=48 THEN 130

120 IF Y<>-1 THEN 80

130 Y=Y-2*D

140 D=-D

150 GOTO 90

999 GOTO 999
```

The change in direction of the moving dot is caused by line 140 D=-D. Note that we must be careful not to accidentally erase part of the boundary. To do this, we not only move the dot back 2 steps with line 130 (after moving it forward 1 in line 100) but we also return to the SET in 90, rather than to RESET in 80. Tricky, tricky. You can kill the whole day messing around with this silly bouncing ball. Rather good resilience, eh?

Real Moving Pictures

We can draw whatever figures we like. Let's try a stick man. First, his legs:

```
10 CLS
20 X=64
30 FOR K=0 TO 7
40  SET (X+K,40+K)
50  SET (X-K,40+K)
60 NEXT K
999 GOTO 999
```

and RUN.

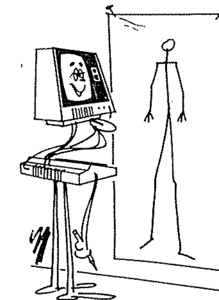
Then add his body and arms:

```
70  FOR K=0 TO 5
80   SET(X+K,34+K)
90   SET(X,34+K)
100  SET(X-K,34+K)
110 NEXT K
```

and RUN.

And finally his head:

```
120 SET(X,32)
```



"I'M A LEG MAN, MYSELF."

```
130  SET(X+1,33)

140  SET(X-1,33)
```

and RUN.

Now let's try and move him to the right. Add

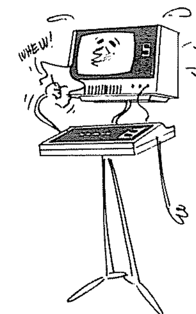
```
45 RESET(X+K-1,40+K)

55 RESET(X-K-1,40+K)

85 RESET(X+K-1,34+K)

95 RESET(X-1,34+K)

105  RESET(X-K-1,34+K)

125  RESET(X-1,32)

135  RESET(X,33)

145  RESET(X-2,33)

150  X=X+1

160  GOTO 30
```

and RUN.

Sure moves funny, doesn't he? Well, I'm no animator either, but I'm sure you're beginning to get the idea.

This has been one long and active Chapter . . . and to think all this with only the SET and RESET statements. Think of the good things to follow with two more commands! And by simply exchanging RESET for SET, in many cases we could have drawn the same pictures, with dark on a light background instead of light on dark. You might want to give that a try.

OK, so I'm no artist . . .!!



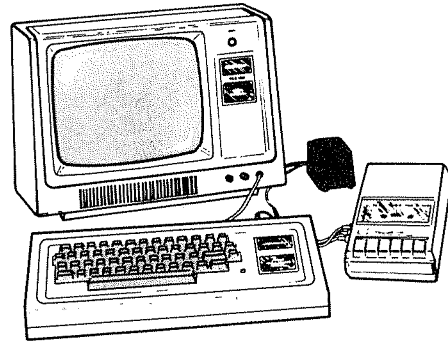
Because the ideas come so fast in the area of graphics, we have deliberately chosen to show you a lot of examples without getting bogged down in detailed explanations of how each one works. There is no substitute for lots of experimenting with graphics, and you now know the basics. Put in your time, study the examples, and soon you can apply for membership in the artists' guild.

Learned in Chapter 20

Statements

SET

RESET



Notes:

Chapter 21

Numeric Arrays

We know that we have the 26 letters of the alphabet available to use as variables. We've also discovered that very few of our programs have required anywhere near that many variables. There are times, however, when we wish to have more variables available — sometimes **hundreds** or even **thousands** of them, to use as names for many different pieces of data we are storing and want to "retrieve" easily.

The way out of this little dilemma is with an *array*. Array is just another word for "lineup", "arrangement" or "series of things". Let's say we're talking about a collection, arrangement or lineup (array) of a number of autos, all of which have different license plates (address numbers).

To be specific, we have 1Ø cars lined up, as in an *array*. They are all the same except for their engine size — and each has a different license plate number. Let's say the license plate numbers are from 1 to 1Ø, and we want to use the Computer to quickly spit out the engine size when we identify a car by license number. This might not seem like a real heavyweight problem — but, as before, we discover the full potential of these things by learning them little steps at a time.

Let's assume the license number and engine sizes are as follows:

LICENSE #	ENGINE (cubic inches)
1	3ØØ
2	2ØØ
3	5ØØ
4	3ØØ
5	2ØØ
6	3ØØ
7	4ØØ
8	4ØØ
9	3ØØ
10	5ØØ

Numeric Array? . . . *must be some kind of new math weapon . . .?*

Now, we could give each of these cars a different letter name, using the variables A through J, but what a waste — and what will we do when we have a thousand cars, not just ten?

Your TRS-80 LEVEL I Interpreter provides for a single array, and it is called "A". This is not the same as the alphabet variable "A", and it is not the same as the "A" used in the string variable A$. It is a third and totally separate "A". You will recognize it as A-sub(something). We will name the cars A(1) through A(1Ø), pronounced *A sub 1* through *A sub 1Ø*. Get the idea?

What's that – you're not sure you believe that there can be three separate and different storage places for these "A" items? OK, try it — type:

```
A = 12
A$ = (YOUR NAME)
A(1) = 999
```

then type:

```
PRINT A, A$, A(1)
```

. . . *NOW* what do you think? Did that make you a believer??

Next, let's store the car engine sizes in a line or two of DATA statements.

Type in:

```
100 DATA 300,200,500,300,200

110 DATA 300,400,400,300,500
```

Notice how careful we are to keep the DATA elements in order, from 1 to 1Ø, so the first car's engine is found in the first DATA Location and the 1Øth one's in the last location.

Now we have to "spin up" an array inside the Computer's memory to make these data elements **immediately addressable**. Think how difficult it would be to try to address the 7th engine (or the 7 thousandth!) for example, using only what we've learned so far. It **can** be done using only DATA, READ and RESTORE statements but it's very messy and slow.

Big words meaning "so we can find a car fast!"

The easy way to create the array is as follows . . . Type in:

```
50 FOR L = 1 TO 10

55  READ A(L)

60 NEXT L
```

. . . and RUN.

Nothing happen? Yes, it did. RUN again and note that **something** happened because it took a little time for READY to return. We simply didn't display what did happen.

We obviously used a FOR-NEXT loop to READ 1Ø DATA elements, and named those elements (or "cells") in which they're stored, A(1) through A(1Ø). Let's see if we can PRINT out the values in those array elements.

Type:

```
200 FOR N = 1 TO 10

210  PRINT A(N)

220 NEXT N
```

. . . and RUN.

Aha! It works, but how? We read the DATA elements into an array called A(L), but printed them out of an array called A(N). What gives? Oh, nothing, really. The array's **name** is "A". The location of each data element within that array is identified by the number which we place inside the parentheses. We can bring that number to inside the parentheses by using any of our 26 letter variables, and can even do some simple arithmetic inside those parentheses if we wish.

Remember, though, there is only one array, and its name is "A". Its elements are numbered, and called A-sub (number).

Some pure mathematicians might insist on calling A(X) — A "OF" X. Who needs that added confusion? Best that you know, just in case.

Let's work some more on the program.

Type:

```
170 PRINT

180 PRINT "LICENSE #","ENGINE SIZE"

210 PRINT N,A(N)
```

. . . and RUN .

Now that's more like it. We have every license number, every engine size, and are not "using up" any of the 26 alphabetic variables. Having demonstrated that point, erase lines 2ØØ, 21Ø and 22Ø, and type:

```
10 IN. "WHICH CAR'S ENGINE SIZE DO YOU WANT TO KNOW";W

210 PRINT W,A(W)
```

. . . and RUN.

Get the idea? Can you see the beginning of a simple inventory system for a small business?

Let's go one small step (for mankind) further. Suppose you know the color of each of the 1Ø cars, and for simplicity, suppose they are coded 1, 2, 3 and 4. We might then have a master chart that looks like this:

LICENSE #	ENGINE SIZE	COLOR CODE
1	3ØØ	3
2	2ØØ	1
3	5ØØ	4
4	3ØØ	3
5	2ØØ	2
6	3ØØ	4
7	4ØØ	3
8	4ØØ	2
9	3ØØ	1
1Ø	5ØØ	3

In the language of professional computer types, this is called a *matrix*. A *matrix* is just an array that has more than one dimension. (Our first array had the dimension of 1 by 10.) This array has a horizontal dimension of 2 and a vertical dimension of 10. If you wanted to be terribly inefficient about the matter, you **could** say that this is a 3 by 10 array, counting the license number. If so, then our original one would have been a 2 by 10 array — but who needs it? As long as we keep our license numbers in a simple 1 to 10 FOR-NEXT loop, and our DATA in proper sequence, we can keep our arrays simpler and easier to handle.

How then can we handle this 2 by 1Ø *matrix?* We have already used up our A array elements numbered 1 through 1Ø. *Oh, you want to know how many array elements we have to work with?* Very good! What was your name again? *(Let me mark that down.)*

Assuming you left our last program untouched, type:

"ENGINE SIZE? WHAT ENGINE?"

You might want to think of a matrix as a chart with a certain number of columns of information. First you set up the chart, then how many columns of info can you get in . . .?

```
PRINT MEM
```

and you will get a return of about

```
3370
```

We have 3370 memory cells left unusued. Again using the calculator mode, type:

```
PRINT 3370/4 -1
```

and we get

```
841.5
```

Thus, by dividing the remaining memory by 4, and subtracting 1, we found there is room for 841 more array elements. Lots of room left. We never had it so good. Each array element occupies 4 memory locations, whereas each letter variable requires only 1.

Well, with memory to burn, let's just arbitrarily assign array locations 101 through 110 to hold the color code. We also have to put the color code info in the program using a DATA statement. From the table, type:

```
300 DATA 3,1,4,3,2,4,3,2,1,3
```

and

```
80 FOR S = 101 TO 110
85 READ A(S)
90 NEXT S
```

. . . to load the color code DATA into the array. The array element numbers 11 through 100 are not used, nor are those from 111 to the end of memory, since they have not been formally assigned any values.

Now we need to find some way to display all this good information. Change these lines:

```
10 IN. "WHICH CAR'S ENGINE & COLOR DO YOU WANT TO KNOW";W
180 P. "LICENSE #", "ENGINE SIZE", "COLOR CODE"
210 PRINT W,A(W),A(W+100)
```

That's for 4K of memory. With more memory this number would be higher.

The 841.5 figure assumes you have only 4K of memory. With more memory, the number would be correspondingly higher.

You might stash that bit of information away for safekeeping. In a long program where memory might get scarce you may want to refer to it.

. . . then RUN .

Check your answers against the 2 dimensional matrix chart we gave you earlier. They should agree.

Let your imagination go. Can you envision entire charts and tables stored in this manner? Entire inventory lists? How about trying to **find** a car which has a certain size engine AND a certain color? Hmmm. We will come back to the Logic needed for that last one in a later chapter.

EXERCISE 21–1: Assume that your inventory of 1Ø cars includes 3 different body styles, coded 1Ø, 2Ø and 3Ø, as follows:

LICENSE #	BODY STYLE
1	2Ø
2	2Ø
3	1Ø
4	2Ø
5	3Ø
6	2Ø
7	3Ø
8	1Ø
9	2Ø
1Ø	2Ø

Modify the resident program to print the body style information along with the rest when the car is identified by license number.

A Smith & Wesson Beats 4 Aces

If we want to create a computerized card game (they make good examples to show so many things), how can we set it up so we draw the 52 or so (watch the dealer at all times) cards in a totally random way? **Answer**: Spin up the deck into a single-dimension array, pick array elements using a random number generator, as each card is "drawn", set its array

element value equal to zero, then test each card drawn to be sure it isn't zero. Now that is *really* simple!

We will now, a step at a time, write a program which will draw, at random, all 52 cards numbered from 1 through 52, and print the card numbers on the screen as they are drawn. No card will be drawn more than once. When all cards have been drawn, it will print "END OF DECK."

You do a step first, then check against my example. Then change yours to match mine — otherwise we might not end up at the same place at the same time.

Step 1: Spin up all 52 cards into an array.

```
30 FOR C=1 TO 52:READ A(C):NEXT C

50 DATA 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20

55 DATA 21,22,23,24,25,26,27,28,29,30,31,32,33,34,35

60 DATA 36,37,38,39,40,41,42,43,44,45,46,47,48,49,50,51,52
```

At this point, all you can tell when RUNning is that it is taking some processing time since the READY doesn't come back right away.

Shhhh! I know there's an easier way to program this special case, but it doesn't teach what's needed.

Step 2: Draw 52 cards at random, printing their values.

```
90 FOR N = 1 TO 52
100 V = RND(52)
110 PRINT A(V);
120 NEXT N
```

True, 52 card values are printed on the screen, but if you look carefully, the same number appears more than once. This means that some "cells" are not being READ and some READ more than once.

Step 3: When a card is drawn, set its array address equal to zero. Test each card drawn to be sure it is not Ø. When 52 cards have been drawn and printed, type END OF DECK.

```
90 P = 52

105 IF A(V) = 0 G.100

120 A( V) = 0

130 P = P - 1

140 IF P<> 0 GOTO 100

150 PRINT "END OF DECK!"
```

Line 12Ø sets the value in cell A(V) equal to zero only if line 1Ø5 finds it NOT equal to zero already, letting the program pointer fall through.

When a "fall through" occurs:

1. The card's value is printed (line 11Ø)
2. The number stored in that cell is set to zero (line 12Ø)
3. Line 13Ø counts down the number of cards printed. Line 9Ø initialized the number of prints at 52.
4. The number of prints is tested (line 14Ø). When there are no more prints to go, END OF DECK! is printed (line 15Ø).

Pretty slick — and you don't have to watch the dealer (just the programmer).

But how do you really know that every card has been dealt? Write a quick addition to the program to "interrogate" each array cell and print its contents.

```
200 FOR T = 1 TO 52

210 PRINT A(T);

220 NEXT T
```

RUN . . . and every cell comes up zero. If you don't really trust all this, change line 9Ø to read:

```
9Ø P = 5Ø
```

RUN , and see what happens.

AHA! It flushed out those 2 cards in the sleeve, didn't it.

Reinitialize P at 52, eliminate your test program lines 2ØØ, 21Ø and 22Ø and you end up with a good card-drawing routine. You might want to clean it up to your satisfaction and save it on tape for future projects.

Question: Why does the printing of card numbers slow down to a near halt as those last few cards are being drawn. Is the dealer reluctant?

Answer: The random number generator has to keep drawing numbers until it hits one that is the array address of an element which has not been set to zero. Near the end of the deck, almost all elements have been set to zero. The random number generator has to keep drawing numbers as fast as it can to find a "live" one.

Look again at the card numbers printed. There will not be any duplication. No stray aces.

There's More?

In the unlikely event you have a program which takes all 26 letter variables, you can use array locations to serve as numeric variables. Remember, however, each one requires 4 times as much memory space as a simple letter of the alphabet. Clear out the memory. Then type:

```
1Ø A(870) = 3

2Ø A(871) = 4

3Ø Z = A(87Ø) * A(871)

4Ø PRINT A(870),A(871),Z
```

. . . and RUN.

The answers of course are:

```
3                    4                    12
```

Try typing

```
10 A(1000) = 3
```

. . . and RUN.

Why does it blow up? What does the SORRY mean? A check of the unused memory would have immediately told us that the largest usable array element number was about 875. Try

```
P.M./4 -1
```

Again, 875 is the result when you only have 4K of memory. More memory will allow you a larger array.

EXERCISE 21-2: Study the User programs in Part C to better understand the use of arrays for storage and access purposes. Time spent studying programs written by others is wisely invested.

Learned in Chapter 21

Miscellaneous
Arrays

Chapter 22

Advanced Graphics

Remember the bouncing dot? Wouldn't it be nifty if we could get the screen to say "PING" each time the dot bounced off the barrier? Well *I* think it would be nifty, so we're going to do it. But first . . .

We learned all about SET and RESET earlier. Now we will learn about

PRINT AT — a special type of PRINT statement especially useful in graphics, and

POINT(X,Y) — a quite unrelated statement which allows us to look at any of the 6144 graphic block locations and get an answer to the question "which ones are ON and which are OFF?" A super powerful statement, and it's even useful!

I thought you printed ON, not printed AT

Learn something new every day. The PRINT AT (also written PRINTAT and P.AT) statement allows us to **begin** printing starting at a location number. Example, type:

```
10 CLS

50 PRINT AT 200, "HELLO THERE 200, WHEREVER YOU ARE."
```

. . . and RUN.

Where is 200? Back to the Radio Shack graphics layout chart (Video Display Worksheet). If you don't have one handy, there's one back in Chapter 20.

With the aid of an ordinary household electron microscope, the words "PRINT AT" are clearly seen on the upper left hand corner of the sheet. Also, an arrow pointing to a set of numbers. Further scrutiny discloses a tiny "X", obviously referring to the address numbers on the "X" Street — and a tiny "Y" pointing to the "Y" numbers for "Y" Avenue. A truly astute researcher will also see the "TAB" numbers — all 64 of them (starting with 0).

The PRINT AT numbers start at 0 and go through 63 — in the **first line**. They then pick up on the second line with #64 and continue through #127. The third line starts with #128, etc. The PRINT AT divisions are really the same as those for TAB except PRINT AT does not start over again with zero on the second line. It keeps going right on through PRINT posi-

Pardon our attempt at humor . . . poking a little fun at ourselves here. .!

tion #1023. This perhaps strange sort of numbering is not so strange at all when you consider the problems we had very early in the graphics game with the fool carriage return scrolling our light right off the screen. The PRINT AT statement does not trigger a scroll after it has done its printing, *EXCEPT IN THE LAST LINE*, between print positions #960 and #1023. Further, P.AT can **directly** address any of the 1023 printing locations (not light block locations — they are very different). Trailing semicolons are needed only after statements printed on that last or bottom line of the video "page".

Remember what scrolling is? An upward line roll.

You will soon see how valuable all this is.

Oh, It's That Time Already?

Let's create a 24-hour clock. (Why not . . . sounds like more fun than digging through all this obscure print statement logic.) Type:

```
10 CLS
20 PRINT AT 407, "H   M   S"
30  FOR H = 0 TO 23
40   FOR M = 0 TO 59
50    FOR S = 0 TO 59
60     PRINT AT 470, H; ":";M; ":"; S
70      FOR N = 1 TO 500: NEXT N
80    NEXT S
90   NEXT M
100 NEXT H
```

and RUN.

Nothing to it. *Ahem!*

"Hello? Bureau of Standards?"

Of course the accuracy of this timer depends on how closely you calibrate it. We know that the TRS-80 with LEVEL I BASIC will execute somewhere around 500 simple FOR-NEXT loops per second when written as shown in line 70 — a multiple statement line. If you really

get carried away with this program, you will want to calibrate it with a precision-type timepiece (increasing or decreasing the "5ØØ" figure as needed). Over the short run, this is quite a good timer. Note that we are not triggering this with the 6Ø Hz line frequency, but relying solely on the amount of time required to execute FOR-NEXT loops.

Oh, Yes . . . The PRINT AT . . .

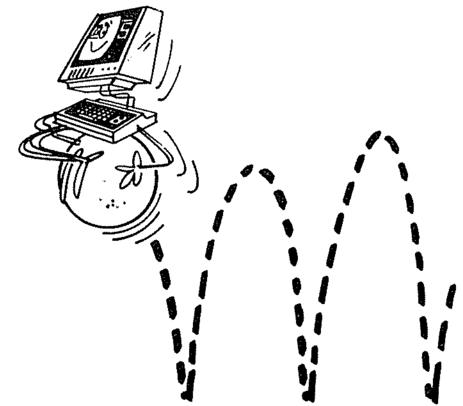
Anyway — let's not lose sight of the forest for the trees (or something equally trite). The purpose of this little program is to demonstrate the PRINT AT statement. We used it twice. By carefully squinting at the layout chart you can find address #4Ø7, with #47Ø right below it. With blazing speed, the HMS (no, no, not Her Majesty's Service — it stands for Hours, Minutes and Seconds), are printed — and the HM&S updated each second.

NOTE: No carriage-return-suppressing semicolons follow the PRINT AT statements — since they are not on the bottom print line.

For the real clock nut, see Part C for an operational clock program. It only needs your closer calibration to be an acceptable sundial. Most expensive clock in the house!

That's How the Ball Bounces

Meanwhile, back with the bouncing ball. Let's reload the program from the first graphics chapter. It reads:



```
 1Ø CLS
 2Ø FOR M=Ø TO 127
 3Ø  SET (M,Ø)
 4Ø  SET (M,47)
 5Ø NEXT M
 6Ø Y = 14
 7Ø D = 1
 8Ø RESET (64,Y+48-D)
 9Ø SET (64,Y)
1ØØ Y = Y+D
11Ø IF Y=48 THEN 13Ø
```

```
120  IF  Y<>-1  THEN  80
130  Y  =Y-2*D
140  D=-D
150  GOTO  90
999  GOTO  999
```

Since we did not explain in detail how that fairly simple program worked, take time now to see if you can follow it through. When you have it figured out, tackle this exercise:

EXERCISE 22-1: Using PRINT AT statement(s), cause the word "PING" to appear near the ball each time it bounces off either the top or bottom boundary.

Isn't it amazing how close we are to getting to some of the actual video games that are all the rage? — and yet it's really so simple and logical.

Merely for Display Purposes

A good way to get a feel for PRINT AT (or any feature) is to look at a fairly simple program which illustrates its use. This program lays out a graph format on the screen. What you do with it beyond that point depends on your own needs and interests, but it is worth entering, studying and getting a feel for its use. Type:

```
10  CLS
20  K  =  900
30  F.X=1TO59
40   P.ATK+X,"  ." ;
50  N.X:  K  =  964
60  FOR  Y  =  0  TO  13
70   P.ATY*64+5,"  ." ;
80  NEXT  Y
```

One noteworthy procedure in this program is the use of trailing semicolons after PRINT AT statements. The reason, again, is that the printing is taking place in the last line on the screen so the carriage return would activate a scroll. We therefore have to suppress the carriage return with the semicolon.

```
100 P.AT20,"G R A P H  H E A D I N G"
150 F.N = 0 TO 14
200  P.ATN*64,14-N;
250 N.N
300 F.X = 0 TO 5
310  P.AT K+10*X,X;
320 N.X
400 F.X=6TO56 STEP10
410  P.AT K+X, ":";
420 N.X
999 G.999
```

What is the POINT of all this?

The POINT(X,Y) statement stands pretty much isolated from the other 3 graphics statements. It needs them, but they don't need it at all.

POINT(X,Y) interrogates (what a great technical word) that graphics point on the screen with the address of X,Y. If that point is lit, the POINT statement says "1". If it is dark, the POINT statement says "∅". That's really all there is to it. Of such great simplicity great power is derived.

Interrogates — just asks a question; but it's in logic form . . . true or yes = 1 and false or no gives a ∅.

Let's give POINT a little exercise before looking closer. Since it also works in the calculator mode, type

```
PRINT POINT(30,30)
```

Since we had not lit 3∅,3∅ the answer came back with ∅. It also can be abbreviated. Type:

```
P.P.(30,30)
```

Same thing: ∅

Tho they can both be abbreviated P., the parenthesis following the second P. tells the Computer that it is POINT and not PRINT.

Let There Be Light

Let's light up a spot on the screen, then interrogate that point and see what happens. Type:

```
10 Y=1:N=0
20 IN. "DO YOU WISH TO LIGHT THE BLOCK (Y/N)";Q
30 CLS
40 IF Q = 0 GOTO 80
50 SET(75,20)
60 GOTO 100
80 RESET(75,20)
100 IF POINT(75,20) = 1 P.AT200,X;Y,"IS LIT"
200 IF POINT(75,20) = 0 P.AT200,X;Y,"IS DARK"
999 G. 999
```

Y and N stand for Yes and No. Your input can be either Y or Yes, N. or No.

And RUN several times. Answer either YES or NO, following the program action to see what is happening. Pretty simple isn't it? Really sort of a status-reporting system. Think what we could do if we set something like this up in 2 nested FOR-NEXT loops so we scanned the entire screen and got a status report on each point. *Hmmm.* Almost like a radar scan of the terrain. *Hmmm* some more.

2001 Here We Come

Snug up your seat belt, type and RUN the following program, then sit back and watch POINT in action. Study the display very carefully as it runs, looking for the many things that occur. This 3½ minute "moving picture" really tells all you need to know about the POINT statement.

```
10 REM * DEMONSTRATION OF GRAPHICS 'POINT' STATEMENT *
20 P=15:L=119
30 CLS
40 P.AT5,"THIS IS A DEMONSTRATION OF THE POINT STATEMENT---";
```

```
50 P.AT56, "X       Y";

100 F.I=1TO P:SET(RND(113),RND(45)+2):N.I

110 F.X=0 TO 111:F.Y=0 TO 47

120  IF POINT(X,Y) = 0 GOTO 160

130  P.AT L,X;:P.AT L+4,Y;

140  L=L+64

150  G.170

160  SET(X,Y):RESET(X,Y)

170 N.Y:N.X

180 P.AT5," THE COORDINATES OF THE GRAPHICS BLOCKS ARE >>-->>" ;

190 P.AT 0;

200 G.200
```

Vectoring in on Darth Vader's Death Star Fleet . . .

If that one didn't blow your mind, let's take the program apart a line at a time:

Line 10 is just the program identification note

Line 20 P is set at 15, the number of "targets" to be randomly placed on the screen by the FOR-NEXT loop and the random number generator in line 100. L is set at 119, the starting point for printing the coordinates and their headings in lines 130 and 140.

Line 30 clears the screen for action.

Line 40 and 50 use PRINT AT to print the heading.

Line 100 generates 15 addresses and SETs 15 lights.

Line 110 uses two nested FOR-NEXT loops to establish a "scanner", testing every graphics point on the screen.

Line 120 tests to see if the point being addressed is off. If so, the address printing and related incrementing of the next PRINT AT location in lines 130 and 140 are bypassed.

Line 130 prints the values of X and Y if the POINT test in line 120 "fell through", meaning that the point being tested was not off.

Line 140 increments the PRINT AT address for the next time line 130 prints the coordinates. By adding 64, the next printing will directly line up under the current heading and numbers. (See the layout chart if you can't follow it in your head.)

Line 15Ø is critical, since it jumps over a RESET. Without this jump, the light blocks would be erased when they are interrogated. (Try deleting the line and RUNning to see.)

Line 16Ø is just a foxy display trick. It causes a blinking light to "appear" to be scanning all points. Actually, line 16Ø is just turning blocks ON and OFF as the **Computer** interrogates them. Since we don't want to turn off blocks that are already supposed to be lit, we hop over this line when a real live block is hit. In reality then, this "roving eye" never actually "hits" an "ON" block.

Line 17Ø merely closes the FOR-NEXT loops started in line 11Ø

Line 18Ø replaces the heading that was erased by the POINT scanning process.

Line 19Ø simply moves the cursor (which can be a pesky light in graphics displays) back to the far upper-left corner, to get it out of the way. It could have been moved to any point, or just left alone.

Line 2ØØ is the locking loop used to keep READY and the prompt from goofing up the display.

Pretty simple when taken a line at a time, isn't it?

Oh yes, did you notice that the "moving dot" turned off the original heading? Did you also notice that it took two passes of the dot to equal the width of one printed character (which of course fits right in with what the layout sheet shows)?

The reason the heading was erased is that we deliberately chose not to protect it (like we protected the blocks) from the RESET in line 16Ø in order to make the point. You could write a little protective line if you wanted to (or reduce the vertical length of the scan to avoid it).

Alpha or Omega?

There you have it — a good running start into graphics. Go now to part C where you will find more ready-to-run programs. By giving them careful study, you will see the four graphics statements in use, plus most of the rest you have learned about BASIC programming. The possibilities from here are unlimited.

Learned in Chapter 22

Graphics Functions

POINT (X,Y)

PRINT AT

Chapter 23

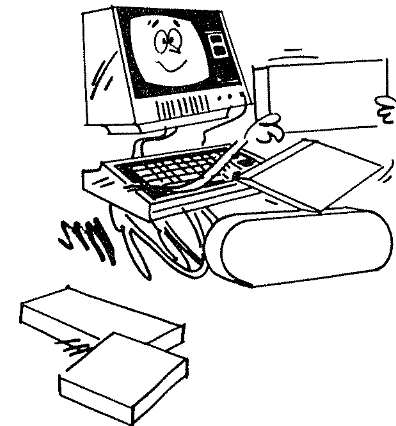
Flowcharting

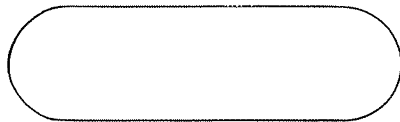
Most of the programs we wrote so far were simple; but, they met fairly simple needs. Suppose you want to write a program to play chess or bridge, evaluate complicated investment alternatives, keep records for a bowling league or a small business, or do stress calculations for a new building? How would you go about writing a complex program like that?

Answer: You break down the big program into a series of smaller programs. This is called **Modular Programming** and the individual programs are called **Modules.** But how are the modules related — and how do you write them anyway?

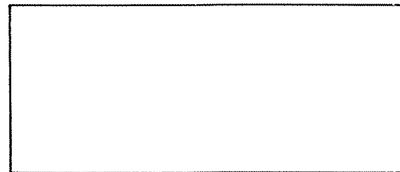
Module is just a 75-cent word for "section" or "building block"

One way to plan a program is to make a picture displaying its logic. Remember, a picture is worth a thousand words (or is it the other way around?). The picture that programmers use is called a **flowchart.** Flowcharts are so widely used that programmers have devised standard symbols. There are many specialized symbols in use, but we will examine only the most common ones.

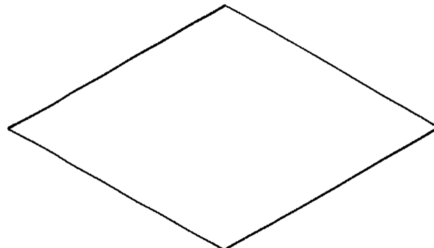




BEGIN or END



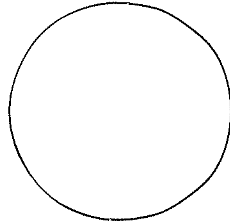
PROCESSING BLOCK (encloses something the computer does without making any decisions)



DECISION DIAMOND (it branches off in different directions, depending on the decision it makes.)

Each decision point asks a question such as "IS A LARGER THAN B?" or "HAVE ALL THE CARDS BEEN DEALT?" The different branches are marked by YES or NO.

Another useful symbol is:



CONTINUATION

The circle usually has a number inside it which corresponds to a number on another page if the flowchart is too large for a single sheet.

CONNECTOR ARROWS

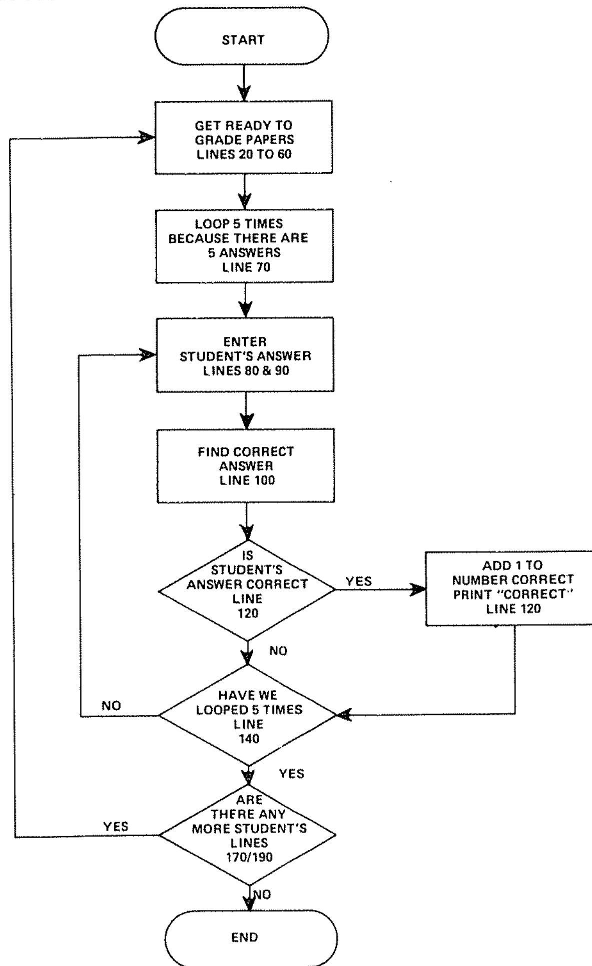
Arrows indicate the direction in which program execution proceeds.

Flowcharts are most helpful in designing programs when they are kept simple. A cluttered flowchart is hard to read and usually isn't much more helpful than an ordinary written program list. A good flowchart is also helpful for "documentation" to give you (or others) a picture of how the program works — for later on, when you've forgotten.

There are no hard-and-fast rules about what goes into a flowchart and what doesn't. A flowchart is supposed to help you . . . not be more work than it's worth. It helps you plan the logic of your program. When it stops helping and makes you feel like you're back in arts and crafts designing mosaics, then you've gone as far as the flowchart will take you (or more typically, you've passed its point of usefulness).

Let's look at some examples. Suppose we want to grade a 5-question test by comparing each of the students' answers with the correct answer. We will put the correct answers in a DATA statement in the program, enter a students' answers through the keyboard, compare (grade) them, then print the % of correct answers. This procedure will be repeated until all the students papers are graded.

The flowchart might look like this:



This flowchart has three decision diamonds. In the first, the Computer determines if an answer is correct. In the second, the Computer determines if all the questions in a single student's paper have been graded. The third one terminates execution when all the tests have been graded.

EXERCISE 23-1: Using the flowchart as a guide, write a program that grades a test having five questions.

For more complicated problems, you may want to subdivide the flochart into larger modules. A **master flowchart** will then show the relationship between the flowcharts of individual programs.

For example, let's say you want to write a program that calculates the return on various investments. The options might be:

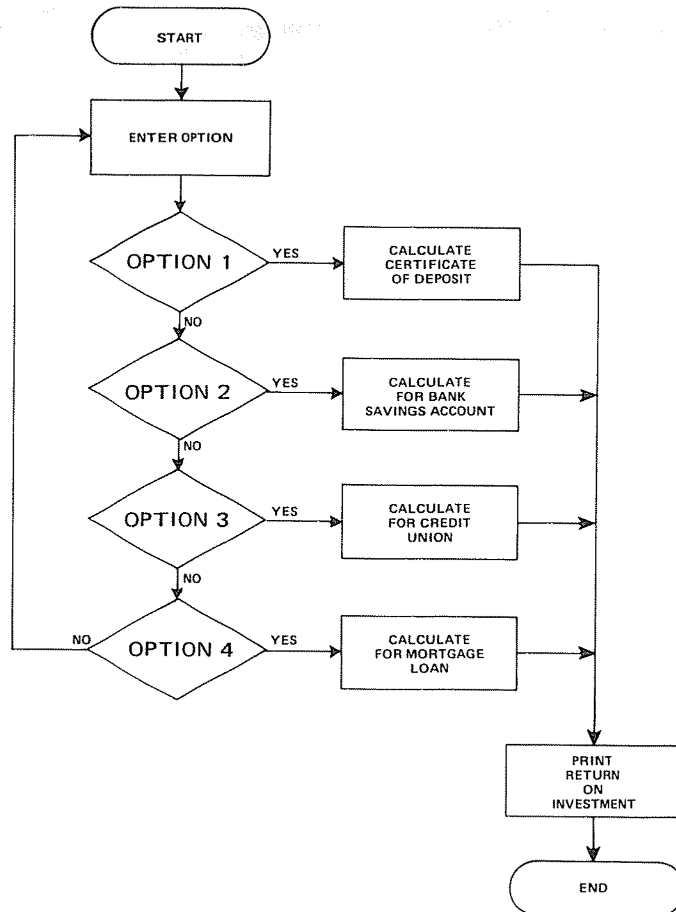
1 — CERTIFICATE OF DEPOSIT

2 — BANK SAVINGS ACCOUNT

3 — CREDIT UNION

4 — MORTGAGE LOAN

The main (or Control) program will select one of these 4 options using an input question, execute the correct subprogram, and print the answer. Its flowchart might be:



We could now flowchart each of the individual programs in the blocks separately. The Certificate Of Deposit program would, for example, have to contain the rate of return, size of deposit, and number of years in which the certificate matures. The order in which that program inputs data and performs the calculations would be specified in its own flowchart.

EXERCISE 23-2: Write the master program as flowcharted, with a branch to a program to calculate the return on a Bank Savings account paying simple interest.

EXERCISE 23-3: Choose a program from an early Chapter and design your own flowchart.

Learned in Chapter 23

Miscellaneous

Flowcharting

Notes:

Chapter 24

AND & OR

In classical mathematics (fancy words for simple ideas) there exist what are known as the "logical AND" and the "logical OR".

So the One Cow Said to the Other Cow . . .

In Figure 1, if Gate A **AND** gate B **AND** gate C are open, the cow can move from Pasture # 1 to Pasture #2. If any gate is closed, the cow's path is blocked.

By the way, this cow's name is Bessie.

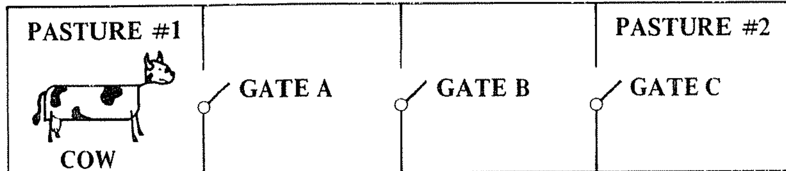


FIGURE 1

The principle is called "logical AND".

In Figure 2, if gate X **OR** gate Y **OR** gate Z are open, then old Bess can move from Pasture #3 to #4. That principle is called "logical OR". These ideas are both pretty logical. If the cow can figure them out surely you can!

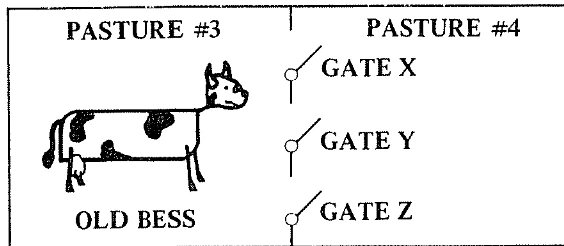
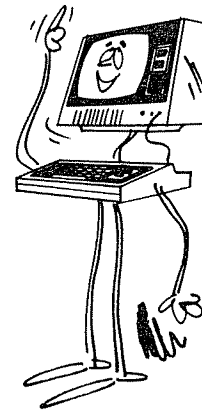


FIGURE 2



"BESSIE?-IT'S ONLY LOGICAL."

Grit Your Teeth and Prepare to Say AAAAAGH!!!!

Somewhere in the misty history of classical mathematics, a budding genius dedicated the symbol "X" to mean AND, and "+" to mean OR. Ordinary arithmetic of course uses "X" to mean multiply and "+" to mean add. To further confuse the matter, instead of "X", for computers we use "*" to mean multiply. Our logical AND symbol, therefore, is "*".

AAAAAAGH!!!!!

(I told you.)

Now don't forget . . . in "logical" computer work:

* means AND
+ means OR

Now For the Good News

Despite the frustration these so-called "logical" symbols inflict, using them is very simple. Type:

```
10 Y=1:N=0
20 IN. "IS GATE 'A' OPEN";A
30 IN. "IS GATE 'B' OPEN";B
40 IN. "IS GATE 'C' OPEN";C
50 PRINT
60 IF (A=1) * (B=1) * (C=1) THEN 100
70 P. "OLD BESSIE IS SECURE IN PASTURE #1."
80 END
100 P. "ALL GATES ARE OPEN. OLD BESSIE IS FREE TO ROAM."
```

Be sure to use the Shift-7 key to get the single quote mark.

Remember . . . we're using * here as the logical AND.

. . . and RUN. Answer (Y/N) the questions differently during RUNs to see how the logical AND works in line 60.

Where is the LOGIC in all this?

You should by now understand every line in the program except perhaps line 60.

Line 10 initializes the Y and N values at 1 and 0 respectively.

Lines 20, 30 and 40 input the gate positions as **open** (which we defined as equal to "1"), or **closed** (defined as "0"). We could have defined them the other way around in line 10 and rewritten line 60 to match, if we'd wanted to.

Line 60 is the key. It reads, literally, "If gate A is **open**, AND gate B is **open**, AND gate C is **open**, then go to line 100. If any one gate is closed, report that fact by defaulting to line 70."

Imagine how this simple logic could be used to create a super-simple "computer" consisting of only an electric switch on each gate — add a battery and put a light bulb in the farmer's house. The bulb could indicate whether the gates are all open. Such a "gate-checking" computer would have only three memory cells — the switches.

Hmm. It would do the job a lot cheaper than a TRS-80 . . . but would be awfully hard to play Blackjack with.

Back to the Subject

Look at line 6Ø very carefully. You have seen every symbol there before — but is there something different about how they are arranged? Hmmmm?

Ah yes — the parentheses. They are the tip-off. There has been (until now) no reason at all to enclose something like

```
A=1
```

in parentheses. When you come across a pair of parentheses enclosing an = sign, a ⟩, a ⟨ (or a combination of these), you know logical math is being used. (Whew — that's simple enough!) Having used the * (which you know means AND) now it will all make sense.

EXERCISE 24-1: Using the above program as a model, and the "OR logic" seen in Figure 2, write a program which will report Bessie's status as determined by the position of Gates X, Y and Z.

Teacher's Pet

Here is a simple program which uses > instead of the equals sign in a logical test. The student passes if he or she has a final grade over 6Ø **OR** a midterm grade over 7Ø **AND** a homework grade over 75. Enter the program, RUN it a few times, and see how efficiently the logical OR and logical AND tests work in the same program line (4Ø).

```
1Ø INPUT "FINAL GRADE" ;F

2Ø INPUT "MIDTERM GRADE" ;M

3Ø INPUT "HOMEWORK GRADE" ;H

4Ø IF (F>6Ø) + ((M>7Ø) * (H>75)) THEN 7Ø

5Ø PRINT "FAILED"

6Ø END

7Ø PRINT "PASSED"
```

Does this give you some idea of the power and convenience of logical math? The actual grade numbers could, of course, be set at any level.

Logical Variations

This next program example mixes equals signs, greater-than and less-than in the same program. It determines and reports whether the two numbers you input are both positive, both negative, or have different signs.

Analyze the program. Note the parentheses. They tell you to shift your thinking to "logical". Type it in and RUN.

```
1Ø INPUT "FIRST NUMBER IS" ;X

2Ø INPUT "SECOND NUMBER IS" ;Y

3Ø IF (X>=Ø) * (Y>=Ø) THEN 7Ø

4Ø IF (X<Ø) * (Y<Ø) THEN 9Ø

5Ø PRINT "OPPOSITE SIGNS"

6Ø END

7Ø PRINT "BOTH POSITIVE"

8Ø END

9Ø PRINT "BOTH NEGATIVE"
```

With Graphics Too, Yet

Yes, the logical symbols also work along with the graphics statements. See if you can figure out the surprise caused by the logical OR in line 4Ø. Type this program in, and RUN.

```
1Ø CLS

2Ø FOR X=Ø TO 127

3Ø  FOR Y=Ø TO 47

4Ø  IF (X>=64) * (Y>=24) THEN 6Ø

5Ø   SET (X,Y)

6Ø  NEXT Y

7Ø NEXT X
```

```
99 GOTO 99
```

What happens if you replace the * in line 40 with a +? After you think you have it figured out, do it and see the result.

Did you guess right???

Use [BREAK] to get out of the program's endless loop.

There's More?

Oh, yes — the only limit is your imagination. See how easily the logical notation makes the drawing of lines? Type and RUN:

```
10 CLS
20 FOR X=0 TO 127
30  FOR Y=0 TO 47
40  IF (X=64)+(Y=24) THEN 60
50   SET (X,Y)
60  NEXT Y
70 NEXT X
99 GOTO 99
```

What happens to the program if you replace + (OR) with * (AND)? Sketch your estimated result, then change line 40 and try it.

Hope you got it right. If not, it really sneaked up on you, didn't it!

Using the INT function we can create an elaborate checkerboard. The reasoning is:

In the horizontal dimension.

The INT(X/16)*16–X will equal 0 when X equals 0, 16, 32, 48, 64, 80, 96 and 112

In the vertical dimension

The INT(Y/6)*6–Y will equal 0 when Y equals 0, 6, 12, 18, 24, 30, 36 and 42.

Oh come on, it's very simple if you take the time and think it through!

Replace the old line 4Ø with

```
40 IF ((INT(X/16)*16-X)=0)+((INT(Y/6)*6-Y)=0) THEN 60.
```

and you will create an elaborate eight-by-eight checkerboard.

And on and on it goes . . .

And In Conclusion

The illogic of logical math is worth the hassle. As one last fun program, enter and RUN this "Midnight Inspection." Line 1ØØ checks each response for a NO answer (instead of a YES). Using logical OR, it branches to the "no-go" statement (line 130) if any one of the tests does not match the expectation.

```
10 CLS
20 Y=1:N=0
30 P. "ANSWER THESE QUESTIONS WITH 'YES' OR 'NO'.":P.
40 IN. "HAS THE CAT BEEN PUT OUT";A
50 IN. "IS THE PORCH LIGHT TURNED OFF";B
60 IN. "ARE ALL DOORS AND WINDOWS LOCKED";C
70 IN. "IS THE TELEVISION TURNED OFF";D
80 IN. "DID YOU TURN THE THERMOSTAT DOWN";E
90 P.:P.
100 IF (A=N)+(B=N)+(C=N)+(D=N)+(E=N) THEN 130
120 P."                GOODNIGHT":END
130 P. "SOMETHING HAS NOT BEEN DONE. DO NOT GO TO BED"
140 P. "UNTIL YOU FIND THE PROBLEM!"
150 GOTO 40
```

In most cases, AND and OR statements are interchangeable if other parts of a program are rewritten to accommodate the switch.

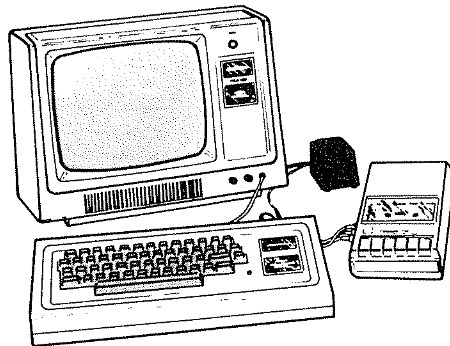
EXERCISE 24-2: Rewrite line 4Ø in the checkerboard program to produce a black-on-white checkerboard instead of white-on-black.

Learned in Chapter 24

Miscellaneous

* as a logical AND symbol

+ as a logical OR symbol



Notes:

Chapter 25

Advanced Subroutines

Back in Chapter 15, we touched on the subject of subroutines. We even "called" one, just to get the hang of it. But then we rewrote that subroutine as a part of our main program, and got the same results just as easily. So what's so special about subroutines? That's what this chapter is about.

To refresh your memory: A subroutine is a special kind of program which the Computer ignores until a GOSUB statement calls for it. After executing the subroutine, the Computer automatically RETURNs program control to a point right after the GOSUB statement. So no matter how many different times and places your program "branches" to a subroutine, program control always returns to the point where it left off.

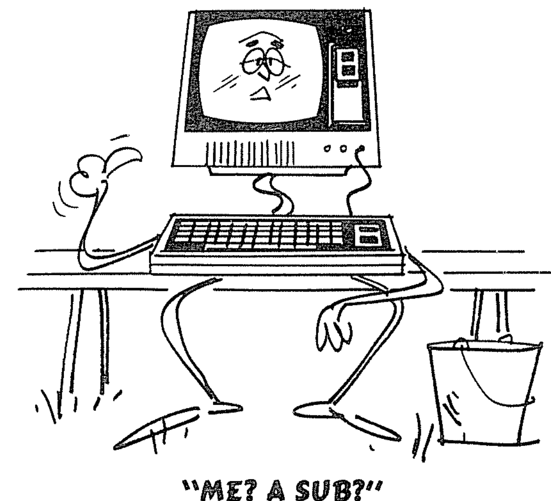
An important application of subroutines is the calling of special routines that allow the Computer to do things that it couldn't do otherwise. Take square roots, for example. Many larger forms of BASIC (like Radio Shack LEVEL II BASIC) will let you compute $\sqrt{X}$ simply by using the statement, Y = SQR(X). LEVEL I BASIC doesn't have this ability, so we need to add a fairly simple program to accomplish the same thing — a subroutine.

There are many other special routines that we can call to make the computer "educated beyond its intelligence". Most are very mathematical, and are only for rather special applications. But when they are needed, they are *badly* needed. Even if you don't think you'll have use of them, go through this lesson anyway. You'll probably find some special program in a magazine or elsewhere that you desperately want to run on your Computer — but it needs a trigonometric, logarithmic, or other higher-math function. You don't have to like or even understand these special routines to be able to use them like an expert. Give it a good shot.

Whatever became of good old Pythagoras – and who cares?

Remember the fun days in geometry and algebra class when you were engaged in such exciting things as trying to find the length of the hypotenuse of a right triangle when the lengths of the other two sides were known? Welcome back! The Pythagorean rule says, "The length of the hypotenuse is equal to the square root of the sum of the squares of the remaining two sides." No wonder you took ceramics instead. We know all about equations now, though, so we can state it much more simply:

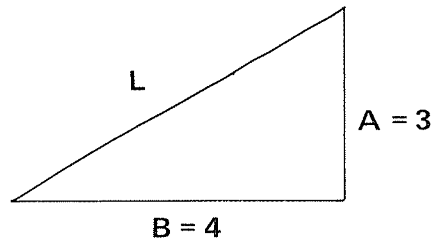
$$L = \sqrt{A^2 + B^2}$$



"ME? A SUB?"

Functions like SQR are called **intrinsic** (built-in) when they are available directly, without the need for calling a special subroutine. ABS, INT and RND are a few of the intrinsic functions available in LEVEL I BASIC.

where the triangle looks like this:



Okay. That's not too grim. So let's write a program to find the value of L, when A = 3 and B = 4. If we had the built-in square root function (which we don't), our program might look like this:

```
10 REM * SQUARE ROOT SOLUTION WITH SUBROUTINE *

20 IN."THE LENGTH OF SIDE A = ";A

30 IN."THE LENGTH OF SIDE B = ";B

40 L = SQR(A*A + B*B)

50 P. "A","B","L"

60 P. A,B,L
```

Now type in the program carefully and RUN.

Caarash! The

```
WHAT?

40 L=S?QR(A*A + B*B)
```

tells us the Computer does not recognize SQR. That means we'll have to call up the SQR subroutine from Appendix A to make a workable program.

See the list at the end of this Chapter to determine which functions are available as subroutines in Appendix A.

Turn to page 216 of Appendix A and find the Square Root Subroutine. There are three important things to look for when checking out any subroutine:

1. What is the input variable?
2. What is the output variable?

Whenever you come across a program with an intrinsic function you don't have, you'll go through this same procedure. First type in the subroutine, then make a few minor changes in the main program as described in the text.

3. What other variables are used by the subroutine for internal calculations? You have to have a high need for dangerous living to use those same variable names in the main part of the program. Either that, or know exactly what you're doing. Best to change the program to avoid re-using a subroutine's "internal variables".

Type in the square root subroutine **exactly** as it's listed. There is no room for error.

Now we need to interface it (that's high-powered computer jargon to impress your friends with) to the resident program (more jargon — get the idea?). In short, we have to make them match up.

Make these changes in the resident program:

```
40 X = A*A+B*B : GOSUB 30030

45 L = Y
```

and RUN.

If your program ran, there was a slight pause before the Computer came back with the answer:

```
A        B        L

 3        4        5
```

That's because the Computer has to do quite a bit of thinking to compute square roots. (See, computers aren't so smart after all!)

If the program didn't RUN, go over the main program and the subroutine very carefully. Did your GOSUB statement in Line 40 call 30030? And does your subroutine really begin at line 30030?

Here's why we changed Line 40 and added Line 45:

Line 40 has two separate statements. X=A*A+B*B gets our input variables X ready. GOSUB 30030 directs the Computer to the subroutine beginning at Line 30030. Lines 30010 and 30020 are remarks only. To speed things up, we skip over them. The Computer executes the instructions there until it hits the RETURN statement in Line 30080, which makes it return to the very next statement in our main program, Line 45.

Line 45 gives our hypotenuse L the value of the subroutine's output Y.

Line 30000 prevents the Computer from crashing into the subroutine after it completes your main program. Always insert this line when using subroutines.

We send control to 30030 rather than 30010 because 30010 and 30020 are the "non-working" parts of the program — they're just remarks for identification only.

We want the square root of the **entire expression** A*A+B*B, so we set it equal to X, which is the proper input for the subroutine.

CSAVE!

Before going on to the next section, you may want to save the square root subroutine on a cassette, to avoid the tedium of typing it in correctly again later. You can save all the subroutines separately, all together, or in various combinations. This will let you load just the ones you need for a given purpose.

From Square Roots to Circles (Well, Ovals Anyway!)

While you've got the square root subroutine loaded, we'll demonstrate how scientific subroutines can be put to some fairly entertaining uses. Type in the following program:

```
10 CLS
20 FOR R=2 TO 22 STEP 4
30 FOR A=-R TO R
40 X=R*R-A*A : GOSUB 30030 : Y=INT(Y-.5)
50 SET (A+60,23+Y)
60 SET (A+60,23-Y)
70 NEXT A
80 NEXT R
90 GOTO 90
```

And RUN.

Use [BREAK] to get out of the program.

If you entered the program correctly, the Computer will generate a series of concentric circles. The program uses the formula for finding the coordinates of a circle on a graph:

$$Y = \sqrt{R*R - X*X}$$

(Y is the Y-coordinate, X is the X-coordinate, and R is the radius.)

Actually they're ovals, because the graphics points are rectangular instead of square — it would take a slightly modified program to generate more perfect circles. Care to try?

You'll probably want to try "graphing" other curves using the subroutines, so we'll go into a little detail on how our concentric circles program works.

Line 1Ø gives us a nice clear screen to start with.
Line 2Ø sets up a loop which increments the Radius R from 2 to 22 in steps of 4.
Line 3Ø sets up a "nested" loop which increments the X-coordinate of our graph from −R to R.
Line 4Ø computes the Y-coordinate of our graph as a function of the radius R and the X-coordinate A. The square root subroutine is called.
Lines 5Ø and 6Ø center the circle on the Display and "draw it". Line 5Ø produces the lower half of the circle, and line 6Ø produces the upper half.
Lines 7Ø, 8Ø and 9Ø . . . you can figure them out for yourself.

From One Subroutine to Another

So far we've used GOSUB commands in the main program to call subroutines. Now let's be neighborly — and let one subroutine call on another.

Suppose we want to compute 3^{11} — that's 3 times itself 11 times. We can compute it directly as 3*3*3*3*3*3*3*3*3*3*3, right? But what about $3^{11.3}$ — that is, 3 to the 11.3 power? Simple multiplication isn't going to get us anywhere on this one. It looks like a job for Supersub!

That's our Exponential Subroutine, which actually calls on two other subroutines before it's through — one for logs, one for anti-logs. (These are **not** the opposing sides in a conservation dispute. They're extremely useful math functions.)

We're using natural logs and anti-logs, as opposed to common logs. Remember, this is a classy operation!

Supersub derives its number-crunching power from a rather complex-looking equation:

$$X^Y = e^{Y*\log X}$$

(You don't have to understand it, but it's nice to know it's there.)

All we have to do is provide the subroutine with the values for X and Y, in this case 3 and 11.3, and the rest is automatic. The subroutine goes and gets log 3, etc., and returns to our main program with the final answer.

Turn to page 217 of Appendix A and find the Exponentiation Subroutine.

Type it in slowly (as if we needed to say that) and carefully.

Now type in the following "demonstration program".

```
10 PRINT "SEEKING THE VALUE OF X TO THE Y POWER"

20 INPUT "X = ";X

30 INPUT "Y = ";Y

40 GOSUB 30120

50 PRINT "THE ANSWER IS";P

60 GOTO 10
```

And RUN. (Use [BREAK] when you want to get out of the program.)

Without trying to understand the mathematics behind it, let's trace the flow of the program control from main program to the various subroutines and back.

Lines 20 and 30 provide values for X and Y.
Line 40 transfers program control from the main program to the Exponentiation Subroutine.
Line 30140 calls the log subroutine to obtain log(X).
Line 30230 returns L = log(X) to the exponentiation subroutine. (Note that control passes to the statement immediately following the last GOSUB command — even though that statement is on the same line in this case.)
Line 30140 now calls the exponentiation subroutine to compute e^{Y*L}.
Finally, **Line 30150** adjusts the magnitude of P *(don't ask questions!)* and returns the computed value $P = X^Y$ to the main program, Line 50, for output.

Now let's go off on a tangent about Christmas trees.

Selecting a Christmas tree in the middle of a forest on a snowy evening in December can be a trying process. Especially when you're seeking a tree that's exactly 28 feet tall (the tree is going to be set up in a park downtown). You can climb up each tree, attach a 28-foot tape measure, climb back down and check to see if the tape just touches the ground — and

It's a long one — after you type it in and get it running properly, you'll want to save it on tape for later use.

Note that the input variable X is changed by the subroutine. Suppose we need this input value later in the program. We can't refer to this original value by calling it X, because X has take on a new value. The way around this common problem is to use a "dummy variable" to hold the original value of X. The dummy variable assignment must be before the entry to the subroutine:

```
35 S=X
```

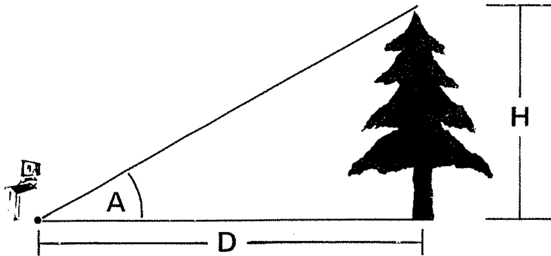
Adding this line to our resident program saves the value for later use. For example, we can now change line 50 to

```
50 PRINT S; "TO THE";Y;"=";P
```

Watch out for subroutines which change the value of the input variable. The lowly "dummy variable" is quite handy in such cases.

repeat the process until you find the right tree . . . or surrender to frostbite. Or you can use a little trigonometry.

Trigonometry will let you figure the height, H, of the tree from two simple facts: your distance, D, from the base of the tree, and the angle, A, between the base and tip of the tree as measured from the point at which you're standing:



If you're standing as indicated in the drawing, then

H = D*TAN(A)

(That's "H equals D times the tangent of angle A.")

Here's where the TRS-80 comes in. (You'll also need a very long extension cord to run from the nearest electrical outlet to the site of the tree.)

TAN(X) is one of the trigonometric functions available as a subroutine for the TRS-80. Turn to page 218 of Appendix A and find the Tangent Subroutine. (It's the longest of the "trig" subroutines, because it actually contains two of the others, Sine and Cosine).

Type NEW to clear out the program memory and carefully type in the tangent subroutine (steps 3Ø3ØØ to 3Ø455). Be sure to add a protective END block: `3ØØØØ END`

Now type in the following program:

```
1Ø IN. "HOW FAR ARE YOU FROM BASE OF TREE";D

2Ø IN. "WHAT IS ANGLE BETWEEN TIP AND BASE OF TREE";A

3Ø X=A:GOSUB 3Ø32Ø

4Ø H=INT(D*Y+.5)

5Ø IF H=28 THEN 8Ø

6Ø P. "FIND ANOTHER TREE--THIS ONE IS";H; "FEET TALL."
```

```
7Ø P.:GOTO 1Ø

8Ø P. "CHOP IT DOWN AND TAKE IT HOME!"
```

RUN it. (After you've tried a few values for Distance and Angle, use Distance=16 and Angle=6Ø.) Hit BREAK to get out of the program.

A few notes on how the program works:

Line 3Ø gives X the value of angle A. This is necessary because the subroutine needs an X-input. Control is then transferred to 3Ø32Ø, the beginning of the tangent subroutine, which returns a value for Y = TAN(X).
Line 4Ø computes the height as D times Y (D times the Tangent of the Angle), and then rounds the answer off to the nearest integer.
Line 50 checks to see whether we've found our tree. If we have, program control goes to **Line 80**, where a suitable message is printed. Otherwise, **Line 60** tells us to find a new tree and **line 70** starts the program over again.

Why do we add .5 before using INT? Well, suppose Y*D=27.6. Simply taking INT(27.6) would give us an answer of 27 — even though the unrounded answer was closer to 28. By adding .5, we ensure that the number is rounded properly. Try it for a few other numbers just to make sure it works (e.g., 27.4, 14.8, 14.2). The technique is very useful throughout programming — whenever you want a properly rounded number.

But you can't find an extension cord that's long enough.

And you can't see beyond 12 feet due to the fast-falling snow. So now you need to know (in advance) what the angle will be when you're standing 12 feet away from a 28-foot tree. Then all you'll have to do is find a tree that gives you that angle reading on your surveyor's transit (or simple protractor) when you're standing 12 feet away.

Remember our formula,

H=D*TAN(A).

Well, in this case, we know H(Height) and D(Distance). What we're seeking is a certain angle such that

H/D=TAN(A).

In short, we want to find "the angle whose tangent is equal to H divided by D". In trigonometry, that's known as the Arctangent of A.

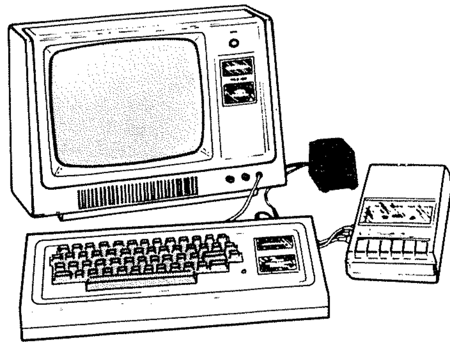
Don't worry — we've got a subroutine for that one, too.

EXERCISE 25-1: Write a program which accepts inputs for the height of the tree and your distance from the tree, and computes what the angle should be. Use the Arctangent Subroutine on page 219 of Appendix A.

EXERCISE 25-2: Write a program that produces the graph of SIN(X), with X taking on values from 0 to 360 degrees in 1-degree steps. Refer to the concentric ovals program for ideas on how to put your points on the screen. Remember that the ranges of X and SIN(X) will have to be adjusted to fit the 128 by 48 position screen.

Learned in Chapter 25

Miscellaneous
Subroutines — *regular* and *super*
"Dummy variables"



Subroutines available in Appendix A:

- Square Root
- Exponentiation
- Logarithms (Natural and Common)
- Exponential (Powers of e)
- Tangent
- Cosine
- Sine
- Arc Cosine
- Arc Sine
- Arc Tangent
- Sign

Notes:

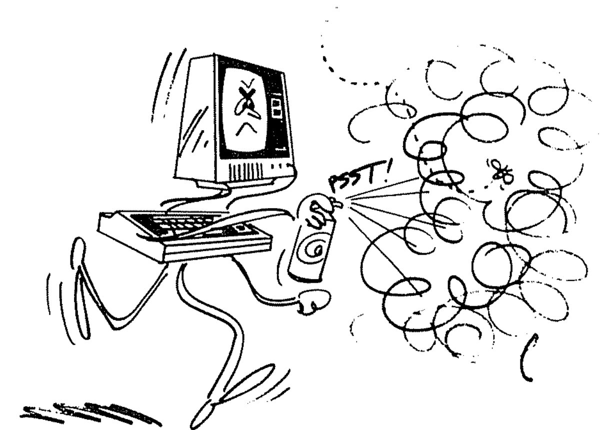
Chapter 26

DEBUGGING PROGRAMS

Quick – the RAID!

By now, the Computer has given you plenty of nasty messages like WHAT?, HOW? and SORRY. You know something's wrong, but it isn't always obvious exactly where, or why.

How do you find it? The answer is simple — **Be Very Systematic**. Even experienced programmers make lots of silly mistakes . . . but the experience teaches how to locate mistakes quickly.



Hardware, Cockpit or Software?

The first step in the "debugging" process is to isolate the problem as being either

1) A hardware problem,
2) An operator problem, or
3) A software problem.

Is it Farther to Ft. Worth or By Bus?

Starting with the least likely possibility — is the Computer itself working properly? Chances are (and our fondest desire is) that the Computer is working perfectly. There are several very fast ways to find out.

A. Type

```
PRINT MEM
```

or just `P.M.`

If there is no program loaded into memory, the answer should be

```
3583
```

or 7679 (8K machine); 15871 (16K).

If there is a program loaded, the answer should be some value LESS than 3583.

If the answer is MORE than 3583 (assuming of course you have not added more Radio Shack RAM), there may be trouble. Or, it's possible that the answer is a NEGATIVE number . . . same solution.

That's Random Access Memory.

Possible Solution

In either of the above cases, shut the Computer off completely. (Or, as they say in the big time, "Take it all the way down.") Let it sit for a full minute before turning it on again.

Yes, you will lose any program in memory, but at this point it's probably shot anyway. You could try to save it on cassette tape before turning off the machine if it makes you feel any better.

Turn the machine back on, type NEW and try the P.M. test again. If the results are the same, there is probably a RAM or ROM failure that will require Radio Shack Factory Authorized servicing.

ROM — that's Read Only Memory.

B. One Last Try

Before full panic sets in, however, type NEW and enter this program. It assigns every free memory location in RAM a specific value, then reads that value back out onto the screen.

Type (very carefully):

```
1ØF.X=1TO876:A(X)=X:N.X:F.Y=1TO876:P.A(Y);

2ØIFA(Y)-A(Y-1)<>1P."BAD"

3ØN.Y
```

No spaces please!

For 8K of memory, use 1889 instead of 876 in Line 1Ø; for 16K, use 3947.

Then, initialize A(Ø) to zero by typing

```
A(Ø) = Ø
```

Then, RUN

After about a 10-second wait for the array to "spin-up", the monitor should display

1 2 3 4 5 6 7 8 (etc., through 876)

. . . or 1889 . . . or 3947.

If the word "BAD" appears on the screen, you may have found the problem.

You will probably want to enter this test program into your computer, try it out *before you need it*, then save it on tape and hope that you won't . . . (need it, that is).

Video Display Problems?

The Video Display is very similar to a television set. It has adjustments for brightness, contrast, horizontal and vertical sync, etc. If these fail to give the desired display, the problem could be in the Computer.

Horizontal and vertical centering and "jitter" can be controlled by simple internal **computer**

adjustments, but you have to know where to adjust. Don't mess with it, or you could end up goofing up the voltage regulators instead, and wiping out the entire lineup of integrated circuits. Very expensive fiddling around!

BESIDES . . . AS THE WARRANTY POINTS OUT . . . OPENING THE CASE VOIDS YOUR WARRANTY.

Idiot here – What's *your* excuse?

Of course *you* don't make silly mistakes!

Now that's settled,

1. Is everything plugged in? Correctly? Firmly?
2. Are the recorder batteries fresh (if you're using batteries)?
3. Have you avoided the recorder ground loop problems discussed in the cassette recording chapter?
4. Is the recorder volume level properly set?
5. Is the recorder tone switch on "high"?
6. Are you using "legal" commands?

Remember . . . with some recorders it's a good idea to plug in either Aux or Earphone – but not both. Connecting both can produce "ground loop" problems – resulting in unwanted hum.

if so

Go walk the dog, then check it all over again.

If . . . Then

If the trouble was not found in the cockpit or with the hardware, there is probably something wrong with your program. Dump out the troublesome program. Load in one that is known to work and run it as a final hardware and operator check.

For a really complete checkout, load the combined Function and RAM Test program listed in Appendix C. If the test program runs okay, then the bug is in your program.

Error Messages

When the Computer gives us a `WHAT?` or `HOW?` message, it usually points out the offending program line, as in:

```
WHAT?
 10 X=S?QR(1-X*X)
```

In the case of `WHAT?` messages followed by a program line, the Computer inserts a question mark **just before** the error. In this case, the Computer doesn't recognize SQR (remember, we don't have a built-in square root function in LEVEL I BASIC), so it reads 10 X=S and then looks for a math operator, colon, end-of-line or other valid continuation. "Q" just doesn't fit, so the Computer treats it as an error.

With HOW? messages, the Computer inserts a question mark right **after** the error. For example:

```
HOW?

  10 PRINT INT(I)?/23
```

The question mark tells us that the error was discovered during execution of the INT function. We can guess that the value for I probably exceeded the allowable range for the INT function (value should be greater than −32768 and less than +32768).

How could we verify this immediately? Just command the Computer to PRINT I (using no line number).

Now let's take a look at some of the common sources of "computer-detected errors".

1. Assume the error is in a PRINT, or INPUT statement.
   Did you:
   a. Forget one of the needed pair of quotation marks?
      Example:

      ```
      10 PRINT "ANSWER IS, X: GOTO 5
      ```

      ERROR: No ending quotation mark after IS

      Yes, we know it's ok if the missing quote is the last character in the line.
   b. Use a variable name other than a single letter of the alphabet?
      Example:

      ```
      10 INPUT AG,S1
      ```

      ERROR: LEVEL I variable names can have neither more than one letter, nor a letter/number combination.
   c. Forget a semicolon or comma separating variables or text, or bury the semicolon or comma inside quotation marks?
      Example:

      ```
      10 PRINT "THE VALUE IS;"V
      ```

      ERROR: The semicolon is inside the quotation marks (so the "string" of words and the variable are not properly separated.)
   d. Forget the line number, accidentally mix a letter in with the number, or use a line number larger than 32767?
      Example:

      ```
      72B3 PRINT "BAD LINE NUMBER."
        ↑
        └─ERROR
      ```
   e. Accidentally have a double quotation mark in your text?
      Example:

      ```
      10 PRINT "HE SAID "HELLO THERE.""
      ```

f. Type a line more than 7Ø characters long?
g. Misspell PRINT or INPUT *(it happens!)?*
h. Accidentally type a stray character in the line, especially an extra comma or semi-colon?

2. If the error is in a READ statement, almost all the previous possibilities apply, plus:
   a. Is there really a DATA statement for the computer to read? Remember, it will only read a piece of DATA once unless it is RESTOREd.
   Example:

```
10 READ X,Y,Z

20 DATA 2,5,
```

   ERROR: There are only two numbers for the Computer to read. If you mean for Z to be zero, you must say so.

```
20 DATA 2,5,0
```

3. If the bad area is a FOR-NEXT loop, most of the previous possibilities also apply, plus:
   b. Do you have a NEXT statement to match the FOR?
   Example:

```
10 FOR A=1 TO N
```

   ERROR: Where's the NEXT A?
   c. Do you have all the requirements for a loop — a starting point, an ending point, a variable name, and a STEP size if it's not 1?
   Example:

```
10 A=1 TO N
```

   ERROR: Must have a FOR and a NEXT.
   d. Did you accidentally nest 2 loops using the same variable in both loops?
   Example:

```
10 FOR X=1 TO 5

20  FOR X=1 TO 3

30   PRINT X

40  NEXT X

50 NEXT X
```

   ERROR: The nested loops must have different variables.

NOTE: The "trailing comma" in the first line 2Ø can cause a full-blown program crash, requiring the Computer to be turned off, then on again to clear the trouble.

Some of these FOR-NEXT loop errors won't cause actual error messages; instead your program may wind up in endless loops, requiring the use of the BREAK key.

e. Does a variable in a loop have the same letter as the loop counter?
Example:

```
10 A=22
20 FOR R=1 TO 5
30  R=18
40   Y=R*A
50    PRINT Y
60 NEXT R
```

ERROR: The value of R was changed by another R inside the loop, and NEXT R was overrun, since 18 is larger than 5.

f. Did you nest loops incorrectly with one not completely inside the other?
Example:

```
10 FOR X=1 TO 6
20  FOR Y=1 TO 8
30   SET (X,Y)
40  NEXT X
50 NEXT Y
```

4. If the goofed-up statement is an IF-THEN or GOTO
   a. Does the line number specified by the THEN or GOTO really exist? Be especially careful of this error when you eliminate a line in the process of "improving" or "cleaning up" a program.

5. The error comes back as SORRY but the P.M. indicates there is room left in memory: If you get a SORRY and are using the A(X) numeric array, be sure to check P.M. then subtract 4 bytes for each array element used. You have probably overrun the amount of available memory.

6. The error comes back as HOW? and the program line containing the error is printed out with a question mark buried somewhere inside:
   a. Did you exceed the limits of one of the built-in functions?
   b. Did one of the values on the line exceed the maximum or minimum size for LEVEL I numbers?

c. Did you tell the Computer to divide by zero? (The Computer isn't about to let you get away with that one!)

To find out whether you did any of these things, PRINT the values for all the variables used in the offending line. If you still don't see the error, try carrying out the operations indicated on the line. For example, the error may occur during a multiplication of two very large numbers.

PRINT in calculator mode (no line number).

These certainly aren't all the possible errors one can make, but at least they give you some idea where to look first. Since we can't completely avoid silly errors, it's necessary to be able to recover from them as quickly as possible.

By the way . . . a one-semester course in beginning typing can do wonders for your programming speed and typing accuracy.

From the Ridiculous to the Sublime:

All the Computer can tell us is that we have (or have not) followed all of its rules. Assuming we have followed all the rules, the Computer will not ask "WHAT?" or "HOW?" — even if we're asking it to do something that's quite silly and isn't at all what we intended. It will dutifully put out garbage all day long if we feed it garbage — even though we follow its rules. Remember GIGO? If the program has no obvious errors, what might be the matter?

Typical "unreported" errors are:

1. Forgetting to initialize variables (and they are starting out with old values). Remember you cannot assume that unused variables are zero.
2. Accidentally reinitializing a variable — particularly easy when using loops.
   Example:

```
10 FOR N=1 TO 3
20  READ A
30  PRINT A
40  RESTORE
50 NEXT N
60 DATA 1,2,3
```

3. Reversing conditions, i.e. using "=" when you mean "<>", or "greater than" when you mean "less than."

4. Accidentally including "equals", as in "less than or equals", when you really mean only "less than."
5. Confusing similarly named variables, particularly the variable A, the string A$, and the array A(X). **They are not at all related.**
6. Forgetting the order of program execution — from left to right on each line, but multiplications and divisions always having priority before additions and subtractions. And intrinsic functions (INT, RND, ABS, etc.) having priority over everything else.
7. Counting incorrectly in loops. FOR I=Ø TO 7 causes the loop to be executed **eight**, not seven, times.
8. Using the same variable accidentally in two different places. This is okay if you don't need the old variable any more, but disastrous if you do. Be especially carefuly when combining programs or using the special subroutines in Chapter 25.

But how do you spot these errors if the Computer doesn't point them out? Use common sense and let the TRS-80 help you. The rules to follow are:

1. Isolate the error. Insert temporary "flags." Add STOP, END, and extra PRINT statements until you can track the error down to one or two lines.
2. Make your "tests" as simple as possible. Don't add complications until you've found the error.
3. Check simple cases by hand to test your logic, but let the Computer do the hard work. Don't try to wade through complex calculations with pencil and paper. You'll introduce more new mistakes than you'll find. Use the calculator mode, or a separate hand calculator to do that work.
4. Remember that you can force the Computer to start running a program at any line number you choose. Just type RUN ### (where ### represents the desired line number). This is a useful tool for working your way back through a program. You give the variables acceptable values using calculator-mode statements, and then RUN the program starting from some point midway through the program flow. If the answers are what you expect, then the error is before the "test point" you've created. Otherwise, the error is after the test point.
5. Remember also that it's not necessary to list the entire program just to get a look at one section of it. Just type LIST ### (where ### tells the Computer which line you'd like to start the list with).
6. Practice "defensive programming." Just because a program "works okay", don't assume it's dependable. Programs that accept input data and process it can be especially deceptive. Make a point of checking a new program at all the critical places. Examples: A square root program should be checked for inputs less than or equal to zero. Math functions you have programmed should be checked at points where the function is undefined, such as TAN(90°).

Examples of useful flags:

```
299 PRINT "        LINE #299"
399 IF X<Ø THEN PRINT "X OUT OF RANGE AT #399": STOP
```

Line 299 will help you check whether the line immediately following line 299 is executed. This helps you follow program flow.
Line 399 might be used to locate the point where X goes out of range.

Although the details would be different for your program, these techniques can be applied easily.

Beware of Creeping Elegance

Programs can grow to become more and more elegant with the ego reinforcement of the programmer as success follows success. With this "creeping elegance" comes increased chance of silly errors. It's fun to let your mind wander and add on some more program here, and some more there, but it's easy to lose sight of the purpose of the program. It is at times like this when the flow chart is ignored and the trouble begins. Nuff said.

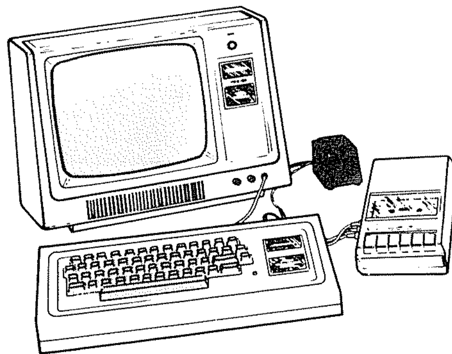
We'll leave you some space to make notes on your own debugging and troubleshooting ideas . . .

Share them with us too . . . especially if you come up with some really neat ones.

Learned in Chapter 26

Miscellaneous

- Defensive programming
- Computer-detected errors
- Flags
- Hardware checkout procedures



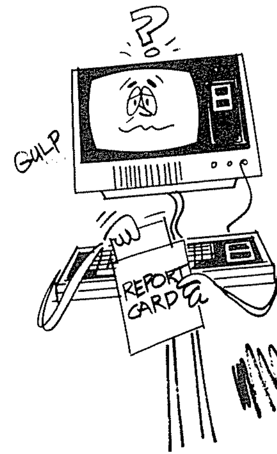
Notes:

Notes:

Notes:

Part B:

Answers to Exercises in the Chapters



Part B: Sample Answers

SAMPLE ANSWER FOR EXERCISE 3-1

```
5Ø PRINT D ENTER
```

SAMPLE RUN FOR EXERCISE 3-1

```
3ØØØ
```

Note: You may have used a different line number in your answer but the way to get the answer printed on the screen is by using the PRINT statement. If you didn't get it right the first time don't be discouraged. Type in line 50 above and RUN the program. Then return to Chapter 3 and continue.

SAMPLE ANSWER FOR EXERCISE 3-2

```
1Ø REM * TIME SOLUTION KNOWING DISTANCE AND RATE *

2Ø D = 3ØØØ

3Ø R = 5ØØ

4Ø T = D/R

5Ø PRINT "THE TIME REQUIRED IS";T; "HOURS."
```

NOTE: Remember to ENTER each line.

SAMPLE RUN FOR 3-2

```
THE TIME REQUIRED IS 6 HOURS.
```

Note: In order to arrive at the formula in line 4Ø it is necessary to transpose D = R * T and express in terms of T.

SAMPLE ANSWER FOR EXERCISE 3-3:

```
10 REM * CIRCUMFERENCE SOLUTION *

20 P = 3.14

30 D = 35

40 C = P * D

50 PRINT "THE CIRCLE'S CIRCUMFERENCE IS";C; "FEET."
```

SAMPLE RUN FOR 3-3

```
THE CIRCLE'S CIRCUMFERENCE IS 109.9 FEET.
```

Note: Since π is not included in Radio Shack's LEVEL I BASIC, we have to set a variable (in this case P was used) equal to the value of pi (3.14)

SAMPLE ANSWER FOR EXERCISE 3-4:

```
10 REM * CIRCULAR AREA SOLUTION *

20 P = 3.14

30 R = 5

40 A = P * R * R

50 PRINT "THE CIRCLE'S AREA IS";A; "SQUARE INCHES."
```

SAMPLE RUN FOR 3-4

```
THE CIRCLE'S AREA IS 78.5 INCHES.
```

Note: The LEVEL I BASIC system does not have a function which means "raise to the power" to handle R^2. (LEVEL II BASIC does.) In easy cases like this one, we can simply use R times R (R*R). If you have a LEVEL II system, you'll learn how to use the simple EXPONENTIATION function in the USER'S MANUAL for LEVEL II.

SAMPLE ANSWER FOR EXERCISE 3-5:

A bare-minimum effort might look like this: (C = checks, D = deposits, B = old balance, N = new balance.)

```
10 B=225

20 C=17+35+225

30 D=40+200

40 N=B-C+D

50 PRINT "YOUR NEW BALANCE IS $";N
```

SAMPLE ANSWER FOR EXERCISE 4-1:

```
10 REM * CAR MILES SOLUTION PROGRAM *

20 N = 1000000

30 D = 10000

40 T = N * D

50 PRINT "THE TOTAL NUMBER OF MILES DRIVEN IS "; T
```

SAMPLE RUN FOR 4-1

```
THE TOTAL NUMBER OF MILES DRIVEN IS 1E+10
```

Note: As discussed earlier, this answer is the number 1 followed by ten zeroes. 10,000,000,000. Ten Billion. The Computer will not print any numbers over 999,999 without converting them to exponential notation.

SAMPLE ANSWER FOR EXERCISE 4-2:

```
20 N = 1E+6

30 D = 1E+4
```

SAMPLE RUN FOR 4-2:

```
THE TOTAL NUMBER OF MILES DRIVEN IS 1E+10
```

Note: The answer came out exactly the same as before, meaning we not only receive answers in SSN, but can also use it in our programs.

SAMPLE ANSWER TO EXERCISE 5-1:

```
10 REM * FAHRENHEIT TO CELSIUS CONVERSION *

20 F = 65

30 C = (F - 32) * (5/9)

40 PRINT F; "DEGREES FAHRENHEIT =" ;C; "DEGREES CELSIUS."
```

SAMPLE RUN FOR 5-1:

```
65 DEGREES FAHRENHEIT = 18.3333 DEGREES CELSIUS.
```

Observe carefully how the parentheses were placed. As a general rule, when in doubt — use parentheses. The worst they can do is slow down calculating the answer by a few millionths of a second.

SAMPLE ANSWER TO EXERCISE 5-2:

```
30 C = F-32 * (5/9)
```

SAMPLE RUN FOR 5-2:

```
65 DEGREES FAHRENHEIT = 47.222 DEGREES CELSIUS.
```

Note how silently and dutifully the computer came up with the **WRONG** answer. It has done as we directed, and we directed it wrong.

A common phrase in computer circles is GIGO (pronounced "gee-joe"). It stands for "Garbage In — Garbage Out". We have given the computer garbage and it gave it back to us by way of a wrong answer.

Phrased another way, "Never in the history of mankind has there been a machine capable of making so many mistakes so rapidly and confidently." A computer is worthless unless it is programmed correctly.

SAMPLE ANSWER TO EXERCISE 5-3:

```
3Ø C = (F-32)* 5/9
```

SAMPLE RUN FOR 5-3:

```
65 DEGREES FAHRENHEIT = 18.3333 DEGREES CELSIUS.
```

SAMPLE ANSWER TO EXERCISE 5-4:

Two possible answers: 30 — (9 — 8) — (7 — 6) = 28

30 — (9 —(8—(7—6))) = 28

Sample Programs:

```
1Ø A = 3Ø - (9 - (8 - (7 - 6)))

2Ø PRINT A
```

Or line 10 might be

```
A = 3Ø - (9-8)-(7-6)
```

Try a few on your own.

SAMPLE ANSWER FOR EXERCISE 6-1:

```
1Ø A = 5

2Ø IF A<>5 THEN 5Ø

3Ø PRINT "A EQUALS 5"

4Ø END

5Ø PRINT "A DOES NOT EQUAL 5"
```

SAMPLE RUN FOR 6-1

```
A EQUALS 5
```

SAMPLE ANSWER FOR EXERCISE 6-2:

```
10 A = 6

20 IF A <>5 THEN 50

30 PRINT "A EQUALS 5"

40 END

50 PRINT "A DOES NOT EQUAL 5"

60 IF A<5 THEN 90

70 PRINT "A IS LARGER THAN 5"

80 END

90 PRINT "A IS SMALLER THAN 5"
```

SAMPLE RUN FOR 6-2:

```
A DOES NOT EQUAL 5

A IS LARGER THAN 5
```

Note: We had to put in another END statement (line 80) to keep the program from running on to line 90 after printing line 70.

SAMPLE ANSWER TO EXERCISE 11-1:

```
2 INPUT "HOW MANY SECONDS DELAY DO YOU WISH" ;S

3 P = 500

4 D = S * P

5 FOR X = 1 TO D

6 NEXT X

7 PRINT "DELAY IS OVER. TOOK" ;S;" SECONDS."
```

Explanation:

Line 2 used the input statement to obtain desired delay, S, in seconds.
Line 3 defined P, the number of passes required for a one-second delay.
Line 4 multiplied the delay for one second times number of seconds desired, and called that product D.
Line 5 began the FOR-NEXT loop from 1 to whatever is required.
Line 6 is the other half of the loop.
Line 7 reports the delay is over, and prints S, the number of seconds. Obviously, S is only as accurate as the program itself since it merely copies the value of S you entered in line 2.

SAMPLE ANSWER TO EXERCISE 11-2:

```
60 PRINT "RATE" , "TIME" , "DISTANCE"
65 PRINT "(MPH)" , "(HOURS)" , "( MILES)"
```

If you honestly had trouble with this one, better go back and start all over because you've missed the real basics.

SAMPLE ANSWER TO EXERCISE 11-3:

```
5 CLS
10 PRINT"      ***   S A L A R Y   R A T E   C H A R T   ***"
20 PRINT
30 PRINT "YEAR " ,"MONTH " ,"WEEK " ,"DAY "
40 PRINT
50 FOR Y=5000 TO 25000 STEP 1000
55  REM*CONVERT YEARLY INCOME INTO MONTHLY*
60  M=Y/12
65  REM*CONVERT YEARLY INCOME INTO WEEKLY*
70  W=Y/52
```

The FOR-NEXT-STEP function is limited to numbers between -32767 and +32767 (inclusive). If you specify upper or lower limits or a step size outside this range, you'll get a HOW? message.

Another not-so-obvious error will result if all your numbers are within the range but the sum of the upper limit and the step size exceeds 32767. For example, try

```
50 FOR Y=5000 TO 32700 STEP 1000
```

in the Salary Rate Chart program.

The way around this problem is to use smaller upper and lower limits and step size, and then use a scale factor in the loop to get the larger number. For example:

```
50 FOR Z=500 TO 3270 STEP 100
53 Y=Z*10
110 NEXT Z
```

will accomplish the same thing.

```
75  REM*CONVERT WEEKLY INCOME INTO DAILY*

80  D=W/5

100  PRINT Y,M,W,D

110  NEXT Y
```

SAMPLE RUN FOR 11-3:

```
              ***   SALARY RATE CHART   ***

 YEAR        MONTH          WEEK          DAY

  5000        416.667        96.1538       19.2308
  6000        500            115.385       23.0769
  7000        583.333        134.615       26.9231


                        ETC.
```

SAMPLE ANSWER FOR EXERCISE 11-4:

```
10 R = .01

20 D = 1

30 T = .01

35 CLS

40 PRINT "DAY " ,"DAILY " ,"TOTAL "

50 PRINT "  #" ,"RATE " ,"EARNED "

60 PRINT

70 PRINT D,R,T

80 IF R > 1E6 END

90 R = R * 2
```

```
100 D = D + 1

110 T = T + R

120 GOTO 70
```

SAMPLE RUN FOR 11-4:

```
DAY               DAILY                        TOTAL
 #                RATE                         EARNED


 1                 1.000000E-02                 1.000000E-02
 2                 2E-02                        3E-02
 3                 4E-02                        7E-02
 4                 8E-02                        .15
 5                 .16                          .31
 6                 .32                          .63


                     ETC.
```

SAMPLE ANSWER FOR EXERCISE 11-5:

```
  1 REM * FIND THE LARGEST AREA *

  5 CLS

 10 PRINT "WIRE  FENCE","LENGTH ","WIDTH ","AREA "

 20 PRINT "  (FEET)","(FEET)","(FEET)","(SQ. FEET)"

 30 F = 1000

 40 FOR L = 0 TO 500 STEP 50

 50  W = (F-2*L)/2

 60   A = L * W

 70   PRINT F,L,W,A
```

```
8Ø  NEXT L

9Ø END
```

SAMPLE RUN FOR EXERCISE 11-5:

WIRE FENCE (FEET)	LENGTH (FEET)	WIDTH (FEET)	AREA (SQ.FEET)
1ØØØ	Ø	5ØØ	Ø
1ØØØ	5Ø	45Ø	225ØØ
1ØØØ	1ØØ	4ØØ	4ØØØØ
1ØØØ	15Ø	35Ø	525ØØ
1ØØØ	2ØØ	3ØØ	6ØØØØ
1ØØØ	25Ø	25Ø	625ØØ
1ØØØ	3ØØ	2ØØ	6ØØØØ

ETC. . . .

ADDENDUM TO EXERCISE 11-5:

Here's a program that lets the Computer do the comparing:

```
5 CLS
9 REM *SET MAXIMUM AREA AT ZERO*
1Ø M=Ø
14 REM *SET DESIRED LENGTH AT ZERO*
15 N=Ø
19 REM *F IS TOTAL FEET OF FENCE AVAILABLE*
2Ø F=1ØØØ
24 REM *L IS LENGTH OF ONE SIDE OF RECTANGLE*
25 FOR L=Ø TO 5ØØ STEP 5Ø
29 REM *L IS WIDTH OF ONE SIDE OF RECTANGLE*
3Ø W=(F-2*L)/2
35 A=W*L
39 REM *COMPARE A WITH CURRENT MAXIMUM. REPLACE IF NECESSARY*
4Ø IF A<=M THEN GOTO 55
45 M=A
49 REM *ALSO UPDATE CURRENT DESIRED LENGTH*
5Ø N=L
55 NEXT L
6Ø PRINT "FOR LARGEST AREA USE THESE DIMENSIONS:"
65 PRINT N; "FT. BY";5ØØ-N;"FT. FOR TOTAL AREA OF";M;"SQ.FT."
```

SAMPLE RUN FOR ADDENDUM TO EXERCISE 11-5:

```
FOR LARGEST AREA USE THESE DIMENSIONS:
250 FT. BY 250 FT. FOR TOTAL AREA OF 62500 SQ.FT.
```

SAMPLE ANSWER FOR OPTIONAL EXERCISE 11-6 :

```
 10 REM * FINDS OPTIMUM LOAD TO SOURCE MATCH *

 20 CLS

 30 PRINT "LOAD ","CIRCUIT ","SOURCE ","LOAD "

 40 PRINT "RESISTANCE ","POWER ","POWER ","POWER "

 50 PRINT " (OHMS)"," (WATTS)"," (WATTS)"," (WATTS)"

 60 PRINT

 70 FOR R=1 TO 20

 80  I = 120/(10+R)

 90  C = I*I*(10+R)

100  S = I*I*10

110  L = I*I*R

120  PRINT R,C,S,L

130  NEXT R
```

NOTE: Use ↑ key to stop the display so you can examine it.

SAMPLE RUN FOR EXERCISE 11-6:

```
LOAD              CIRCUIT           SOURCE          LOAD
RESISTANCE        POWER             POWER           POWER
(OHMS)            (WATTS)           (WATTS)         (WATTS)
 1                 1309.09           1190.08         119.008
 2                 1200              1000            200
 3                 1107.69           852.071         255.621
```

ETC.

SAMPLE ANSWER FOR EXERCISE 12-1:

```
10 PRINT "THE " ,"TOTAL " ,"SPENT "

20 PRINT "BUDGET                YEAR'S             THIS"

30 PRINT TAB(0);"CATEGORY";TAB(16);"BUDGET";TAB(32);"MONTH"
```

SAMPLE ANSWER FOR EXERCISE 12-2:

```
30 PRINT TAB(1);"YEAR " ;TAB(12);"MONTH " ;TAB(25);"WEEK " ;

40 PRINT TAB(38);"DAY " ;TAB(51);"HOUR "

85 REM-CONVERT WEEKLY INCOME INTO HOURLY

90 H=W/40
```

SAMPLE RUN FOR 12-2:

```
          *** SALARY RATE CHART ***

YEAR        MONTH        WEEK         DAY          HOUR

5000        416.667      96.1538      19.2308      2.40385
6000        500          115.385      23.0769      2.88462

                  ETC.
```

SAMPLE ANSWER FOR 12-3:

```
30 PRINT "INTER " ;TAB(10); "LOAD " ;TAB(21);"CIRCUIT " ;
35 PRINT TAB(36);"SOURCE " ;TAB(51);"LOAD "
40 PRINT "RESIST " ;TAB(10);"RESIST " ;TAB(21);"POWER " ;
45 PRINT TAB(36);"POWER " ;TAB(51);"POWER "
50 PRINT " (OHMS)" ;TAB(10);" (OHMS)" ;TAB(21);" (WATTS)";
55 PRINT TAB(36);" (WATTS)" ;TAB(51);" (WATTS)"


120 PRINT"    10";TAB(10);R;TAB(20);C;TAB(35);S;TAB(50);L
```

SAMPLE RUN FOR EXERCISE 12-3:

```
INTER      LOAD      CIRCUIT      SOURCE     LOAD
RESIST     RESIST    POWER        POWER      POWER
(OHMS)     (OHMS)    (WATTS)      (WATTS)    (WATTS)

  10         1       1309.09      1190.08    119.008
  10         2       1200         1000       200
  10         3       1107.69      852.071    255.621


                ETC.
```

SAMPLE ANSWER FOR EXERCISE 13-1:

```
10 FOR A = 1 TO 3
20  PRINT " A LOOP"
30   FOR B = 1 TO 2
40    PRINT "  " ," B LOOP"
```

```
42      FOR C = 1 TO 4
44       PRINT "   ","   ","C LOOP"
48      NEXT C
5Ø    NEXT B
6Ø NEXT A
```

SAMPLE ANSWER FOR EXERCISE 13-2:

The program will be the same as the answer to Exercise 13-1 with the following additions:

```
45        FOR D = 1 TO 5
46         PRINT "   ","   ","   ","D LOOP"
47        NEXT D
```

Note: To get the full impact of this "4-deep" nesting, stop the RUN frequently to examine the nesting relationships between each of the loops.

SAMPLE ANSWER FOR EXERCISE 14-1:

Addition of the following single line gives a nice clean printout with all values "rounded" to their integer value:

```
55 A = INT(A)
```

Worth all the effort to learn it, wasn't it?

SAMPLE ANSWER FOR EXERCISE 14-2:

```
55 A = INT(1Ø * A)/1Ø
```

When 3.14159 was multiplied times 1Ø it became 31.4159. The INTEGER value of 31.4159 is 31. 31 divided by 1Ø is 3.1. Etc.

SAMPLE ANSWER FOR EXERCISE 14-3:

This was almost too easy.

```
55 A = INT(100 *A)/100
```

SAMPLE ANSWER FOR EXERCISE 14-4:

Oh Pshaw! And it seemed so easy.

You should have entered:

```
55 A = INT(1000 *A)/1000
```

Oh, you did? And you got 3 answers, then it crashed, saying:

```
HOW?
 55 A = INT(1000 * A)?/1000
```

It is all correct. Then why doesn't it work?

Well, if you have LEVEL II BASIC it did work. If you can't figure out why it didn't work with LEVEL I and you don't know why, you forgot to read the NOTE at the beginning of the lesson. Go back and read it.

OK. 32727 is the largest permissible number inside the brackets of INT(A). When the program tried to execute the fourth pass of the loop, it hit INT(1000 * 50.2654) etc. which becomes INT(50265.4) which is, of course, too big. So the Computer said HOW? But wait — there is a way to get the desired result. Try

```
55A = INT(A) + INT((A-INT(A))*1000/1000
```

Remember: Just because LEVEL I BASIC can't solve a problem one way doesn't mean it can't be done. You've just got to be a little fancier, sneakier . . . or whatever. And that's half the fun of programming!

SAMPLE ANSWER FOR EXERCISE 15-1:

```
10 INPUT " TYPE ANY NUMBER "   ;X

20 REM * SGN ROUTINE *

22 IF X <0 THEN T = -1

24 IF X = 0 THEN T = 0
```

```
26 IF X >Ø THEN T = +1

3Ø ON T+2 GOTO 5Ø,6Ø,7Ø

45 END

5Ø PRINT " THE NUMBER IS NEGATIVE."

55 END

6Ø PRINT " THE NUMBER IS ZERO."

65 END

7Ø PRINT " THE NUMBER IS POSITIVE."
```

SAMPLE ANSWER FOR EXERCISE 16-1:

```
 3 PRINT " SEE MY FOXY " ;

 5 FOR N = 1 TO 2

1Ø READ A$

2Ø DATA RADIO SHACK,TRS-80

3Ø PRINT A$; "  " ;

4Ø NEXT N
```

Analysis:

Line 3 PRINTs the first part, leaving a space for the printing from the upcoming A$, and has a trailing semicolon so the carriage return is suppressed.

Line 5 establishes a two-pass FOR-NEXT loop.

Line 1Ø Reads RADIO SHACK

Line 2Ø contains the two DATA strings, separated by a comma

Line 3Ø PRINTs RADIO SHACK and a space on the first pass of the N loop. Note the trailing semi-colon to again suppress the carriage return.

Line 4Ø returns control to line 5 and the loop. The second pass through the loop PRINTs TRS-80 (and another space, but it doesn't matter), finishing the job.

SAMPLE ANSWER FOR EXERCISE 20-1:

```
 10 INPUT "STARTING HORIZONTAL BLOCK (0 TO 127)";H
 20 INPUT "ENDING HORIZONTAL BLOCK (0 TO 127)";I
 30 INPUT "STARTING VERTICAL BLOCK (0 TO 47)";V
 40 INPUT "ENDING VERTICAL BLOCK (0 TO 47)";W
 50 CLS
 60 FOR X = H TO I
 70  FOR Y = V TO W
 80   SET(X,Y)
 90  NEXT Y
100 NEXT X
999 GOTO 999
```

SAMPLE ANSWER FOR EXERCISE 20-2:

The following lines are changed. The rest are the same.

```
 50 FOR L = X TO X+K*2+1
 70 SET(L,Y+K+1)
 90 FOR M = Y TO Y+K+1
110 SET(X+K*2+1,M)
```

SAMPLE ANSWER FOR EXERCISE 20-3:

A. MOVE THE DOT UP

```
10 INPUT "HORIZONTAL STARTING POINT (0 TO 127)";X
```

```
20 INPUT " VERTICAL STARTING POINT (Ø TO 47)";Y
30 CLS
40 RESET  (X,Y+1)
50 SET(X,Y)
60 Y = Y-1
70 IF Y> =Ø THEN 40
80 Y = Y+48
90 GOTO 40
99 GOTO 99
```

B. MOVE THE DOT TO THE LEFT

```
 10 INPUT " HORIZONTAL STARTING POINT(Ø TO 127)" ;X
 20 INPUT " VERTICAL STARTING POINT (Ø TO 47)" ;Y
 30 CLS
 40 RESET  (X+1,Y)
 50 SET  (X,Y)
 60 X = X-1
 70 OF X>=Ø THEN 4Ø
 80 X = X+128
 90 GOTO 4Ø
999 GOTO 999
```

SAMPLE ANSWER FOR EXERCISE 21-1:

Add or change the following lines:

```
10 IN."WHICH CAR'S ENGINE, COLOR & STYLE DO YOU WANT TO KNOW";W

130 FOR B = 201 TO 210

135  READ A(B)

140 NEXT B


180 P."LICENSE #","ENGINE SIZE","COLOR CODE","BODY STYLE"


210 P.W,A(W),A(W+100),A(W+200)
400 DATA 20,20,10,20,30,20,30,10,20,20
```

SAMPLE ANSWER FOR EXERCISE 22-1:

Insert the following lines:

```
105 IF Y=46 THEN 180

115 IF Y=1 THEN 180

150 PRINT AT Y*64+32,"    "

160 G.90

180 PRINT AT Y*64+32,"PING "

190 G.90
```

Note that line 180 prints the "ping" and line 150 makes it disappear by printing blanks in its place.

SAMPLE ANSWER FOR EXERCISE 23-1:

```
10 REM * TEST GRADER *
20 CLS
30 P. "THIS IS A TEST GRADING PROGRAM"
40 P. "ENTER THE STUDENT'S FIVE ANSWERS AS REQUESTED"
50 RESTORE
60 N=0
70 FOR I=1 TO 5
80 PRINT "ANSWER NUMBER";I;
90 INPUT A
100 READ B
110 PRINT A,B;
120 IF A=B THEN PRINT "CORRECT";:N=N+1
130 PRINT
140 NEXT I
150 PRINT N;"RIGHT OUT OF 5";
160 PRINT N/5 *100;"% "
170 P."ANY MORE TESTS TO GRADE";
180 IN." --1=YES, 2=NO";Z
190 IF Z=1 GOTO 50
200 DATA 65,23,17,56,39
```

SAMPLE ANSWER FOR EXERCISE 23-2:

```
10 REM * SAMPLE ANSWER 23-2*
```

```
100 CLS
110 P.:P.
120 P."ENTER  THE NUMBER OF ONE OF THE FOLLOWING INVESTMENTS"
130 P.
140 P."   1 - CERTIFICATE OF DEPOSIT"
150 P."   2 - BANK SAVINGS ACCOUNT"
160 P."   3 - CREDIT UNION"
170 P."   4 - MORTGAGE LOAN"
180 P.:IN." INVESTMENT" ;F
190 ON F GOTO 1000,2000,3000,4000
200 GOTO 100: REM USED IF NUMBER NOT BETWEEN 1 AND 4
1000 REM * CERTIFICATE OF DEPOSIT PROGRAM GOES HERE *
1010 P."THE C.D. PROGRAM HAS YET TO BE WRITTEN."
1020 GOS.10000:G.100
2000 REM * BANK SAVINGS ACCOUNT PROGRAM *
2010 CLS:P.:P."THIS ROUTINE CALCULATES SIMPLE INTEREST ON"
2020 P." DOLLARS HELD IN DEPOSIT FOR A SPECIFIED PERIOD"
2030 P."USING A SPECIFIED PERCENTAGE OF INTEREST.":P.
2040 P.:IN."HOW  LARGE IS THE DEPOSIT (IN DOLLARS)" ;P
2050 IN."HOW  LONG WILL YOU LEAVE IT IN (IN DAYS)" ;D
2060 IN."WHAT  INTEREST RATE DO YOU EXPECT (IN %)" ;R
2070 CLS:P.:P.:P."FOR  A STARTING PRINCIPAL OF $" ;P;" AT A"
```

```
2080 P." RATE OF " ;R;" % FOR " ;D;" DAYS, THE INTEREST "
2090 P."AMOUNTS TO $" ;
2100 REM INTEREST = ( % / YR ) / (DAYS/YR) * DAYS * PRINCIPAL
2200 I = R/100 / 365 * D * P
2300 P.:P."   "," $" ;I
2400 END
3000 REM * CREDIT UNION PROGRAM GOES HERE *
3010 P."THE C.U. PROGRAM HAS YET TO BE WRITTEN."
3020 GOS.10000:G.100
4000 REM * MORTGAGE LOAN PROGRAM GOES HERE*
4010 P." THE M.L. PROGRAM HAS YET TO BE WRITTEN."
4020 GOS.10000:G.100
10000 F.I=1TO2000:N.I:RET.
```

SAMPLE ANSWER TO EXERCISE 24-1:

Changes in only two lines are required:

```
60 IF (A=1) + (B=1) + (C=1) THEN 100
100 P."A GATE IS OPEN. OLD BESSIE IS FREE TO WANDER."
```

Line 60 reads "If gate A is open **OR** gate B is open **OR** gate C is open, then GOTO 100."

SAMPLE ANSWER TO EXERCISE 24-2:

```
40 IF ((INT (X/16) * 16-X)<>0) * ((INT (Y/6) * 6-Y)<>0) THEN 60
```

Here's one way to create uniform boundaries:

```
20 FOR X = 1 TO 126

30 FOR Y = 1 TO 46
```

SAMPLE ANSWER FOR EXERCISE 25-1:

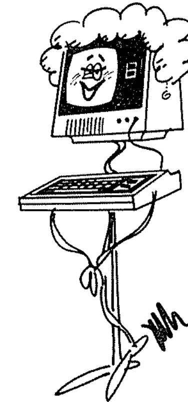
```
10 INPUT "DISTANCE FROM TREE";D

20 INPUT "HEIGHT OF TREE YOU'RE SEEKING";H

30 X=H/D:GOSUB 30660

40 PRINT "REQUIRED ANGLE IS" ;C; "DEGREES. "
```

SAMPLE ANSWER FOR EXERCISE 25-2:

```
1 CLS

10 FOR A=0 TO 360

20 X=A:GOSUB 30370

30 Y=-Y*20

40 SET(A/3,Y+22)

50 NEXT A

60 GOTO 1
```

Part C:

Some User's Programs



"USE ME-BUT BE GENTLE."

Part C: Some User's Programs

Test Grader Program

```
10 REM  * TEST GRADER *
20 CLS
30 A=1
40 B=2
50 C=3
60 D=4
70 E=5
80 T=10
90 F=11
100 N=10
110 P."WOULD  YOU LIKE TO INPUT THE ANSWERS"
120 P."ONE  AT A TIME OR 5 AT A TIME (ENTER 1 OR 5)";:IN.T
130 IF (T=1)+(T=5) GOTO 150
140 GOTO 120
150 IN."ENTER  THE STUDENT'S NAME (LAST NAME,FIRST NAME)"; A$, B$
160 CLS: P. "TEST FOR ";B$;"   " ;A$
170 REST.
180 R=0
190 FOR I=1 TO N STEP T
200 P."ENTER  ANSWER";:IF T=5 P."S ";I; "THROUGH ";
210 P.I+T-1;
220 IF T=5 IN.A(I),A(I+1),A(I+2),A(I+3),A(I+4):G.240
230 IN. A(I)
240 NEXT I
250 CLS
260 P." RESULTS ON TEST FROM " ;B$;"  " ;A$;" :"
270 FOR I=1 TO N
280 P.A(I);
290 READ Z
300 P.Z,
310 IF A(I)=Z P. "CORRECT "; : R=R+1
320 P.
330 NEXT I
340 P."PERCENTAGE  CORRECT:" ;INT(R/N*100+.5)
350 P.
360 G.150
370 DATA 5,3,A,D,C,E,T,T,F,T
```

Slowpoke

The kiddies (of all ages) will enjoy this one. It tests reaction time. When the computer says "G", you press any key to stop it. Then it's the next player's turn to RUN it. The player who stops it on the smallest number wins. Any player who gets a "SLOWPOKE" has to go take the dog for a walk.

With a little easy rework of the PRINT statements it can be converted into a "drunkometer" reaction time tester.

To change the speed of the printing, you can add a short FOR-NEXT loop between 190 and 200.

```
10 PRINT "    GET READY . . . . . . . . . . . "
20 FOR B = 1 TO 500
30 NEXT B
40 PRINT
50 PRINT
60 PRINT
70 PRINT TAB(30) , "GET  SET . . . . . . . . . . . . . "
```

```
8Ø X = RND(15ØØ)
9Ø FOR N = 1 TO X
1ØØ NEXT N
11Ø CLS
12Ø PRINT
13Ø PRINT
14Ø PRINT
15Ø PRINT
16Ø PRINT
17Ø PRINT TAB(3Ø) , " G O ! ! !"
18Ø FOR Z = 1 TO 1Ø
19Ø PRINT Z
2ØØ NEXT Z
21Ø PRINT
22Ø PRINT
23Ø PRINT
24Ø PRINT "      S L O W   P O K E"
25Ø FOR N = 1 TO 1ØØØ
26Ø NEXT N
```

12-Hour Clock

```
1Ø IN."THE HOUR IS";E
2Ø F=INT(E/1Ø):E=E-(F*1Ø)
3Ø IN."THE MINUTES ARE";C
4Ø D=INT(C/1Ø) :C=C-(D*1Ø)
5Ø IN."THE SECONDS ARE";A
6Ø B=INT(A/1Ø) :A=A-(B*1Ø)
7Ø F.N=1 TO 5ØØ:N.N
8Ø A=A+1
9Ø IF A>9 G.11Ø
1ØØ G.3ØØ
11Ø A=Ø
12Ø B=B+1
13Ø IF B>5 G.15Ø
14Ø G.3ØØ
15Ø B=Ø
16Ø C=C+1
17Ø IF C>9 G.19Ø
18Ø G.3ØØ
19Ø C=Ø
2ØØ D=D+1
21Ø IF D>5 G.23Ø
22Ø G.3ØØ
23Ø D=Ø
24Ø E=E+1
25Ø IF E>9 G.27Ø
26Ø G.29Ø
27Ø E=Ø
28Ø F=F+1
29Ø IF (F=1)*(E=3)A=Ø:B=Ø:C=Ø:D=Ø:E=1:F=Ø
3ØØ CLS
31Ø P.AT 47Ø,F;E;":";D;C;":";B;A
32Ø G.7Ø
```

Checksum For Business

For those responsible for inventory numbers or check clearing and balancing in business, a checksum is a most useful testing "code". This simple program calculates error-free checksums almost instantly. It is designed for 6-digit numbers and so can be used for stock number verification or other applications.

```
8 PRINT
9 REM * CHECKSUM PROGRAM *
10 IN." THE FIRST DIGIT IS " ;A
30 IN." THE SECOND DIGIT IS " ;B
50 IN." THE THIRD DIGIT IS " ;C
70 IN." THE FOURTH DIGIT IS " ;D
90 IN." THE FIFTH DIGIT IS " ;E
110 IN." THE SIXTH DIGIT IS " ;F
160 P.
170 P." THE NUMBER IS " ;A;B;C;D;E;F,
180 S = A + 2*B + C + 2*D + E + 2*F
210 T = INT(S/10)
220 U = S - T*10
230 S = T + U
240 IF S>9 G.210
290 P."    THE CHECKDIGIT IS " ;S
```

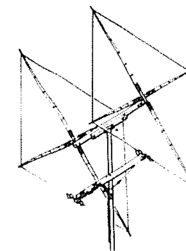
Design Program For Cubical Quad Antenna

The cubical quad is an exceptionally fine antenna for use in receiving and transmitting ham, citizens band, short wave broadcasting, industrial, public service radio and television signals. It is rotatable, being well-balanced and lightweight, even when not raised very far off the ground.

Electrically, it consists of two loops of wire, one of which is fed with coaxial cable or twin lead, the other simply soldered together at its ends. Figure 1 is an illustration of a quad.

The program inputs only your desired operating frequency. It then calculates and outputs all the mechanical dimensions needed so you can construct your own quad. Happy designing!

Figure 1



```
5 CLS
8 P." CUSTOM DESIGNING YOUR OWN HIGH GAIN ANTENNA >>-->>"
9 P.
10 IN." CENTER FREQUENCY (IN MEGAHERTZ) = " ;F
20 CLS
25 P." >-->   C U B I C A L  Q U A D  A N T E N N A  <--<"
30 P.
40 E=.985 * F
50 G=1.033 * F
60 D=1000/F
100 R=1032/F
140 B=118/F
180 X=(2*(R*R/64))
190 GOSUB 10000
195 S=Y
200 X=(S*S + (B*B/4))
210 GOSUB 10000
215 P=Y
220 X = ((R*R/64) + 75 * 75/(F*F*4))
230 GOSUB 10000
235 T=Y
240 X = ((R*R/64) + 125*125/(F*F*4))
242 GOSUB 10000
245 U=Y
260 W=468/F
520 P." THE DESIGN CENTER FREQUENCY IS" ;F;" MHZ.
```

```
525 P."THIS 2 ELEMENT QUAD SHOULD EXHIBIT A STANDING WAVE RATIO"
530 P."OF 2:1 OR LESS OVER THE FREQUENCY RANGE FROM";E;"TO"
535 P.G;" MHZ WHEN USED WITH 50 TO 75 OHM FEED LINE."
540 P.
570 P."THE BOOM LENGTH CAN VARY BETWEEN";75/F; "FEET AND"
575 P.125/F;" FEET WITH LITTLE EFFECT. A LENGTH OF";B
580 P."FEET IS OPTIMUM."
590 P.
620 P." TOTAL LENGTH OF THE WIRE IN THE DRIVEN ELEMENT IS" ;D
625 P." FEET, WHICH COMES TO" ;D/4;" FEET ON EACH SIDE."
630 P.
640 P.T.(20), "PRESS ENTER TO CONTINUE";:IN. A$
650 CLS
660 P." TOTAL LENGTH OF WIRE IN THE REFLECTION ELEMENT IS" ;R
665 P." FEET, WHICH IS" ;R/4;" FEET ON EACH SIDE."
670 P.
690 P." THE MINIMUM LENGTH OF BAMBOO. FIBERGLASS
    OR OTHER"
695 P."WILL BE";S; "FEET, MEASURED FROM THE CENTER OT THE"
697 P." BOOM.  IF A SPIDER (BOOMLESS) QUAD MOUNT IS USED,"
700 P."EACH SPREADER WILL HAVE TO BE AT LEAST";P; "FEET."
730 P.
740 P." THE TURNING RADIUS (FOR TREE CLEARANCE, ETC)
    WILL VARY"
745 P." BETWEEN" ;T;" FEET AND" ;U;" FEET, DEPENDING
    ON THE LENGTH"
750 P." OF THE BOOM."
755 P.
760 P.T.(20)," PRESS ENTER TO CONTINUE";: IN. A$
770 CLS
790 P." THIS QUAD ANTENNA WILL WORK WELL EVEN AT
    LOW HEIGHTS"
795 P." ABOVE THE GROUND, BUT IT WORKS BEST WHEN UP IN THE AIR"
800 P." A HALF-WAVELENGTH ---  " ;W;" FEET, OR MORE."
810 P.
840 P." THE FRONT-TO-BACK RATIO (ABILITY TO REDUCE
    UNWANTED"
845 P." SIGNALS FROM THE OPPOSITE DIRECTION) SHOULD
    EXCEED 10"
850 P." DECIBELS FROM ABOUT" ;.97*F;" TO" ;1.03*F;
    " MEGAHERTZ."
855 P." APPROACHING 25 DB AT" ;F;" MEGAHERTZ."
860 P.
880 P."   * * * * *  G O O D D X   * * * * * "
9999 END
10000 REM * SQUARE ROOT SUBROUTINE *
10010 IF X >= 0 GOTO 10040
10020 P." NO SUCH THING AS SQUARE ROOT OF NEGATIVE NUMBER"
10030 END
10040 Y=X/2
10050 Z=0
10060 W=(X/Y - Y)/2
10070 IF W=0 RET.
10080 IF W=Z RET.
10090 Y=Y+W:Z=W
10100 GOTO 10060
```

Speed Reading

Your Computer is your own personal Tachistoscope, a device used to practice speed reading. Study this sample program carefully to see how easy it is for you to substitute your own reading material at whatever reading level you want. The variable time loop lets you input the desired reading speed in words-per-minute.

```
3 REM * SPEED READING PROGRAM *
4 G.1Ø
5 F.I=1 TO B:N.I :PRINT AT 448;:RET.
6 REM AUDIO PROMPT GOES BEFORE RETURN IN ABOVE LINE.
1Ø I." HOW MANY WORDS PER MINUTE DO YOU READ" ;W
2Ø B=(12*6Ø/W) * 5ØØ
3Ø REM 5ØØ=FOR/NEXT LOOPS IN ONE SECOND
4Ø CLS
5Ø P.AT448;
1ØØ P."   SCARLETT O'HARA WAS NOT BEAUTIFUL, BUT MEN SELDOM
  " :GOS.5
1Ø2 P." REALIZED IT WHEN CAUGHT BY HER OWN CHARM AS THE TARLETON
  " :GOS.5
1Ø4 P." TWINS WERE.  IN HER FACE WERE TOO SHARPLY BLENDED THE
  " :GOS.5
1Ø6 P." DELICATE FEATURES OF HER MOTHER, A COAST ARISTOCRAT OF
  ":GOS.5
1Ø8 P." FRENCH DESCENT, AND THE HEAVY ONES OF HER FLORID IRISH
  " :GOS.5
11Ø P." FATHER.  BUT IT WAS AN ARRESTING FACE, POINTED OF CHIN,
  " :GOS.5
112 P." SQUARE OF JAW.  HER EYES WERE PALE GREEN WITHOUT A TOUCH
  " :GOS.5
114 P." OF HAZEL. STARRED WITH BRISTLY BLACK LASHES AND SLIGHTLY
  " :GOS.5
116 P." TILTED AT THE ENDS.  ABOVE THEM, HER THICK BLACK BROWS
  " :GOS.5
118 P." SLANTED UPWARDS, CUTTING A STARTLING OBLIQUE LINE IN HER
  " :GOS.5
12Ø P." MAGNOLIA-WHITE SKIN--THAT SKIN SO PRIZED BY SOUTHERN
  " :GOS.5
122 P." WOMEN AND SO CAREFULLY GUARDED WITH BONNETS, VEILS, AND
  " :GOS.5
124 P." MITTENS AGAINST HOT GEORGIA SUNS.
  " :GOS.5
```


The Wheel Of Fortune

(Or . . . Never Give a Sucker an Even Break.)

Modeled after the large wheels of fortune found at carnivals and other such gatherings, this graphics program accurately replicates its odds. The numbers are read from a DATA bank and "rotated" through "windows" as the wheel is "spun".

As commonly played, a $1 bet on any number, 1, 2, 5, 10, 20 or 40 (the Joker and TRS-80) returns those amounts – if that number comes up. If not – it's a cheap education.

Step right up, stranger. Try your luck at the wheel of fortune.

```
1Ø REM * WHEEL OF FORTUNE *
11 CLS:J=13:T=8Ø
13 P." STEP RIGHT UP, STRANGER.  TRY YOUR HAND AT THE" :P.:P.
15 P."         W H E E L  O F  F O R T U N E" :P.:P.
17 P." PAYOFFS IN DOLLARS FOR $1 BET ARE 1, 2, 5, 1Ø, 2Ø."
19 P.:P." SPECIALS ARE THE JOKER AND TRS-8Ø, EACH PAYING 4Ø."
22 P.:P." ENTER YOUR CHOICE AS 1,2,5,1Ø,2Ø, JOKER, OR TRS-8Ø." ;
23 IN.G
24 IF(G=1)+(G=2)+(G=5)+(G=1Ø)+(G=2Ø)+(G=13)+(G=8Ø)G.4Ø
26 P.AT64Ø;
3ØP."PLEASE ENTER A 1, 2, 5, 1Ø, 2Ø, JOKER, OR TRS-8Ø." ;:G.23
```

```
40 REM
50 CLS:P.AT24," WHEEL OF FORTUNE"
60 T=65
65 P=RND(54)
70 REST.
71 FOR I=1 TO 54
72 READ A(I)
73 NEXT I
80 REST.
81 FOR I=55 TO 60
82 READ A(I)
83 NEXT I
100 X=0:Y=18:GOS.3000
110 X=18:Y=12:GOS.3000
120 X=36:Y=9:GOS.3000
130 X=56:Y=6:GOS.3000
140 X=76:Y=9:GOS.3000
150 X=94:Y=12:GOS.3000
160 X=112:Y=18:GOS.3000
170 P.AT92," >>--->>" ;
190 P.AT594," ROUND & ROUND IT GOES . . ." ;
195 P.AT729," JOKER (13) &"
196 P.AT794," TRS-80 (80)"
198 P.AT856," BOTH PAY 40 TO 1"
200 FOR S=1 TO 100 + RND(2)
210 P.AT450,A(P);:P.AT331,A(P+1);:P.AT276,A(P+2);
215 P.AT222,A(P+3);
220 P.AT296,A(P+4);:P.AT369,A(P+5);:P.AT506,A(P+6);
221 IFS<TG.235
224 R=(S-T)*(S-T)*(S-T)/T
226 IFS<98P.AT594,"    PAYOFFS GO TO THE    " :G.230
227 P.AT594,"    ALMOST THERE . . . ";
230 IF S<102 FOR Z=1 TO R:NEXT Z
235 P=P-1
236 IF P=0 P=54
240 N.S:P.TAB(29);:Q=A(P+4):GOS.2000:X=0
250 P.TAB(22);" YOUR CHOICE WAS" ;: Q=G:GOS.2000
260 P.TAB(23);:IFG=A(P+4)P." YOU WIN AT" ;O;" TO 1" :E.
270 P."    YOU LOSE.    " :E.
500 D.1,2,80,1,5,1,2,1,10,1,2,1,5,1,2,1,5,1,2,1,20,1,2,10
510 D.1,2,1,5,1,2,1,5,1,2,13,1,2,1,10,1,2,1,2
520 D.1,20,1,2,5,1,2,10,1,2,5
2000 O=Q:IF(Q<>13)*(Q<>80)P." " ;Q:RET.
2010 O=40:A$="JOKER " :IFQ=80 A$=TRS-80
2020 P.A$:RET.
3000 FOR I=0 TO 7
3010 S.(X,I+Y):S.(X+1,I+Y)
3015 S.(X+14,I+Y):S.(X+15,I+Y)
3020 S.(I*2+X,Y):S.(I*2+1+X,Y)
3025 S.(I*2+X,7+Y):S.(I*2+1+X,7+Y)
3030 NEXT I
3040 RET.
```

Dow-Jones Industrial Average Forecaster

There is no guarantee that this program will make you instantly wealthy, but it is an example of converting a financial magazine article into a useable computer program. The article describing the market premises on which this program is built appeared on Page 90 of *Forbes*, June 1, 1977.

```
10 REM * FROM FORBES 6/1/77, P.90. ARTICLE BY BROWN *
20 P." ***PROJECTS TARGET DOW-JONES INDUSTRIAL AVERAGE AS A"
30 P." FUNCTION OF YEARS DJI EARNINGS AND INFLATION RATE*** "
40 P.
50 REM * K = COST OF MONEY.  ASSUME 3% *
60 K=.03
70 REM * P = RISK PREMIUM OF STOCKS OVER BONDS.  ASSUME 1% *
80 P=.01
85 Y=1
86 N=0
90 P. " DO YOU KNOW YEARS PROJECTED EARNINGS OF 30 DJI (Y/N)" ;
100 INPUT A
110 IF A = 1 THEN 270
120 P.
130 P." THIS METHOD WILL GIVE AN EARNINGS APPROXIMATION USING"
140 P." THE NEWSPAPER PRICES AND P/E RATIOS.  BETTER FORECASTS"
145 P." OF EACH COMPANY'S EARNINGS MAY GIVE AN IMPROVED"
150 P." OVERALL FORECAST."
160 P.
170 D=0
175 FOR N = 1 TO 30
180 READ A$
200 P." WHAT IS THE CURRENT PRICE OF >--> " ;A$;"  <--<" ;
210 INPUT P
220 P." THE CURRENT P/E RATIO" ;
230 INPUT R
240 E=P/R
250 D=E+D
260 N.N
265 P.
```



```
266 G.310
270 P." WHAT IS THE TOTAL PROJECTED EARNINGS FOR 1  SHARE OF" ;
280 P." EACH" ;
285 INPUT D
290 REM * I = ESTIMATED INFLATION RATE *
310 P." WHAT PERCENTAGE IS THE INFLATION RATE" ;
320 INPUT I
330 T = D/(K+P+I*.01)
340 R=T/D
350 P.
360 P." INFL. RATE" ," DJI EARN." ," PROJ DJ AVG" ," AVG/EARN RATIO"
370 P.
380 P.I,D,T,R
390 D.ALLIED CHEM, ALCOA, AMER BRANDS, AMER CAN, A.T.&T
400 D. BETH STEEL, CHRYSLER, DUPONT, E. KODAK, ESMARK, EXXON
410 D.G.E., GEN FOODS, GEN MOTORS, GOODYEAR, INCO
420 D.INT. HARV., INT. PAPER, JOHNS-MAN, MINN MM, OWENS-ILLS
430 D.PROCTER & G., SEARS, STD OIL CAL, TEXACO, UNION CARBIDE
440 D.U.S. STEEL, UNITED TECHNOL., WESTINGHOUSE, WOOLWORTH
```

On A Snowy Evening . . .

by Robert Frost

Who says computers only make stuffy mathematical calculations and are not for folks who appreciate the better things. If this one doesn't grab you, nothing will.

```
40 CLS
50 P.AT7," ON A SNOWY EVENING ......BY ROBERT FROST" ;
55 F.N=1TO2000:N.N
68 F.Z=1TO300
```

```
  70 SET(RND(127),RND(47))
  72 N.Z
  80 I=0
1000 P.AT525, " WHOSE WOODS THESE ARE I THINK I KNOW." ;
1001 GOSUB6000
1100 P.AT525," HIS HOUSE IS IN THE VILLAGE, THOUGH" ;
1101 GOSUB6000
1200 P.AT525," HE WILL NOT SEE ME STOPPING HERE    " ;
1201 GOSUB6000
1300 P.AT525," TO WATCH HIS WOODS FILL UP WITH SNOW" ;
1301 GOSUB6000
1400 P.AT525," MY LITTLE HORSE MUST THINK IT QUEER " ;
1401 GOSUB6000
1500 P.AT525," TO STOP WITHOUT A FARMHOUSE NEAR     " ;
1501 GOSUB6000
1600 P.AT525," BETWEEN THE WOODS AND FROZEN LAKE    " ;
1601 GOSUB6000
1700 P.AT525," THE DARKEST EVENING OF THE YEAR.      " ;
1701 GOSUB6000
1800 P.AT525," HE GIVES HIS HARNESS BELLS A SHAKE " ;
1801 GOSUB6000
1900 P.AT525," TO ASK IF THERE IS SOME MISTAKE." ;
1901 GOSUB6000
2000 P.AT525,"THE ONLY OTHER SOUND'S THE SWEEP";
2001 GOSUB6000
2100 P.AT525," OF EASY WIND AND DOWNYFLAKE.     " ;
2101 GOSUB6000
2200 P.AT525," THE WOODS ARE LOVELY, DARK  AND DEEP" ;
2201 GOSUB6000
2300 P.AT589," BUT I HAVE PROMISES TO KEEP." ;
2305 I=3
2310 GOSUB6000
2400 P.AT653," AND MILES TO GO BEFORE I SLEEP." ;
2405 I=6
2410 GOSUB6000
2500 P.AT717," AND MILES TO GO BEFORE I SLEEP." ;
2505 I=9
2510 GOSUB6000
5000 SET(RND(127),RND(47))
5001 G.5000
6000 F.N=1TO20
6020 X=RND(127)
6030 Y=RND(47)
6070 IF Y = 24+IG.6020
6080 IF Y = 25+I G.6020
6090 IF Y = 26+I G.6020
6100 SET(X,Y)
6150 F.A=1TO20:N.A
6200 N.N
6300 RETURN
```

Termites

A malicious sense of humor helps on this one. Its avowed purpose is to demonstrate the graphic RESET (X, Y) function, turning off the "lights" in a random fashion, but it's not without other redeeming value. If you don't like to sit by the fire and watch it snow while reading good poetry, you can always watch the termites eat your house down.

```
 40 CLS
 50 F.X=1TO127
 60 F.Y=3TO47
100 SET(X,Y)
120 N.Y:N.X
```

```
170 N=5715
180 P."           SEE THE TERMITES EAT.   ONLY";
185 P.AT45," BITES LEFT!";
200 X=RND(127)
220 Y=RND(45) + 2
300 IF POINT(X,Y)=0 G.200
500 RESET(X,Y)
550 N=N-1
600 P.AT36,N;
700 IF N=0 G.999
800 G.200
999 G.999
```

Sorry

SORRY is a popular board game by Parker Brothers. This program demonstrates how to load a deck of cards into a numerical array, draw them out in a random fashion, "reshuffle" the deck after the last card is drawn, and continue drawing. You may specify how many seconds delay you wish between each drawing of the cards, allowing as much time as desired to actually move the pieces on your own SORRY board. Have fun!

```
10 REM * RANDOM GENERATOR FOR GAME OF SORRY *
11 IN. "ENTER A NUMBER FROM 1 TO 100" ;N
12 F. I=1 TO N:J=RND(32767):N.I
15 CLS
20 P." STAND BY FOR THE SHUFFLING OF THE DECK OF CARDS."
21 P.
22 P.
30 P.
40 FOR N=1 TO 45
50 READ A(N)
60 NEXT N
```



```
66 P.:P.:P.
70 Y=1
75 P. "SHUFFLING COMPLETED . . . GAME CONTINUES!"
80 B=0
90 GOTO 110
100 B=35
110 R=INT(RND(45))
120 M=A(R)
130 IF M=0 GOTO 110
140 A(R) = 0
150 T=0
160 FOR Z=1 TO 45
170 T=A(Z) + T
180 NEXT Z
185 P.T.(27)," PRESS ENTER" ;:IN. A$
190 IF T=0 GOTO210
200 GOTO 240
210 P." END OF DECK.  THE CARDS ARE BEING RESHUFFLED."
220 RESTORE
230 GOTO 30
240 IF Y < 0 G.270
250 P.TAB(10);" RED"
260 GOTO 280
270 P. TAB(40);" GREEN"
280 IF M = 13 GOTO 300
290 P. TAB(B+15);M
300 ON M GOTO 320,340,590,380,590,590,400,590,590,430,450,590,470
310 GOTO 590
320 P.TAB(B);" MAY MOVE A NEW PIECE OUT"
330 GOTO 590
340 P.TAB(B);" MAY MOVE A NEW PIECE OUT"
345 P.:P.
350 P. TAB(B+5);" DRAW AGAIN . . . . "
```

```
360 P.
370 GOTO 630
380 P.TAB(B);" MUST BACK UP 4 SPACES"
390 GOTO 590
400 P.TAB(B);" MAY SPLIT THE 7 BETWEEN"
410 P.TAB(B+3);" 2 PIECES"
420 GOTO 590
430 P.TAB(B);" MAY MOVE BACKWARDS 1 SPACE"
440 GOTO 590
450 P. TAB(B);" CAN SWAP PIECES WITH OPPONENT"
460 GOTO 590
470 P.
480 P.
490 IF B=0 GOTO 550
500 P. "  GOTCHA      <<<---<<<   <<<---<<<" ;
510 P.TAB(49);" S O R R Y !"
520 P.
530 P.
540 GOTO 590
550 P.  "S O R R Y !  >>>--->>>   >>>--->>>" ;
560 P. TAB(55);" GOTCHA  !"
570 P.
580 P.
590 FOR X=1TO4
600 P. TAB(30);" *"
610 NEXT X
620 Y=Y*(-1)
630 IF Y>0 THEN 80
640 GOTO 100
650 D.1,1,1,1,1,2,2,2,2,3,3,3,3,4,4,4,4,5,5,5,5,7,7,7,7,8,8
660 D.8,8,10,10,10,10,11,11,11,11,12,12,12,12,13,13,13,13
```

Automatic Ticket Number Drawer

Like to make a big splash at the next Rotary Club, Country Fair, or other ticket drawing giveaway? This program uses the random number generator to pick the lucky number(s) and eliminate charges of stuffing the ticket box, besides giving the whole affair some pizzaz. If your own number comes up and you are charged with rigging the computer, you're on your own.

```
  3 IN.  "ENTER A NUMBER FROM 1 TO 100";N
  4 F.I=1 TO N: J=RND(32767):N.I.
  5 CLS
 10 REM * PICKS WINNER(S) BY DRAWING TICKET NUMBER *
 11 REM * NO MORE THAN 32767 TICKETS CAN BE SOLD *
 12 REM * BUT TICKET NUMBERS CAN RANGE TO 999999 & BEYOND *
 40 IN." THE LOWEST TICKET NUMBER IS" ;B
 50 P.
 70 IN." THE HIGHEST TICKET NUMBER IS " ;H
 80 P.
 90 E=H-B+1
 91 IF E<32768 G.110
100 P." TOO MANY TICKETS SOLD!" :END
110 IN." HOW MANY WINNERS DO YOU WANT " ;W
120 CLS
130 IF W > E G.269
140 P.
141 P.
142 P.
143 P.
180 P."*  A N D  T H E  W I N N I N G  ";
182 IF W > 1 G.185
183 P."T I C K E T  I S *"
```

```
184 G.200
185 P."T I C K E T S   A R E   *"
200 P.
205 A(0) = 0
210 FOR N = 1 TO W
220 Z = RND(E)
260 P.
262 P.TAB(12);" >----->>>   ";Z + B - 1
264 NEXT N
266 END
269 CLS:P.:P.:P.:P.:P.:P.:P.
270 P.TAB(8);" YOU CAN'T HAVE MORE WINNERS THAN" ;
272 P." ENTRIES - DUMMY !"
```

Craps

The game is as old as history. A testimonial to the intelligence and ingenuity of our ancient ancestors. An excellent way to demonstrate the running of twin Random Number Generators.

You don't need to know how to play the game — the computer will quickly teach you. (. . . There's one born every minute . . .)

```
  1 IN."ENTER A NUMBER FROM 1 TO 100";N
  2 F.I=1TON:J=RND(32767):N.I
 10 REM * CRAPS GAME *
 20 CLS
 30 GOSUB 300:P=N
 40 P.:P."YOU ROLLED ****";A;" AND ";B;"****"
 50 ON P GOTO 60, 120,120, 100,100,100,110,100,100,100,110,120
 60 REM * USED FOR THE ON STATEMENT IF P=1 (WHICH IT CAN'T)*
100 P." YOUR POINT IS" ;N:GOTO 130
110 PRINT " YOU WIN!" :P.:END
```



```
120 PRINT " YOU LOSE." :P.:END
130 GOSUB 300:M=N
135 P.:P."YOU ROLLED ****";A;" AND ";B;"****"
140 IF P=M THEN 110
150 IF M=7 THEN 120
160 G.130
300 A=RND(6):B=RND(6):N=A+B:RET.
310 RETURN
```

Fire When Ready, Gridley

You have probably seen this popular graphics display at your Radio Shack Store. It is very well done, and due to popular demand is printed here. Little boys of all ages are fascinated by it, and it's great for showing off your computer. You will want to keep this one on tape for fast loading.

```
CASTLE SHOT

  5 REM * CASTLE SHOT *
 10 INPUT " ENTER YOUR INITIALS" ;A$
 20 CLS
 30 Z=74
 40 FOR Y=17 TO 47
 50 FOR X=Z TO 127
 60 SET (X,Y)
 70 NEXT X
 80 IF Y<23 THEN Z=Z+2
200 NEXT Y
210 FOR X=75 TO 123 STEP 4
220 SET (X,16)
230 SET (X+1,16)
240 NEXT X
250 Q=0
300 FOR X=95 TO 125 STEP 5
```

```
310 FOR Y=47 TO 35 STEP -1
320 RESET (X,Y)
330 NEXT Y
340 NEXT X
400 FOR X=95 TO 125
410 RESET (X,34)
420 NEXT X
500 PRINT AT 688, A$;"'S CASTLE" ;
600 FOR X=73 TO 100
610 SET (X,12)
620 SET (X,13)
630 NEXT X
700 FOR X=85 TO 95
710 SET (X,14)
720 SET (X,15)
730 NEXT X
740 RESET (90,13)
750 RESET (91,13)
1000 FOR Z=1 TO 2
1010 FOR X=2 TO 14
1020 FOR Y=40 TO 43
1030 SET (X,Y)
1040 NEXT Y
1050 NEXT X
1100 FOR X=3 TO 13 STEP 2
1110 RESET (X,41)
1120 NEXT X
1130 RESET (7,43)
1140 RESET (8,43)
1190 REM---THIS WILL MAKE THE CANNON RECOIL
1200 FOR X=1 TO 100:NEXT X
1210 RESET (73,12)
```

```
1220 RESET (73,13)
1230 RESET (74,12)
1240 RESET (74,13)
1250 SET (101,12)
1260 SET (101,13)
1270 SET (102,12)
1280 SET (102,13)
1290 FOR X=1 TO 100:NEXT X
1300 SET (74,12)
1310 SET (74,13)
1320 SET (73,12)
1330 SET (73,13)
1340 RESET (102,12)
1350 RESET (102,13)
1360 RESET (101,12)
1370 RESET (101,13)

1500 FOR X=71 TO 2 STEP -1
1510 P=X-73
1520 Y=P*P/150 + 12
1530 SET (X,Y)
1540 SET (X-1,Y)
1600 RESET (X+1,Q)
1610 RESET (X,Q)
1620 Q=Y
1630 NEXT X
1640 PRINT AT 771, "KAPOW!" ;
1700 GOSUB 1900
1710 FOR X=1 TO 18
1720 RESET (X,45)
1730 RESET (X,36)
1740 RESET (X,37)
```

```
1750 NEXT X
1800 NEXT Z
1810 PRINT AT 0
1820 END
1900 FOR X=1 TO 1000
1910 NEXT X
1920 RETURN
```

House Security

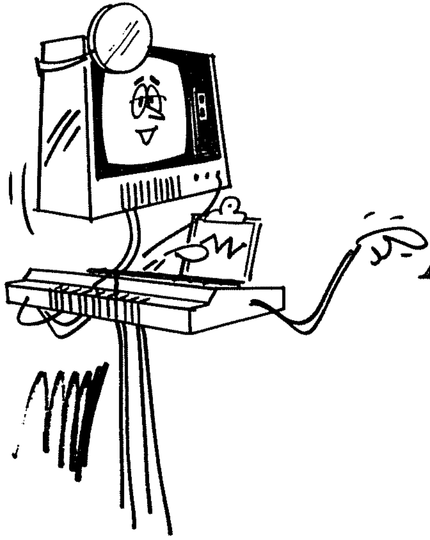
```
10 REM * LOGICAL AND PROGRAM *
15 CLS
20 Y=1:N=0:P." PLEASE ANSWER YES OR NO TO THE FOLLOWING QUESTIONS"
25 P.
30 INPUT " IS THE FRONT DOOR LOCKED" ;A
40 INPUT " IS THE BACK DOOR LOCKED" ;B
50 INPUT " IS THE KITCHEN WINDOW CLOSED" ;C
60 INPUT " IS THE BEDROOM WINDOW CLOSED AND LOCKED" ;D
70 INPUT " IS THE GARAGE DOOR LOCKED" ;E
75 P.:P.
80 IF (A=Y)*(B=Y)*(C=Y)*(D=Y)*(E=Y) THEN 120
90 P." HOUSE NOT LOCKED UP FOR THE NIGHT."
95 P.
100 P." PLEASE CHECK FOR AN UNLOCKED DOOR OR WINDOW."
110 END
120 P." HOUSE SECURITY CHECK SHOWS HOUSE LOCKED UP FOR THE NIGHT."
130 END
```


Loan Amortization

This program provides a fully developed installment plan for the repayment of small-to-moderate size loans, such as car or home improvement loans. The program includes all instructions necessary to using it. Use it with common sense; in the last payment period, amounts may be carried out to a fraction of a cent.

Challenge: modify the program to eliminate fractional-cent payments, without changing the total amount paid as interest or principal.

```
10 C=0:CLS:IN. "PRINCIPAL";P
20 IN. "# OF PERIODS";L
30 IN. "INTEREST RATE";R
40 I=R/12:I=I/100
50 T=1:F.X=1TOL
60 T=T*(1+I):N.X:T=1/T
70 T=1-T
80 M=P*I/T
85 M=INT(M*100+.5)/100
90 GOS.200
100 F.Z=1TOL
110 IFC<13G.120
115 IN. "PRESS ENTER TO CONTINUE";A$:C=0:GOS.200
120 A=(INT(P*I*100+.5))/100
130 B=M-A:P=P-B
140 P.Z;:P.T.(10),P;:P.T.(20),M;
150 P.T.(30),B;:P.T.(40),A
160 C=C+1:N.Z
170 END
200 CLS:P. "PAYMENT REMAINING MONTHLY PRINCIPAL INTEREST"
210 P. "NUMBER PRINCIPAL PAYMENT PAYMENT PAYMENT"
220 RET.
```



"IF THIS BOTHERS YOU-
WE'LL HAVE IT REMOVED."

Appendix:

A. Subroutines

B. Cassette Data Files

C. Combined Function and ROM Test

Appendix A: Subroutines

These subroutines will let you run programs which require advanced math functions not directly available in LEVEL I BASIC.

If you entered all the subroutines exactly as they're listed, you'd have less than 7ØØ bytes of memory left for your main program — not enough to do much of anything. So just enter the subroutines you need, and omit REM statements if you're still short on space.

Once you've entered a subroutine and gotten it running, save it on a Cassette. Try saving different combinations of subroutines on Cassettes: for example, make a SIN/COS/TAN cassette, a SIN/SQR cassette, an EXPONENTIATION/LOG/EXPONENTIAL/SGN cassette — whatever combinations are useful to you.

Subroutines listed in this Appendix:

- **Square Root**
- **Exponentiation**
- **Logarithms (Natural and Common)**
- **Exponential (Powers of e)**
- **Tangent**
- **Cosine**
- **Sine**
- **ArcCosine**
- **ArcSine**
- **ArcTangent**
- **Sign**

Each subroutine listing has a set of instructions in the margin. Study them closely. You'll see that some subroutines require other subroutines for internal calculations. **You must enter these "auxiliary subroutines" when the instructions call for them.**

Always enter 3ØØØØ END as a protective block when using subroutines. For complete information on the use of subroutines, see Chapter 25.

NOTE: Accuracy of the subroutines is less than the accuracy of LEVEL I math operators and intrinsic functions. This is due to two factors: 1. The subroutines contain many chain calculations, which tend to magnify the small error of individual operations. 2. These subroutines are only approximations of the functions they replace. In general, the subroutines are accurate to five or six decimal places over much of their allowable range, with a decrease in accuracy as the input approaches the upper or lower limits for input values.

Square Root

Computes: SQR(X), $\sqrt{X}$

Input: X, must be greater than or equal to zero

Output: Y

Also uses: W,Z internally

Other subroutines required: None

How to call: GOSUB 3ØØ3Ø

```
30000 END
30010 REM *SQUARE ROOT* INPUT X, OUTPUT Y
30020 REM ALSO USES W & Z INTERNALLY
30030 IF X = 0 T. Y = 0 : RET.
30040 IF X>0 T. 30060
30050 P. "ROOT OF NEGATIVE NUMBER?" : STOP
30060 Y=X*.5 : Z=0
30070 W=(X/Y-Y)*.5
30080 IF (W=0) + (W=Z) T. RET.
30090 Y=Y+W : Z=W : G. 30070
```

Exponentiation

Computes: X^Y (X to the Y power)

Input: X, Y. If X is less than zero, Y must be an odd integer

Output: P

Also uses: E, L, A, B, C internally. Value of X is changed.

Other subroutines required: Log and Exponential

How to call: 3Ø12Ø

```
3ØØØØ END
3Ø1ØØ REM *EXPONENTIATION* INPUT X,Y; OUTPUT P
3Ø11Ø REM ALSO USES E,L,A,B,C INTERNALLY
3Ø12Ø P=1 : E=Ø : IF Y = Ø T. RET.
3Ø13Ø IF (X<Ø)*(INT(Y)=Y) T. P=1-2*Y+4*INT(Y/2) : X=-X
3Ø14Ø IF X<>Ø T. GOS. 3Ø19Ø : X=Y*L : GOS. 3Ø25Ø
3Ø15Ø P=P*E : RET.
```

Logarithms (Natural and Common)

Computes: LOG(X) base e, and LOG(X) base 10

Input: X greater than or equal to zero

Output: L is natural log (base e), X is common log (base 10)

Also uses: A,B,C interally. Value of X is changed.

Other subroutines required: None

How to call: GOSUB 3Ø19Ø

```
3ØØØØ END
3Ø17Ø REM *NATURAL & COMMON LOG* INPUT X, OUTPUT L,X
3Ø175 REM OUTPUT L IS NATURAL LOG, OUTPUT X IS COMMON LOG
3Ø18Ø REM ALSO USES A,B,C INTERNALLY
3Ø19Ø E=Ø : IF X<Ø T. P. "LOG UNDEFINED AT" ;X:STOP
3Ø195 A=1 : B=2 : C=.5
3Ø2ØØ IF X>=A T. X=C*X : E=E+A : G. 3Ø2ØØ
3Ø2Ø5 IF X<C T. X=B*X : E=E-A : G. 3Ø2Ø5
3Ø21Ø X=(X-.7Ø71Ø7)/(X+.7Ø71Ø7) : L=X*X
3Ø215 L=(((.598979*L+.961471)*L+2.88539)*X+E-.5)*.693147
3Ø22Ø IF ABS(L)<1E-6 T. L=Ø
3Ø225 X=L*.4342945 : RET.
```

Exponential

Computes: EXP (X) (e to the X power)

Input: X

Output: E

Also uses: L,A internally. Value of X is changed.

Other subroutines required: None

How to call: GOSUB 3Ø25Ø

```
3ØØØØ END
3Ø24Ø REM *EXPONENTIAL* INPUT X, OUTPUT E
3Ø245 REM ALSO USES L,A INTERNALLY
3Ø25Ø L=INT(1.4427*X)+1 : IF L<127 T. 3Ø265
3Ø255 IF X>Ø T. P. "OVERFLOW " : STOP
3Ø26Ø E=Ø : RET.
3Ø265 E=.693147*L-X : A=1.32988E-3-1.41316E-4*E
3Ø275 E=(((A-.166665)*E+.5)*E-1)*E+1 : A=2
3Ø28Ø IF L<=Ø T. A=.5 : L=-L : IF L=Ø T. RET.
3Ø285 F. X=1 TO L : E=A*E : N. X : RET.
```

Tangent

Computes: TAN(X)

Input: X in degrees

Output: Y

Also Uses: A,C,W and Z internally. Value of X is changed.

Other subroutines required: Cosine, Sine

How to call: GOSUB 3Ø32Ø

```
3ØØØØ END
3Ø3ØØ REM *TANGENT* INPUT X IN DEGREES, OUTPUT Y
3Ø31Ø REM ALSO USES A,C,W,Z INTERNALLY
3Ø32Ø A=X : GOS. 3Ø36Ø
3Ø33Ø IF ABS (Y)<1E-5 T. P. "TANGENT UNDEFINED" : STOP
3Ø34Ø C=Y : X=A : GOS.3Ø376 : Y=Y/C : RET.
```

Cosine

Computes: COS(X)

Input: X in degrees

Output: Y

Also uses: W and Z internally. Value of X is changed.

Other subroutines required: Sine

How to call: GOSUB 3Ø36Ø

```
3ØØØØ END
3Ø35Ø REM *COSINE* INPUT X IN DEGREES, OUTPUT Y
3Ø351 REM ALSO USES W,Z INTERNALLY
3Ø36Ø W=ABS(X)/X:X=X+9Ø:GOS.3Ø376:IF(Z=-1)*(W=1)T.Y=-Y
3Ø365 RET.
```

Sine

Computes: SIN(X)

Input: X degrees

Output: Y

Also uses: Z internally. Value of X is changed.

Other subroutines required: None

How to Call: GOSUB 3Ø376

```
3ØØØØ END
3Ø37Ø REM *SIN* INPUT X IN DEGREES, OUTPUT Y
3Ø371 REM ALSO USES Z INTERNALLY
3Ø376 Z=ABS(X)/X:X=Z*X
3Ø38Ø IF X>36Ø T. X=X/36Ø : X=(X-INT(X))*36Ø
3Ø39Ø IFX>9ØT.X=X/9Ø:Y=INT(X):X=(X-Y)*9Ø:ONYG.3Ø41Ø,3Ø42Ø,3Ø43Ø
3Ø4ØØ X=X/57.29578 : IF ABS(X)<2.48616E-4 Y=O:RET.
3Ø4Ø5 G.3Ø44Ø
3Ø41Ø X=9Ø-X : G. 3Ø4ØØ
3Ø42Ø X=-X : G. 3Ø4ØØ
3Ø43Ø X=X-9Ø : G. 3Ø4ØØ
3Ø44Ø Y=X-X*X*X/6+X*X*X*X*X/12Ø-X*X*X*X*X*X*X/5Ø4Ø
3Ø45Ø Y=Y+X*X*X*X*X*X*X*X*X/36288Ø : IF Z=-1T.Y=-Y
3Ø455 RET.
```

ArcCosine

Computes: Arccos(S), angle whose cosine is S

Input: S, Ø<= S<= 1

Output: Y in degrees, W is in radians

Also uses: X,Z internally

Other subroutines required: ArcSine

How to call: GOSUB 3Ø5ØØ

```
3ØØØØ END
3Ø5ØØ REM *ARCCOS* INPUT S, OUTPUT Y,W
3Ø51Ø REM Y IS IN DEGREES, W IS IN RADIANS
3Ø52Ø GOS. 3Ø55Ø : Y=9Ø-Y : W = 1.57Ø796-W : RET.
```

ArcSine

Computes: ArcSin(S), angle whose sine is S

Input: S, Ø<= S<= 1

Output: Y in degrees, W in radians

Also uses: X,Y internally

Other subroutines required: None

How to call: 3Ø55Ø

```
3ØØØØ END
3Ø53Ø REM *ARCSIN SUBROUTINE* INPUT S, OUTPUT Y,W
3Ø535 REM Y IS IN DEGREES, W IS IN RADIANS
3Ø54Ø REM ALSO USES VARIABLES X,Z INTERNALLY
3Ø55Ø X=S : IF ABS(S)<=.7Ø71Ø7 T. 3Ø61Ø
3Ø56Ø X=1-S*S : IF X<Ø T. P. S; "IS OUT OF RANGE" : STOP
3Ø57Ø W=X/2 : Z=Ø
3Ø58Ø Y=(X/W-W)/2 : IF (Y=Ø)+(Y=Z) T. X=W : G. 3Ø61Ø
3Ø6ØØ W=W+Y : Z=Y G. 3Ø58Ø
3Ø61Ø Y=X+X*X*X/6+X*X*X*X*X*.Ø75+X*X*X*X*X*X*X*4.464286E-2
3Ø62Ø W=Y+X*X*X*X*X*X*X*X*X*3.Ø38194E-2
3Ø625 IF ABS(S)>.7Ø71Ø7 T. W=1.57Ø796-W
3Ø63Ø Y=W*57.29578 : RET.
```

ArcTangent

Computes: ATN(X), angle whose tangent is X

Input: X

Output: C in degrees, A in radians

Also uses: B,T internally. Value of X is changed.

Other subroutines required: Sign

How to call: GOSUB 3Ø69Ø

```
3ØØØØ END
3Ø66Ø REM *ARCTANGENT* INPUT X, OUTPUT C,A
3Ø67Ø REM C IS IN DEGREES. A IS IN RADIANS
3Ø68Ø REM ALSO USES B,T INTERNALLY
3Ø69Ø GOS. 3Ø81Ø : X=ABS(X) : C=Ø
3Ø7ØØ IF X>1 T. C=1 : X=1/X
3Ø71Ø A=X*X
3Ø72Ø B=((2.86623E-3*A-1.61657E-2)*A+4.29Ø96E-2)*A
3Ø73Ø B=((((B-7.5289E-2)*A+.1Ø6563)*A-.142Ø89)*A+.199936)*A
3Ø74Ø A=((B-.333332)*A+1)*X
3Ø75Ø IF C=1 T. A=1.57Ø796-A
3Ø76Ø A=T*A : C=A*57.29578 : RET.
```

Sign

Computes: SGN(X), the sign-component of X

Input: X

Output: To equal to −1 for X negative, Ø for X zero,
+1 for X positive

Also uses: No other variables

Other subroutines required: None

How to call: GOSUB 3Ø81Ø

```
3ØØØØ END
3Ø8ØØ REM *SIGN* INPUT X, OUTPUT T=-1,Ø OR +1
3Ø81Ø IF X<Ø T. T=-1
3Ø82Ø IF X=Ø T. T=Ø
3Ø83Ø IF X>Ø T. T=1
3Ø84Ø RET.
```

Appendix B: Cassette Data Files

The material in this Appendix is optional and yet very important. The more practical programming you do, the more you'll appreciate your TRS-80's data file capabilities. They allow you to go from the world of programming to the larger world of **data processing**.

What we mean is, you'll be able to do lots more with larger quantities of information.

Up to now we've relied on LEVEL I's 26 number variables A to Z, 876 (or less) array locations A(X), two string variables A$ and B$, and DATA lines to store the data our programs need. This leaves us with two limitations:

1. The Computer's memory may not be large enough to hold all the data we need (for example, an inventory list).
2. When we turn off the Computer, the values of A,B,A(X), etc., are lost.

Cassette data files solve both of these problems. We can save huge quantities of information on tape and retrieve them later, just as we save and reload programs. Only instead of the commands CSAVE and CLOAD, we use the special statements PRINT # and INPUT #.

To perform the exercises in this Appendix, you'll need to keep your Tape Recorder connected and set in the proper mode — RECORD, PLAY or STOP — as indicated in the text. Insert a blank cassette tape and set the tape counter to zero so you'll know where you started the data file.

Press RECORD and PLAY keys on your Recorder at the same time, then type in the following lines and RUN:

```
50 A=1:B=2:C=3

100 PRINT # A;",";B;",";C
```

Note the special punctuation required to separate each variable to be printed onto tape. **The sequence of five characters (;",";) must be inserted between every two variables in a PRINT # statement.**

This program causes three things to happen:

1. The Tape Recorder is automatically started (assuming you have it set in the RECORD mode).
2. The values of A, B and C are written onto the cassette.
3. The Recorder is automatically stopped. (You should then press STOP on the Recorder to disengage the recording head.)

You now have a permanent record which can easily be read back into the Computer. Note that the variables A, B and C are **not** written onto the tape — just the **values** of those variables (in this case, 1, 2 and 3) are stored.

To read back the data from tape, you must first press REWIND on the Recorder to rewind the tape to the point where the data file started. (You'll have to disconnect the REMote plug to gain manual control of the recorder. When you have rewound the tape to the starting point, reconnect the REMote plug.)

Type NEW to clear out the old program and enter these lines:

```
100 A=0: B=0: C=0

110 INPUT # A,B,C

120 PRINT "THE DATA HAS BEEN READ FROM THE TAPE."

130 PRINT "A=";A,"B=";B,"C=";C
```

Now press PLAY on the Recorder and type RUN.

If the data from the earlier program was stored and read properly, the Computer should display:

```
THE DATA HAS BEEN READ FROM THE TAPE

A= 1          B= 2          C= 3

READY

>_
```

Line 100 sets our variables to zero. If the data is not read properly, A, B and C will be output as zero.

Line 110 causes the Recorder to start, loading three numbers into the variables A, B and C. When the three numbers have been read, the Recorder motion is stopped.

Line 120 prints a reassuring message. This is important when the Computer is using an external device such as a Tape Recorder. Print messages are also valuable as prompting instructions to the user regarding the control of the Recorder. For example, before the Computer executes a PRINT # statement, we can have it print a message telling the user to put the Recorder in the Record mode.

Line 130 prints the data that was read from the tape.

No unusual punctuation is required to separate the variables on an INPUT # statement — just the ordinary commas.

NOTE: If the Recorder is not in the PLAY mode (with proper connections made) when it executes an INPUT # statement, the Computer will keep trying to read the tape until it gets something. You have no keyboard control of the Computer during such an input operation, so it is effectively locked-up. The only way to unlock it is to press the Reset button located in the expansion port on the left rear corner of the Keyboard. This will terminate the entire program, but will not erase it.

One last word of advice: If you PRINT # a list of, say, 1Ø values onto tape, you should INPUT # a list of 1Ø values also. If you don't match up the number of PRINT # items with the number of INPUT # items, you'll end up either losing data or going into the lock-up condition described above.

The following program demonstrates how a data file can be used to create a list of data items, process and update it. Study it carefully and think how similar programs might handle inventories, or any sequential lists.

```
  1 REM *AVG.TEMP AND HUMIDITY USING A DATA FILE*
  5 C=Ø:CLS
  7 B=Ø
 1Ø IN. "WHAT DAY OF THE MONTH IS IT";D
 2Ø IN."WHAT WAS THE TEMPERATURE TODAY";T
 3Ø IN. "WHAT WAS THE HUMIDITY";H
 4Ø IF D=1 THEN 16Ø
 5Ø P."LOAD PREVIOUS TEMPERATURES AND HUMIDITIES THIS MONTH."
 53 P."FIRST REWIND TAPE TO BEGINNING OF DATA FILE."
 55 P."THEN PRESS RECORDER'S PLAY KEY."
 6Ø IN."PRESS ENTER WHEN READY";A$
 7Ø FOR X=1 TO D-1
 8Ø INPUT # Y,Z
 9Ø B=B+Y
1ØØ C=C+Z
11Ø NEXT X
12Ø B=(B+T)/D
13Ø C=(C+H)/D
14Ø CLS:P. "THE AVERAGE TEMPERATURE IS";B
15Ø P."THE AVERAGE HUMIDITY IS";C
16Ø P.:P."NOW THE TRS-8Ø WILL WRITE"
17Ø P."TODAY'S TEMPERATURE AND HUMIDITY"
18Ø P."ONTO THE TAPE."
19Ø P."SO PRESS RECORD AND PLAY KEYS"
2ØØ P."BUT DO NOT REWIND."
21Ø IN."PRESS ENTER WHEN READY";A$
22Ø PRINT # T;",";H
23Ø P.:P."NOW TODAY'S INFO IS ADDED TO THE TAPE FILE."
24Ø P.:P."PLEASE PRESS STOP KEY ON THE RECORDER."
25Ø END
```

Line 7Ø reads back all the previous days' numbers, two at a time. When all the information is read in, the average temperature and humidity are calculated (using the current day's info as well).
Line 21Ø then writes the current day's information at the end of the list.

For a sample run of the program, assume it is the first day of the month. Enter plausible temperature and humidity figures. Continue running the program until you've got a cumulative listing for several days. Getting the feel for data files?

Suggestions for Further Use of Data Files

1. **Teaching/Testing**. Write a program that gives a multiple-choice test, for example, a vocabulary test. Include ten questions. The program should write the student's name and all ten responses onto a cassette data file. Design the program so that any number of students may take the test in sequence. Include instructions about when to use the RECORD, PLAY and STOP keys.

Write a grader program that uses the data file created above to read each student's name and responses, grade the test, and then read the next student's test. Be sure to leave time for the teacher to mark down the names and grades in his or her little black book.

2. **Inventory**. Write a program that sets up an array in which you store the following information about a group of cars;

License No.	Engine Size	Color Code	Body Style

The program should then store the array in a data file.

(See Chapter 21, where this same array was developed.)

Write another program which

1. Asks you which car you're interested in (you enter the license number).
2. Reads the data file until it comes to the correct license number.
3. Prints out all the information about that particular car.

Appendix C:
Combined Function and ROM Test

The following program puts the TRS-80 through its paces — all of them. If you're having trouble running a program, and you think it may be the Computer's fault, try this program on it. (First check to see that the Computer powered-up properly by running the P.M. test described in Chapter 26.)

Program execution is in three stages:

1. Function checkout (takes about 5 seconds)
2. RAM checkout (takes a few minutes)
3. Display checkout. This lets you check centering, straight-line distortion, etc. (Takes hardly any time at all — press **ENTER** to "redraw" test pattern.)

If at any point the Computer comes back with a "BREAK AT ###" (### will be a line number), you know that one of the functions isn't performing properly (ROM error). In case of a RAM error, BREAK message will be preceded by the message "RAM ERROR".

If you don't get a BREAK message (or an infinite loop), you can relax about the TRS-80 and go back to troubleshooting your program.

Type in the program VERY CAREFULLY, get it running properly, then save it on tape for later use.

```
10 IN. "TYPE 1, THEN PRESS ENTER";X

15 CLS:P.AT0; "TRS-80 FUNCTION TEST"

20 READ Y

30 DATA 2

40 RESTORE

50 READ Y

55 F.A=1TO1000:N.A

60 IFX>YSTOP
```

```
70 IFX>=YSTOP
80 IFY<XSTOP
90 IFY<=XSTOP
100 F.X=1TO10STEP2
110 GOTO130
120 STOP
130 GOS.150
140 GOTO160
150 RETURN
160 ONXGOTO180
170 STOP
180 SET(X,Y)
185 IFPOINT(X,Y)G.190
187 STOP
190 RESET(X,Y)
200 IFX<>Y-1STOP
210 IFY=X+1G.230
220 STOP
230 Z=RND(0)
240 X=1.1:X=INT(X)
245 Y=ABS(X)/2+.5
250 IFY=1G.270
260 STOP
270 REM EVERYTHING IS OK
```

```
290 CLS:P.TAB(5),"ALL  FUNCTIONS ARE O.K.,THE RAM TEXT IS NOW
    RUNNING."

300 A=M./4-1:B=0

310 F.Y=1TO8:Q=.5

320 F.B=1TOY:Q=Q*2:N.B

330 F.X=0TOA:A(X)=Q:N.X

340 F.X=0TOA:IFA(X)<>QP. "RAM  ERROR" :STOP

350 N.X

360 P.AT68,Q:N.Y:P.AT0; "THE  RAM TEST IS COMPLETE"

370 F.A=1TO2500:N.A

400 CLS:K=1

410 A$=GH

420 F.X=1TO32:P.A$;:N.X

430 F.X=1TO14:P.T.(29);A$:N.X

440 P.AT 469;

450 F.X=1TO9:P.A$;:N.X:P.

460 P.T.(21);:F.X=1TO9:P.A$;:N.X

470 P.AT960;

480 F.X=1TO31:P.A$;:N.X

490 IN.B$

500 IFK>0A$=80

510 IF K<0A$=GH

520 K=-K

530 CLS:G.420
```

Interface Specifications

Cassette

Suggested Input Level for Playback from Recorder	2 V peak-to-peak at a minimum impedance of 360K ohms
Typical Computer Output Level to Recorder	800 mV peak-to-peak at 1K ohm
Remote On/Off Switching Capability	0.5 A max at 6 VDC

DIN Jack Pin Connections (See Figure 1)

1 — Remote
2 — Signal ground
3 — Remote
4 — Input from recorder's earphone jack
5 — Output to recorder's Aux or Mic jack

Video Signal

DIN Jack Pin Connections (See Figure 1)

1 — +5 VDC at 50 mA
2 — Not used
3 — Not used
4 — Video signal, 1.4 V peak-to-peak, 0.4V negative sync, 75 ohms
5 — Ground

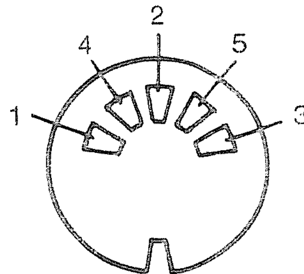


Figure 1. Pin Connections for TAPE and VIDEO DIN Jacks (viewed from rear of keyboard assembly)

Pin Connections for Expansion-Port Edge Card

(See Figure 2)

P/N	SIGNAL NAME	DESCRIPTION
1	RAS*	Row Address Strobe Output for 16-Pin Dynamic Rams
2	SYSRES*	System Reset Output, Low During Power Up Initialize or Reset Depressed
3	CAS*	Column Address Strobe Output for 16-Pin Dynamic Rams
4	A10	Address Output
5	A12	Address Output
6	A13	Address Output
7	A15	Address Output
8	GND	Signal Ground
9	A11	Address Output
10	A14	Address Output
11	A8	Address Output
12	OUT*	Peripheral Write Strobe Output
13	WR*	Memory Write Strobe Output
14	INTAK*	Interrupt Acknowledge Output
15	RD*	Memory Read Strobe Output
16	MUX	Multiplexor Control Output for 16-Pin Dynamic Rams
17	A9	Address Output
18	D4	Bidirectional Data Bus
19	IN*	Peripheral Read Strobe Output
20	D7	Bidirectional Data Bus
21	INT*	Interrupt Input (Maskable)
22	D1	Bidirectional Data Bus
23	TEST*	A Logic "Ø" on TEST* Input Tri-States AØ-A15, DØ-D7, WR*, RD*, IN*, OUT*, RAS*, CAS*, MUX*
24	D6	Bidirectional Data Bus
25	AØ	Address Output
26	D3	Bidirectional Data Bus
27	A1	Address Output
28	D5	Bidirectional Data Bus
29	GND	Signal Ground
30	DØ	Bidirectional Data Bus
31	A4	Address Bus
32	D2	Bidirectional Data Bus
33	WAIT*	Processor Wait Input, to Allow for Slow Memory
34	A3	Address Output
35	A5	Address Output
36	A7	Address Output
37	GND	Signal Ground
38	A6	Address Output
39	+5V	5 Volt Output (Limited Current)
40	A2	Address Output

NOTE: *means Negative (Logical "Ø") True Input or Output

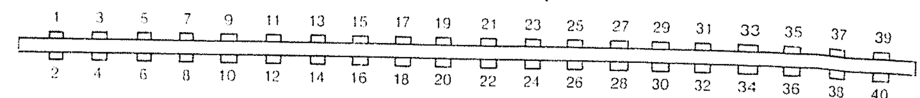


Figure 2. Connection points for Expansion-Port Edge Card (viewed from rear of keyboard assembly)

Notes:

Notes:

Notes:

Summary of LEVEL 1 BASIC

Commands	Purpose	Example	Described in Chapter(s)
NEW	Clears out all program lines stored in memory	NEW (not part of program)	1
RUN	Starts program execution at lowest-numbered line	RUN (not part of program)	1
RUN###	Starts program execution at specified line number	RUN 3ØØ (not part of program)	11
LIST	Displays the first 12 program lines stored in memory, starting at lowest numbered line. Use ↑ key to display higher-numbered lines (if any)	LIST (not part of program)	2
LIST###	Same as LIST, but starts at specified line number	LIST 3ØØ (not part of program)	11
CONT	Continues program execution when BREAK AT ### is displayed	CONT (not part of program)	11

Statements	Purpose	Example	Described in Chapter(s)
PRINT	Prints value of a variable or expression; also prints whatever is inside quotes	1Ø PRINT "A+B="; A+B	1,2,3
INPUT	Tells Computer to let you enter data from the Keyboard	1Ø INPUT A,B,C	7
INPUT	Also has built-in PRINT capability	1Ø INPUT "ENTER A"; A	7
READ	Reads data in DATA statement	1Ø READ A,B,C,A$	16
DATA	Holds data to be read by READ statement	2Ø DATA 1,2,3, "SALLY"	16
RESTORE	Causes next READ statement to start with first item in first DATA line	3Ø RESTORE	16
LET	(Optional) Assigns a new value to variable on left of equals sign	Ø LET A=3.14159	2
GOTO	Transfers program control to designated program line	1Ø GOTO 3ØØ	6

Statements	Purpose	Example	Described in Chapter(s)
IF-THEN	Establishes a test point	1ØIF A=B THEN 3ØØ	6
FOR-NEXT	Sets up a do-loop to be executed a specified number of times	1Ø FOR I=1 TO 1Ø 2Ø NEXT I	10,11 13
STEP	Specifies size of increment to be used in FOR-NEXT loops	1Ø FOR I=Ø TO 1Ø STEP 2	10
STOP	Stops program execution and prints BREAK AT ### message	1Ø IF A<B STOP	11
END	Ends program execution and sets program counter to zero	99 END	2
GOSUB	Transfers program control to subroutine beginning at specified line	1Ø GOSUB 3ØØØ	15,25
RETURN	Ends subroutine execution and returns control to GOSUB line	3Ø1Ø RETURN	15,25
ON	Multi-way branch used with GOTO and GOSUB.	1Ø ON N GOTO 3Ø,4Ø,5Ø 1Ø ON N GOSUB 3ØØØ, 4ØØØ, 5ØØØ	15

Print Modifiers	Purpose	Example	Described in Chapter(s)
AT	(Follows PRINT) Begins printing at specified location on Display	1Ø PRINT AT 65Ø, "HELLO"	22
TAB	(Follows PRINT) Begins printing at specified number of spaces from left margin	1Ø PRINT TAB (1Ø); "MONTH"; TAB (2Ø); "RECEIPTS"	12

Graphic Statements	Purpose	Example	Described in Chapter(s)
SET	Lights up a specified location on Display	1Ø SET (3Ø,4Ø)	20,22
RESET	Turns off a specified graphics location on Display	2Ø RESET (3Ø,4Ø)	20,22
POINT	Checks the specified graphics location: if point is "on", returns a 1; if "off", returns a Ø.	3Ø IF POINT (3Ø,4Ø)=1 THEN PRINT "ON"	22
CLS	Turns off all graphics locations (clears screen)	1Ø CLS	10,20

Built-In Functions	Description	Example	Described in Chapter(s)
MEM	Returns the number of free bytes left in memory	1Ø PRINT MEM	8
INT(X)	Returns the greatest integer which is less than or equal to X (-32768< x< 32768)	1Ø I=INT (Y)	14
ABS(X)	Absolute value of X	1Ø M=ABS (A)	17
RND (Ø)	Returns a random number between Ø and 1	1Ø X=RND(Ø)	19
RND(N)	Returns a random integer between 1 and N (1≦N< 32768)	1Ø X=RND(5ØØ)	19

Math Operators	Function	Example	Described in Chapter(s)
+	Addition	A+B	3
–	Subtraction	A–B	3
*	Multiplication	A*B	3
/	Division	A/B	3
=	Assigns value of right-hand side to variable on left-hand side	A=B	3

Relational Operators	Relationship	Example	Described in Chapter(s)
<	Is less than	A<B	6
>	Is greater than	A>B	6
=	Is equal to	A=B	6
<=	Is less than or equal to	A<=B	6
>=	Is greater than or equal to	A>=B	6
<>	Is not equal to	A<>B	6

Logical Operators	Function	Example	Described in Chapter(s)
*	AND	(A=3)*(B=7) "A equals 3 and B equals 7"	24
+	OR	(A=3)+(B=7) "A equals 3 or B equals 7"	24

Variables	Purpose	Example	Described in Chapter(s)
A through Z	Take on number values	A=3.14159	3
A$ and B$	Take on string values (up to 16 characters)	A$=RADIO SHACK	16
A(X)	Store the elements of a one-dimensional array (X≤ MEM/4-1)	A(Ø)=4ØØ	21

LEVEL I
Shorthand Dialect

Command/Statement	Abbreviation	Command/Statement	Abbreviation
PRINT	P.	TAB (after PRINT)	T.
NEW	N.	INT	I.
RUN	R.	GOSUB	GOS.
LIST	L.	RETURN	RET.
END	E.	READ	REA.
THEN	T.	DATA	D.
GOTO	G.	RESTORE	REST.
INPUT	IN.	ABS	A.
MEM	M.	RND	R.
FOR	F.	SET	S.
NEXT	N.	RESET	R.
STEP (after FOR)	S.	POINT	P.
STOP	ST.	PRINT AT	P.A.
CONT	C.	CLOAD	CL.
		CSAVE	CS.

U.S.A.: FORT WORTH, TEXAS 76102
CANADA: BARRIE, ONTARIO L4M 4W5

TANDY CORPORATION

AUSTRALIA	BELGIUM	U.K.
280-316 VICTORIA ROAD RYDALMERE, N.S.W. 2116	PARC INDUSTRIEL DE NANINNE 5140 NANINNE	BILSTON ROAD WEDNESBURY WEST MIDLANDS WS10 7JN