Getting Started With TRS-80® BASIC

For use with Models I, III & 4

Radio Shack®
The biggest name in little computers®

CUSTOM MANUFACTURED IN THE U.S.A. BY RADIO SHACK A DIVISION OF TANDY CORPORATION

TERMS AND CONDITIONS OF SALE AND LICENSE OF RADIO SHACK COMPUTER EQUIPMENT AND SOFTWARE PURCHASED FROM A RADIO SHACK COMPANY-OWNED COMPUTER CENTER, RETAIL STORE OR FROM A RADIO SHACK FRANCHISEE OR DEALER AT ITS AUTHORIZED LOCATION

LIMITED WARRANTY

I. CUSTOMER OBLIGATIONS

A. CUSTOMER assumes full responsibility that this Radio Shack computer hardware purchased (the "Equipment"), and any copies of Radio Shack software included with the Equipment or licensed separately (the "Software") meets the specifications, capacity, capabilities, versatility, and other requirements of CUSTOMER.

B. CUSTOMER assumes full responsibility for the condition and effectiveness of the operating environment in which the Equipment and Software are to function, and for its installation.

II. RADIO SHACK LIMITED WARRANTIES AND CONDITIONS OF SALE

A. For a period of ninety (90) calendar days from the date of the Radio Shack sales document received upon purchase of the Equipment, RADIO SHACK warrants to the original CUSTOMER that the Equipment and the medium upon which the Software is stored is free from manufacturing defects. THIS WARRANTY IS ONLY APPLICABLE TO PURCHASES OF RADIO SHACK EQUIPMENT BY THE ORIGINAL CUSTOMER FROM RADIO SHACK COMPANY-OWNED COMPUTER CENTERS, RETAIL STORES AND FROM RADIO SHACK FRANCHISEES AND DEALERS AT ITS AUTHORIZED LOCATION. The warranty is void if the Equipment's case or cabinet has been opened, or if the Equipment or Software has been subjected to improper or abnormal use. If a manufacturing defect is discovered during the stated warranty period, the defective Equipment must be returned to a Radio Shack Computer Center, a Radio Shack retail store, participating Radio Shack franchisee or Radio Shack dealer for repair, along with a copy of the sales document or lease agreement. The original CUSTOMER'S sole and exclusive remedy in the event of a defect is limited to the correction of the defect by repair, replacement, or refund of the purchase price, at RADIO SHACK'S election and sole expense. RADIO SHACK has no obligation to replace or repair expendable items.

B. RADIO SHACK makes no warranty as to the design, capability, capacity, or suitability for use of the Software, except as provided in this paragraph. Software is licensed on an "AS IS" basis, without warranty. The original CUSTOMER'S exclusive remedy, in the event of a Software manufacturing defect, is its repair or replacement within thirty (30) calendar days of the date of the Radio Shack sales document received upon license of the Software. The defective Software shall be returned to a Radio Shack Computer Center, a Radio Shack retail store, participating Radio Shack franchisee or Radio Shack dealer along with the sales document.

C. Except as provided herein no employee, agent, franchisee, dealer or other person is authorized to give any warranties of any nature on behalf of RADIO SHACK.

D. Except as provided herein, **RADIO SHACK MAKES NO WARRANTIES, INCLUDING WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.**

E. Some states do not allow limitations on how long an implied warranty lasts, so the above limitation(s) may not apply to CUSTOMER.

III. LIMITATION OF LIABILITY

A. EXCEPT AS PROVIDED HEREIN, RADIO SHACK SHALL HAVE NO LIABILITY OR RESPONSIBILITY TO CUSTOMER OR ANY OTHER PERSON OR ENTITY WITH RESPECT TO ANY LIABILITY, LOSS OR DAMAGE CAUSED OR ALLEGED TO BE CAUSED DIRECTLY OR INDIRECTLY BY "EQUIPMENT" OR "SOFTWARE" SOLD, LEASED, LICENSED OR FURNISHED BY RADIO SHACK, INCLUDING, BUT NOT LIMITED TO, ANY INTERRUPTION OF SERVICE, LOSS OF BUSINESS OR ANTICIPATORY PROFITS OR CONSEQUENTIAL DAMAGES RESULTING FROM THE USE OR OPERATION OF THE "EQUIPMENT" OR "SOFTWARE". IN NO EVENT SHALL RADIO SHACK BE LIABLE FOR LOSS OF PROFITS, OR ANY INDIRECT, SPECIAL, OR CONSEQUENTIAL DAMAGES ARISING OUT OF ANY BREACH OF THIS WARRANTY OR IN ANY MANNER ARISING OUT OF OR CONNECTED WITH THE SALE, LEASE, LICENSE, USE OR ANTICIPATED USE OF THE "EQUIPMENT" OR "SOFTWARE".

NOTWITHSTANDING THE ABOVE LIMITATIONS AND WARRANTIES, RADIO SHACK'S LIABILITY HEREUNDER FOR DAMAGES INCURRED BY CUSTOMER OR OTHERS SHALL NOT EXCEED THE AMOUNT PAID BY CUSTOMER FOR THE PARTICULAR "EQUIPMENT" OR "SOFTWARE" INVOLVED.

B. RADIO SHACK shall not be liable for any damages caused by delay in delivering or furnishing Equipment and/or Software.

C. No action arising out of any claimed breach of this Warranty or transactions under this Warranty may be brought more than two (2) years after the cause of action has accrued or more than four (4) years after the date of the Radio Shack sales document for the Equipment or Software, whichever first occurs.

D. Some states do not allow the limitation or exclusion of incidental or consequential damages, so the above limitation(s) or exclusion(s) may not apply to CUSTOMER.

IV. RADIO SHACK SOFTWARE LICENSE

RADIO SHACK grants to CUSTOMER a non-exclusive, paid-up license to use the RADIO SHACK Software on **one** computer, subject to the following provisions:

A. Except as otherwise provided in this Software License, applicable copyright laws shall apply to the Software.

B. Title to the medium on which the Software is recorded (cassette and/or diskette) or stored (ROM) is transferred to CUSTOMER, but not title to the Software.

C. CUSTOMER may use Software on one host computer and access that Software through one or more terminals if the Software permits this function.

D. CUSTOMER shall not use, make, manufacture, or reproduce copies of Software except for use on **one** computer and as is specifically provided in this Software License. Customer is expressly prohibited from disassembling the Software.

E. CUSTOMER is permitted to make additional copies of the Software **only** for backup or archival purposes or if additional copies are required in the operation of **one** computer with the Software, but only to the extent the Software allows a backup copy to be made. However, for TRSDOS Software, CUSTOMER is permitted to make a limited number of additional copies for CUSTOMER'S own use.

F. CUSTOMER may resell or distribute unmodified copies of the Software provided CUSTOMER has purchased one copy of the Software for each one sold or distributed. The provisions of this Software License shall also be applicable to third parties receiving copies of the Software from CUSTOMER.

G. All copyright notices shall be retained on all copies of the Software.

V. APPLICABILITY OF WARRANTY

A. The terms and conditions of this Warranty are applicable as between RADIO SHACK and CUSTOMER to either a sale of the Equipment and/or Software License to CUSTOMER or to a transaction whereby RADIO SHACK sells or conveys such Equipment to a third party for lease to CUSTOMER.

B. The limitations of liability and Warranty provisions herein shall inure to the benefit of RADIO SHACK, the author, owner and/or licensor of the Software and any manufacturer of the Equipment sold by RADIO SHACK.

VI. STATE LAW RIGHTS

The warranties granted herein give the **original** CUSTOMER specific legal rights, and the **original** CUSTOMER may have other rights which vary from state to state.

Getting Started with TRS-80® BASIC

Radio Shack®
A DIVISION OF TANDY CORPORATION
FORT WORTH, TEXAS 76102

Getting Started with TRS-80 BASIC:*
Copyright © 1981 Tandy Corporation,
Fort Worth, Texas 76102, U.S.A.
All rights reserved.

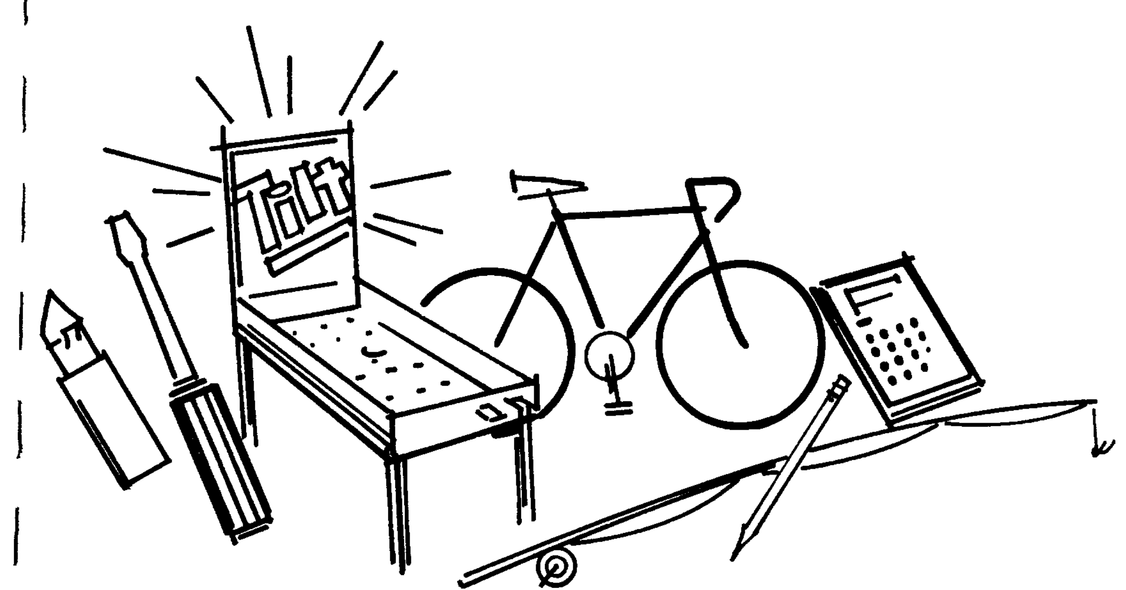
Reproduction or use, without express written permission from Tandy Corporation, of any portion of this manual, is prohibited. While reasonable efforts have been taken in the preparation of the manual to assure its accuracy, Tandy Corporation assumes no liability resulting from any errors or omissions in this manual or from the use of the information obtained herein.

TRS-80 Model III and Level II System Software:* Copyright ©
1980 Tandy Corporation and Microsoft.
All rights reserved.

The system software in these computers is retained in a read-only memory (ROM) format. All portions of this system software, whether in the ROM format or other source code format, and the ROM circuitry, are copyrighted and are the proprietary and trade secret information of Tandy Corporation and Microsoft. Use, reproduction, or publication of any portion of this material, without the prior written authorization by Tandy Corporation, is strictly prohibited.

This manual was written by George Stewart,
illustrated by Jack Wittry, and typeset by
LeWay Composing Service.

Printed in the United States of America
10 9 8 7 6 5



Can-opener . . . screw-driver . . . pin-ball machine . . . bicycle . . . pencil . . . calculator . . . fishing rod . . . Each one has a definite purpose.

BUT WHAT'S A COMPUTER FOR?

Funny thing is, that very question is what makes a Computer so useful and fun. Because *you* tell your Computer what it's going to be and do. . . your own bookkeeper, engineering assistant, secretary, electronic pin-ball machine, teacher, tax consultant, or poet-in-residence (all right, *hack* poet-in-residence). . .

HOW?

Well, you're already off to a good start, because your TRS-80 is the easiest-to-use Computer ever designed. *(Yes, we're a little prej-udiced—but you will be, too, after you've played around with it a while.)*

First, you've got to connect the TRS-80 System and turn it on. That's when the fun begins. Following the examples in this book, you'll start "talking" to the Computer, by typing on the Keyboard. When the Computer has something to say, it'll talk to you on the Video Display.

You'll find the TRS-80 is quite patient with beginners. If it doesn't understand you, it'll tell you so. When it does understand you, it will usually respond immediately.

And by the way . . . you can't break, damage, mess-up or otherwise hurt the Computer by typing on the Keyboard — *even if you press the key that says* **BREAK**!

So get ready to enjoy using your TRS-80!

Table of Contents

Part I: Getting Started

0. That's Right—Zero! Preface 2
1. Settin' Flat on Ready. Using the Keyboard 6
2. What to Do Before you Press (ENTER) 12
3. Re-Usable Instructions (Programs) 24

Part II: BASIC Training

4. BASIC Training (Crew-Cut Optional) 36
5. BASIC Training, Continued 50
6. Tricks and Treats 62
7. Don't Kill that Bug—Edit It! 74
8. Advanced Editing 82
9. A Precision Machine (Numbers) 94
10. A Giant (Scratch) Medium-Sized Step Forward 112
11. From Simple Loops Do Mighty Programs Grow 122
12. Automatic Transmission with Power Steering 134
13. Automatic Party Planner (AND, OR, NOT) 144
14. A Great Big Calculator 152
15. Bob's Corner Grocery (Arrays) 160
16. The Founding of an Empire 172
17. The Menace from Meklovakia (Strings) 184
18. Our Man in Meklovakia, continued 194
19. Assault on the Automatic Censor 200

Part III: Exploring the Territory

20. Advanced PRINT 212
21. PRINT USING 220
22. Control that Cursor! 236
23. Graphics 248
24. Graphics—The Finer Points 260
25. Keys Alive! (INKEY$) 270
26. Does Your Data Go Away? (Data Tapes and Hard Copy) 282
27. Peeking and Poking down Memory Lane 288
28. Lion Tamers, Fire Eaters, and Error Handlers 298

Appendixes

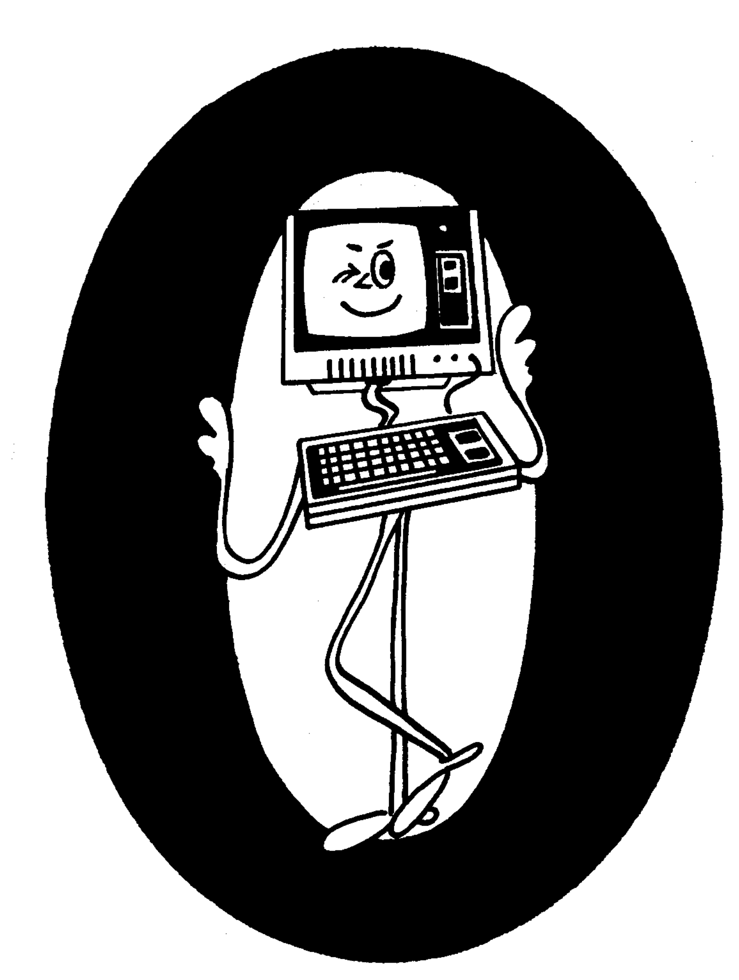
A. Answers to Do-It-Yourself Projects and Checkpoints 312
B. Sample Programs 326
C. Error Codes and Messages 332
D. Tables. TRSCII. Reserved Words. Graphics Characters 334
E. Display Worksheet 339

Customer Information 340
Index 341

Part I
Getting Started

- Using the Keyboard
- Useful Names and Definitions
- Printing Messages and Numbers

Chapter



Zero

That's Right — ZERO!

This is a do-it-now book. Put it close to your TRS-80 and as you read it, do it: type in the "computer instructions" just as they appear on these pages.

But first, get the TRS-80 ready. If this is your first time to use this Computer, then read your Computer's connection and operation instructions very carefully. This will tell you how to set it up, turn it on, and perform some other essential operations.

Where Do You Begin?

This book is divided into three parts:

I. **Getting Started** For first-timers and others who want to start at the beginning.

II. **BASIC Training** Here's where we really get going with TRS-80 BASIC. We'll assume you already know your way around the Computer.

III. **Exploring the Territory** We discover the advanced features of BASIC, and put together some fancy and fanciful programs.

Model III and Model 4 Users: Connection and basic operation are described in the *Model III/4 Operation and BASIC Language Reference Manual,* Part I (Radio Shack Cat. No. 26-1067).

Model I Users: Connection and basic operation are described in the *Level II BASIC Reference Manual* (26-2102).

Model 4 Users: With the BASIC described in this manual, your Computer works the same way as a Model III. Unless otherwise specified, follow the same procedures described for Model III users.

If you own a Model 4 Disk System, hold down (BREAK), then press the reset button. Several questions will appear on the screen. Simply press (ENTER) to answer each question, until the screen shows: READY >. Your Model 4 Owner's Manual summarizes the differences between the BASIC described here and Model 4 Disk BASIC.

Try out Part I; if it's all old-hat to you, skim through it and begin at Part II. Unless you're a real TRS-80 veteran, you'll probably want to work your way up to Part III.

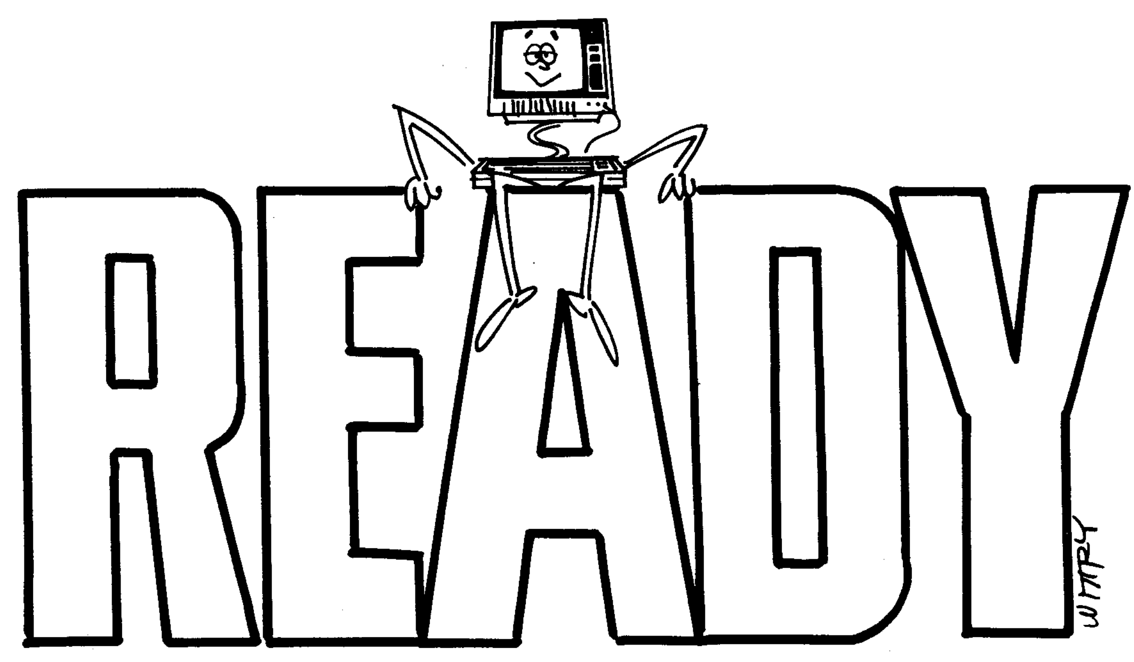
We have placed "Do It Yourself" projects throughout all three parts. Be sure to try these — they will lead you to some exciting discoveries. Whenever possible, we've included blank lines for your use in working out a D.I.Y. project. We've also put these lines at the end of many of the chapters. Use them!

Whenever you complete a project (or maybe you'll be stumped once in a while?), compare your work with ours in Appendix A. There are many ways to solve every problem — sometimes yours will be better than ours. One of the goals of this book is to help you select the best approach for your *own* programming projects.

Marginalia Can Be Fun. . .

In case of questions, look in this margin. Often you'll find helpful notes here (plus a few off-the wall remarks of "marginal" value). This is also where we'll put information about one particular model of TRS-80.

Chapter



One

Settin' Flat on Ready

Okay, your Computer is turned on, and it's telling you that it is ready for you to take charge:

```
READY
>
```

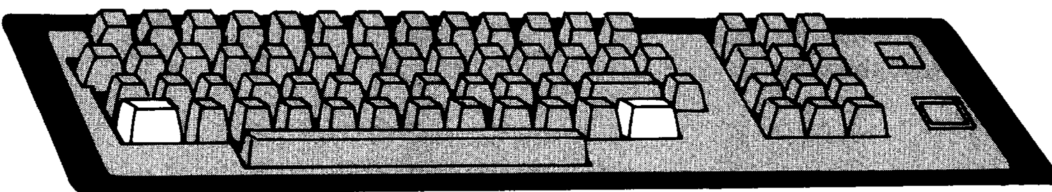
Whenever the Computer displays the ">"prompt, you know it is waiting for you to type something in. A "position indicator" — a blinking block or underline — after the ">" will always tell you where the next key you press will be displayed.

Keyboard Magic

First try out the letter and number keys. Type in your name and address. (To type in a blank space, press the bar at the bottom of the Keyboard.) For example,

```
>MILLIE GRAMM, 65 HUNT AND PECK LANE
```

Notice that many keys have two characters—for example, 1 and ! share the same key. To type in the upper character, press (SHIFT) and hold it down while you press the desired key.



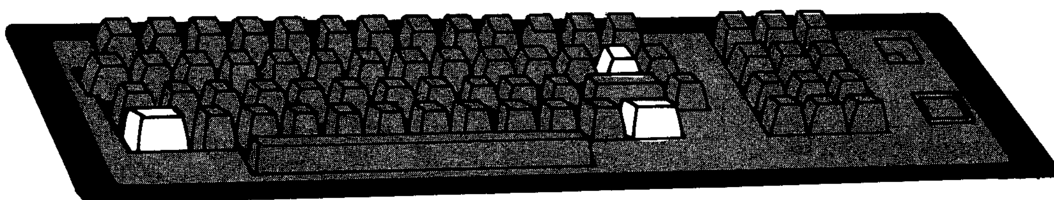
Go ahead and type away at the number, letter and punctuation keys until you've filled up one line and part of the next.

Note: Some Models of the TRS-80 can display lower-case. Check your Owner's Manual for instructions on using this capability.

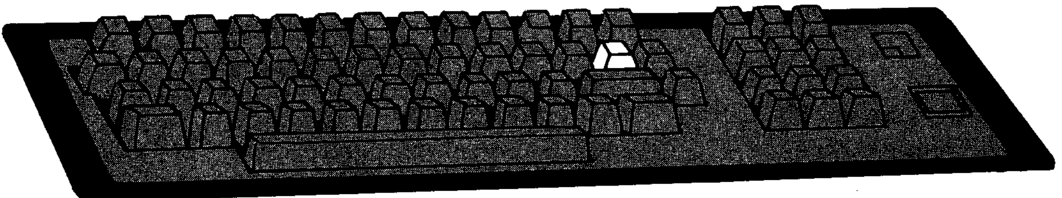
In this manual, everything will be done in the upper-case. If you have the lower-case capability, just put your Computer in the "caps-only" mode as explained in the Owner's Manual.

Backup Keys—Local and Express

It's easy to correct a typing error on the TRS-80 . . . no eraser needed. Just press the back-arrow ⊖ key, and the last character you typed in will disappear. Press it again and the next to last will be wiped out. You can erase your way back to the beginning of the first line this way. Some people call that The Woodpecker Method.



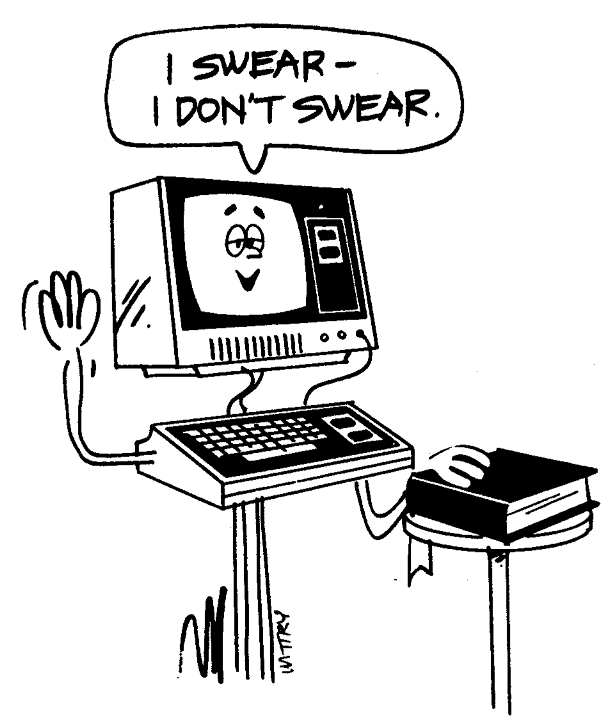
But there's a much faster way. Press the (SHIFT) key and hold it down while you press the back-arrow ⊖ key. Zap—you're back to the beginning of your first line!



Some Strong Language

If you've just pressed (SHIFT) ⊖, the position indicator is sitting at the beginning of the line. While it's sitting there, let's give it a better name than position indicator. How about . . . cursor? No, that's not one who swears, it's Latin for "runner". And that's what the cursor does—runs all over the Display, showing exactly where the next character will be displayed.

In Model III mode, Model 4 has two keys for deleting characters: the back-arrow key and the (F3) key. The (F3) key comes in handy when working on the numeric side of your keyboard. However, you still have to press (SHIFT) ⊖ to simultaneously erase all the characters to the beginning of the line.

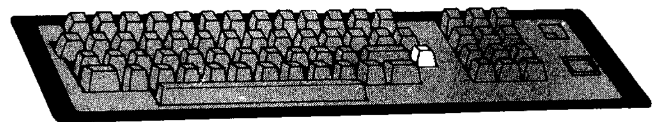


See the Cursor Run. . .

Tapping over and over on the space bar will make the cursor creep along one space at a time (So why didn't we call it a creeper?). To make it run, press the forward arrow →. Go ahead, press → several times.

You typists can think of the → as the tab key. The tab stops are pre-set at 8, 16, 24, 32, etc.

Now press SHIFT ← to see the cursor skip back to the beginning of the line.



Getting tired of looking at all of those old lines you typed in? Well, in case you don't have any masking tape handy, why not just press the CLEAR key? That's called starting off with a clean slate! The only thing left is our familiar little runner.

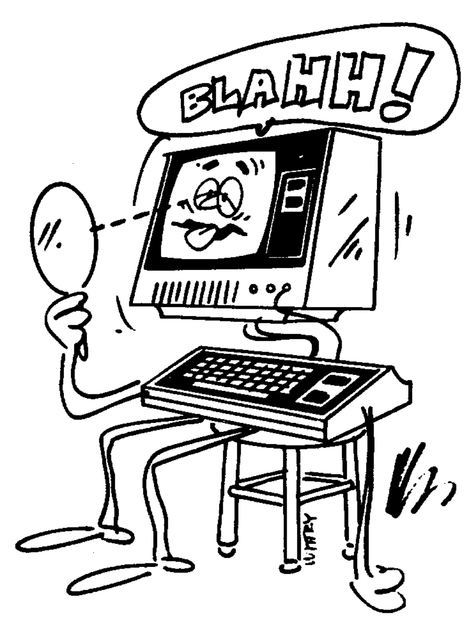
Take a Look at Yourself. . .

Are your eyes starting to feel like you've been looking at the classified ads for half an hour? That's normal, and indicates you're well on your way to becoming an expert programmer.

What? You don't care about appearances—you want something deeper? Well, then you're ready for:

UFO #1-1
(that's short for Useful and Factual Observation):
The Video Display has room for 16 lines, with each line containing 64 characters. That's a total of 1024 character positions.

Feel better?



Surprise—Double Your Pleasure

You've earned your stripes, so how about a little relief for those eyes? Press (CLEAR)(if you haven't already) and then (SHIFT) (→).

This tells TRS-80 to display everything double-size (double-width that is, not double-height).

To get out of the double-size mode, simply press (CLEAR) again.

UFO #1-2
In the double-size mode, the Display has room for 16 lines of 32 characters.

Something strange? Well, just for fun, press (CLEAR) then type STUERBPORAIYSIES! Now press (SHIFT) (→). Surprised?

Thar's Gold in them thar Hills. . .

Now you know your way around the Keyboard and Video Display pretty well. But before going on, play around with the Keyboard a little more, until you discover a few little goodies we left for you to find:

Exercises (Do It Yourself #1-1, 1-2, 1-3, 1-4)

1-1. How to produce the "↑" character (on some models, "[" is displayed).

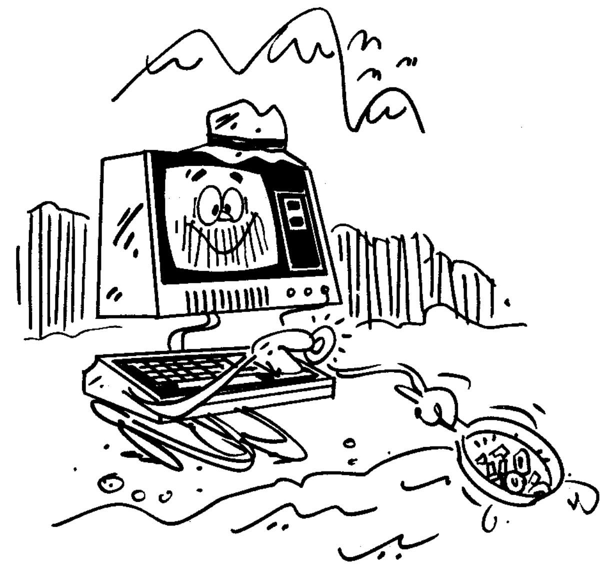
1-2. How to make the cursor jump down to the next line before it reaches the end of the present line.

1-3. The difference between the letter O and the number 0 (zero).

1-4. The difference between " (quotation mark or double-quote) and ' (apostrophe or single-quote).

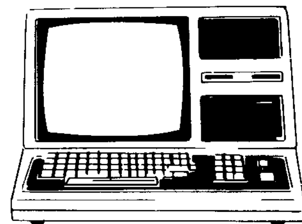
1-5. The difference between l (the lower-case letter L) and 1 (the number one).

If you can't discover all these things after playing around a while, see the answers in Appendix A.



☑ Chapter Checkpoint #1

1. The cursor always tells you:
   a. where the next character will be displayed
   b. when to clear the display
   c. how to get to Florence, Mo.
2. The symbol for the "prompt" is ___.
3. If you want to cancel a letter (or any character):
   a. give it to the postman
   b. press (BREAK)
   c. press ⊖
4. By pressing (   )(key), you can clear the display.
5. If you only have 32 characters per line, you must be in the ______-______ mode.
6. You can erase your line and start over by pressing (   ) and (   )
7. If you want to jump the cursor ahead quickly, press (   ).



For answers, see Appendix A.

Chapter

Two

What to Do Before You Press (ENTER)

Up to now, you haven't asked the Computer to do much. Sure, it's been "listening" to the Keyboard, and "echoing" what you type onto the Display. The Computer has even erased the entire Display a few times.

But now it's time to put this 5-lb (2270 gram) beast to work!

Let's make TRS-80 say something friendly, like

```
HELLO, I'M YOUR TRS-80 COMPUTER!!
```

First press (BREAK). This tells TRS-80 to ignore the line you've been typing in and start a new one. The Computer will respond with another prompt:

```
>
```

Now type:

```
NEW
```

and press (ENTER). This makes sure you're off to a fresh new start after all that experimentation in the last chapter. The Computer will clear the screen and display the familiar READY heading and prompt.

Now for our first real command. Type in the following line (but don't press (ENTER) yet):

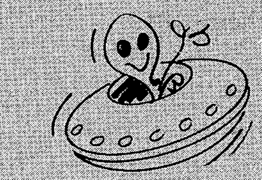
```
PRINT "HELLO, I'M YOUR TRS-80 COMPUTER!!"
```

Double-check what's on the Display to make sure your typing is up to snuff. Use the backspace key ⊖ to erase any errors.

Everything okay? Good.

Small is Beautiful. IBM's first large-scale Computer, Mark I (1941), weighed 5 tons and couldn't even say READY!

A Few Words about Spaces...
In general the Computer ignores them, unless they are in between quotes. So don't worry about counting spaces—just concentrate on the other letters and numbers!



UFO #2-1.
The TRS-80 will not pay attention to your typing until you press (ENTER). When you do, it will try to figure out what you want and DO IT NOW—immediately.

"Immediate" is a word we'll use over and over to describe any instruction which is performed "right now" but which **will not** be stored for future repetition. (The Immediate Mode, in other words, is a lot like money—it vanishes as quickly as it appears!)

Now press (ENTER).

Did it work? If so, the Display will look like this:

```
PRINT "HELLO, I'M YOUR TRS-8Ø COMPUTER!!"
HELLO, I'M YOUR TRS-8Ø COMPUTER!!
READY
>
```

If the Computer did not print the message (maybe you got an error message instead), try again. Type PRINT and then the message inside " ". Double-check what you've typed in; then press (ENTER) when everything's okay.

Once you succeed with this command, try putting other messages inside the quotes. Do it this way:

PRINT "*your message goes here*"

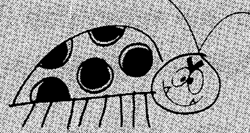
and press (ENTER) when you want the Computer to look at what you've typed in.

Now let's try printing a string of 25 asterisks on the Display. Type:

```
PRINT "*************************" (ENTER)
```

UFO #2-2
With PRINT instructions, you can put anything you want inside the " " (except another set of double quotes), and it will be printed exactly as you typed it.

Throughout this book, our little friend here will point out "Error Messages" that sometimes occur when you make a mistake.



As you can plainly see, he's a BUG; to computers, "bugs" are those little problems you have in communicating with the machine.

Haven't you heard the old proverb, "Computers are never wrong . . ."? Well, that's the case here and this is the way TRS-80 tells you that you've done something wrong somewhere down the line. And speaking of lines . . .

TRS-80 even tells you exactly which line contains the mistake. Furthermore, TRS-80 goes so far to tell you what kind of mistake you've made. Now that's getting your money's worth . . .

We'll go into more detail on Error Messages later in this same chapter.

Are you using the double-size mode? If not, try it. Press CLEAR and then SHIFT ⊖.

So It's a Great Message Printer—Big Deal...

The old 5-ton Computers earned their fame as great number-crunching machines. Let's see how your 5-pound "micro" handles its numbers. Just to be safe, we'll start out with something simple. How about the age-old question,

1 + 1 = ?

First make sure the Computer is ready for a new command.

Command? That's right—you're the boss, so *you* tell the Computer what to do.

Remember how to get the Computer ready? Just press BREAK so the last line displayed looks like this:

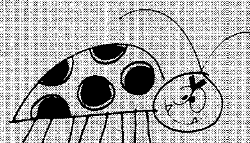
```
>
```

Now type in the following instruction:

```
PRINT "1+1=" ENTER
```

(If you get an error message, type in the line again... carefully... double-check it, and press ENTER.)

Excellent—the Computer has stated the problem very well. But why didn't it solve it? Remember UFO #2-2. **Everything inside the double-quotes is printed exactly as-is.** So how do we make the Computer give us the answer?



?SN is the most common ERROR message that will bug you on a PRINT ENTER situation. (In fact, it's one of the most common ERROR messages in any situation.) It usually just means you have made some mistake with punctuation or spelling. Check whatever it was you wanted to PRINT, correct the error and try again.

Can you guess what command will do it?

In case you typed 1 + 1 in and nothing happened, don't be surprised. You didn't break anything. To the Computer, 1 + 1 is just so much graffiti. Remember, you have to make it very plain that you're commanding TRS-80 to do *something*. (Hint: Add a "verb" to make a complete command...)

Don't Quote Me, But . . .

In English, quotation marks are often used to signify material that is to be taken literally, letter-for-letter, verbatim, with no changes or interpretation allowed. When quotation marks are omitted, the material is subject to interpretation.

The situation is similar in a PRINT statement. PRINT "1 + 1" involves no interpretation. As far as BASIC is concerned, you might have said PRINT "ONE PLUS ONE"—neither tells BASIC to interpret or solve the problem of 1 + 1. But leaving off the quotes tells BASIC to evaluate the information or solve the problem. So try:

```
PRINT 1 + 1 (ENTER)
```

Smart computer, huh?

Notice that we don't include the equals sign:

```
PRINT 1 + 1 =
```

is an error, since the equals sign isn't really part of the problem. But BASIC wouldn't know that.

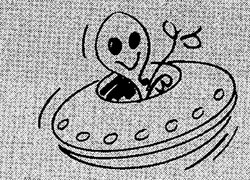
Here's one you can't do in your head

Long division is short work for the TRS-80. For example, what's 17893 divided by 14.5? Type in the following line to find out. . .

```
PRINT 17893/ 14.5 (ENTER)
```

The slash symbol / means "divided by". You've probably seen it used this way before, as in the simple fractions 3/4, 1/2, 9/10, etc.

By now, you've already discovered:



UFO #2-3.
You can make the Computer do calculations by PRINTing the problem without quotation marks.

Print Lists

Suppose you want to make a list and print several items on a single line. For example:

17893 / 14.5 =
followed by the answer.

There are two ways you can do this, each giving you a different kind of listing.

The first way is to put the message and the problem in the same PRINT instruction but to place a semi-colon (;) after each item:

```
PRINT "17893/14.5="; 17893/14.5 (ENTER)
```

You can even extend this list by using more than one semi-colon:

```
PRINT "17893/14.5="; 17893/14.5; "ACCORDING TO TRS-80" (ENTER)
```

The second way is to use a comma (,) instead of a semi-colon:

```
PRINT "17893/14.5=", 17893/14.5 (ENTER)
```

The difference between the semi-colon and comma is that the semi-colon bunches the numbers up together, while the comma causes the next item to be printed in a pre-determined "print zone."

The two punctuations can even be used together:

```
PRINT "17893/14.5="; 17893/14.5, "17893/7.25=";
      17893/7.25 (ENTER)
```

UFO #2-4
To PRINT more than one item on a single line, simply list the items after PRINT, putting either a semi-colon or comma after each item but the last.

The semi-colon will print the next item right after the previous one, while the comma will print the next item in the next 16-column "print zone." **Note**: PRINT will be covered in detail in a later chapter.

Each line is divided into four "print zones." (Do you remember how many spaces there are in a line? If you said "64" you're right! Therefore, each "print zone" contains 16 spaces.) Each comma in a print list moves the next item over to a new print zone.

Some of our computer lines are too long to fit on a single line in this manual. In such cases, we'll break the line at a convenient spot and put the remainder on the next line with a little extra indentation. You should type in the line as if it were all on the same line.

Example (Do It Yourself #2-1). Type in and PRINT the different combinations of the list "1, 2, 3, 4" to see how TRS-80 interprets your information. Use semi-colons, commas, spaces as well as different combinations of each.

You've Got the Keys to the TRS-80 Lab...

So go ahead and experiment with what you've learned. Try to get the Computer to print various other arithmetic results, using addition (+), subtraction (−), division (/) and multiplication (*).

The * asterisk means "times" to the TRS-80 and most other computers. For example, 5*2 means "five times two".

Projects (Do It Yourself #2-2, 2-3, 2-4, 2-5). Make the Computer:

2-2. Give the decimal version of some simple fractions such as 1/3, 2/3, 7/8, 15/16. For example, 1/4=.25

2-3. Do more complicated problems such as 83 + 11/21 − 33.7

2-4. Use large numbers, such as 12345000 * 70010.101 (If you use *really* large numbers the result may throw you, but try it anyway.)

2-5. State the problem and then compute the answer! For example, make the Computer print:

```
100 + 200 = 300
```

(Hint: UFO #2-4).

See Appendix A for our answers.

Now, About those Error Messages...

If you've been going through the examples in this book, and also done some experimenting on your own, you're bound to have run across an error message or two. That's when the Computer doesn't do what you wanted, but instead prints a question mark followed by an abbreviated message.

A very common message is:

```
?SN ERROR
```

Obviously, the computer is warning you that you've committed a sin!

For example; if you type:

```
PRRINT "HOPE THIS IS OKAY" (ENTER)
```

the Computer will reply:

```
?SN ERROR
```

What's that—a SiN error?

Your "sin" was a harmless mistake, but to the Computer, it's a big deal. PRRINT isn't in its vocabulary, and it doesn't know you meant PRINT.

See, it's not so smart, after all!

Does the Word Sin Bother You?

If it does, then take SN as an abbreviation for "syntax" instead. Syntax is like grammar—minding your spelling and punctuation. When you make a spelling, punctuation, or usage mistake, you've committed a syntax error.

There are lots of other wrong things you can do. For example, type in:

```
PRINT 5/Ø (ENTER)
```

The Computer will reply:

```
?/Ø ERROR
```

Well, what did you expect? Division by zero is illegal, even with a liberated personal computer in the privacy of your own home!

Googol-itis

Ask the Computer to print the number 1 followed by 100 zeroes.

```
PRINT 1ØØØØØØØØØØØØØØØØØØØØØØØØØØØØØØØØØØØØØØØØØØØØØØØØ
ØØØØØØØØØØØØØØØØØØØØØØØØØØØØØØØØØØØØØØØØØØØØØØØØØØØØ (ENTER)
```

The Computer will reply:

```
?OV ERROR
```

This number (a googol) is too large for the Computer to handle. So it gives you an overflow message. People get heartburn; Computers get Overflow.

There are lots of other error messages; many of them are associated with particular BASIC instructions, while others can happen anytime, anyplace, to anyone...

In this book, we'll be explaining error messages where we think you'll encounter them. If one of your experiments causes an unfamiliar error message, simply double-check what you have typed against the examples we've given. If you can't find anything wrong, then either:
1. Look up the error message in Appendix C, or
2. Sidestep that experiment and get on with the show!

Stay Tuned for these Exciting Episodes...

You probably think you can solve most of the world's problems with what you know about PRINT but wait! The PRINT you know has more than a dozen smarter cousins, and scores of distant relatives who are just as powerful in their own right.

For example, there's an instruction to clear the Display (it's similar to pressing the (CLEAR) key). Type the following:

That's BASIC for CLear the Screen.

```
CLS (ENTER)
```

For another example, remember when you got the Computer to print a string of 25 asterisks? You had to put the 25 asterisks inside the quotation marks.

Now try this—but careful with the punctuation:

Notice the $ after the word STRING.

```
PRINT STRING$ (25, "*")
```

Now press (ENTER).

If you typed it in right, the Computer will display a row or "string" of 25 asterisks.

The idea of a string of characters is special in BASIC and other computer languages. We'll be explaining it in detail later on, but for now, just play with it a little!

How about this one:

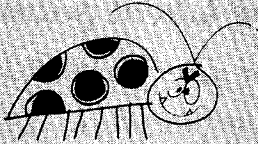
```
PRINT STRING$ (10, "0") ENTER
```

Hint: The first number in the parentheses says how long the string will be; the second item (in quotes) says what character to use in the string. You can try any string length number from 1 to 50.

Now for something *really* different:

```
PRINT STRING$ (40, 149) ENTER
```

The number 149 tells the Computer to print a special character that you can't type in from the Keyboard. Try any number up through 191.



You can run across a couple of bugs whenever you work with strings (and they'll really tie you in knots if you let them).

?OS is the most common error. It simply means you're "out of string space" and occurs whenever your string exceeds 50 characters.

?LS is the other and TRS-80 is just telling you the string is too long.

Later on, we'll show you how to make the Computer display strings of up to 255 characters.

Computers Never Die, They Just Scroll Away...

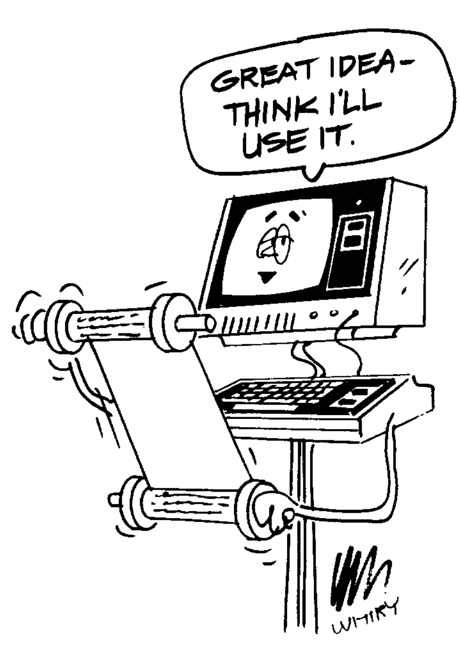
You've probably wondered (if not, do it now...) about what happens to your instructions and the Computer's messages after the Computer finishes with them.

For example, when the cursor gets to the bottom line of the display (it can't go any lower), it makes room for more messages by pushing everything up a line. The top line of the Display disappears off the top of the screen, and every other line moves up. This is known as **scrolling,** because it's just like reading an ancient scroll.

But Does TRS-80 Remember?

By now, you've gotten the Computer to do lots of printing and computing. *You've* learned a lot, but has the Computer? Can you make the Computer repeat what it's already done, without having to re-type all those instructions?

It *would* be nice . . . But before going on to the answer in the next chapter, this might be a good time to take a break. You've learned quite a lot already — enough to put together some grand plans. So, mull it over . . .



☑ Chapter Checkpoint #2

1. To get a new start in (BASIC) life, type in______.
2. ?SN, ?/0, and ?OV are all________ messages.
3. To enter anything into TRS-80, you must first press [   ].
4. By pressing [   ], you can tell TRS-80 to ignore the line you've been typing in and to begin a new one.
5. Use the __________ command to instruct TRS-80 to display any messages or calculations you have in mind.

Chapter

Three

Re-Usable Instructions (Programs)

So far, all your instructions to the Computer have been immediate. . .just like saying, "Do it now!" Each time you press ENTER, the Computer looks to see what you've typed in and tries to do what you asked it to. Once the Computer has completed the instruction, or given you an error message, it forgets about that one and waits for the next instruction. The only way to repeat an instruction is to re-type it and press ENTER.

Programs? Already? But I haven't even learned about CPU's, I/O's, RAM's, ROM's and BVD's yet! This really is easier than I thought!

That's No Way to Run a Computer!

Hold it, don't pack up your Computer and head for the local Radio Shack! There *is* a way to make your Computer remember one instruction or a whole sequence of instructions. In fact, 99 percent of the time, you'll probably give the Computer these reusable instructions. . . or "programs", as they are called.

Make sure > is the last line displayed on the screen (press BREAK if it's not).

Let's start out by clearing your previous experiments. Remember how to get off to a NEW start? (The answer is in the right margin.)

Type
```
NEW ENTER
```

Now type in the following line (but *don't* press ENTER yet):

```
10 PRINT "HI! I'M YOUR TRS-80 COMPUTER!!"
```

(That's the number 10 followed by a space, followed by the PRINT instruction we used at the beginning of Chapter 2.)

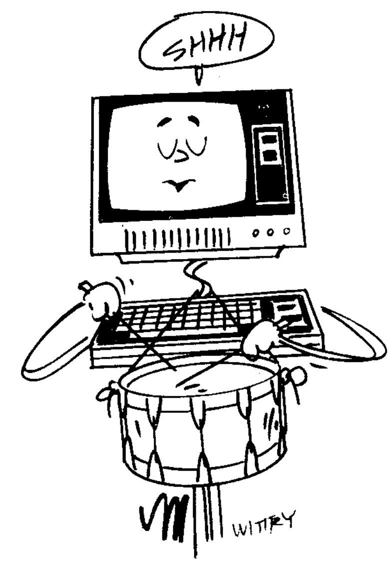
Check the line you typed in to make sure everything's okay. If it's not, backspace to the error and correct it.

A Light Drum Roll, Please

Now press (ENTER) and see what happens.

Nothing happened! Is the Computer tongue-tied? But it worked before (without a number before PRINT)...

Something really *did* happen, though. When you pressed (ENTER), the TRS-80 looked at what you had typed in. When it saw the number 10, it said: "Aha! This is no ordinary **immediate** (do-it-now) instruction. This is a **re-usable** instruction." And so the Computer tucked the instruction away in its memory for use whenever you ask for it.

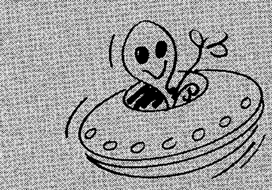


So How Do You Tell the Computer to DO IT?

Your PRINT command is now a one-line program in memory, waiting to be performed, "done", or..."run". So type the following:

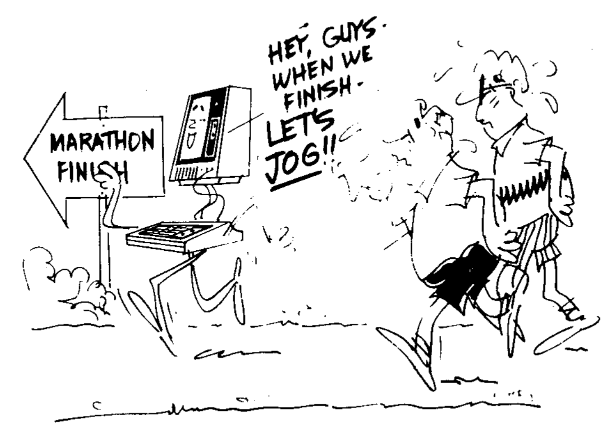
```
RUN (ENTER)
```

That's more like it!



UFO#3-1.
RUN tells the Computer to do, or "execute", the program that you've stored in memory.

"Execute" sounds more important, so we'll use that word. (Your programs *are* important, aren't they?)



See the Computer RUN

Now type RUN (ENTER) again...and again...You'll find that you can't wear out the Computer *or* the program.

The Acid Test

Press (CLEAR) and then press (ENTER). Now your PRINT program is nowhere to be seen. But it *is* in the Computer's memory. Prove it? Just tell the Computer to LIST it, by typing:

```
LIST (ENTER)
```

UFO #3-2.
LIST tells the Computer to display whatever program lines it has stored in memory.

Adding to the Program

Type in the following line:

```
5 CLS (ENTER)
```

(That's the number 5 followed by a blank followed by the CLS statement from Chapter 2.)

Now type in one more line:

```
20 PRINT "MASTER, WHAT'S YOUR NAME?" (ENTER)
```

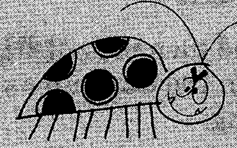
LIST the program. The Computer will display:

```
5 CLS
10 PRINT "HI! I'M YOUR TRS-80 COMPUTER!!"
20 PRINT "MASTER, WHAT'S YOUR NAME?"
```

The TRS-80 has been busy! While you weren't looking, it added your two lines to the original one-line program, and it *added them in order,* according to the numbers at the beginning of each line.

TRS-80 is an all-around orderly beast. Later, you'll learn how this marvelous microcomputer even puts letters and words (among other things) in order as well. "What order?" you ask. Well, just about any order *you* want. Remember, with TRS-80, you're the boss.

UFO #3-3.
Every line in a program has a number, called . . . you guessed it . . . the *line number*. In general, the Computer will execute the lines in a program in order, and stop when it completes the last line. You can use any line number from zero to 65529.



It shouldn't happen too often, but if you ever try using a "line number" above 65529 our little friend will bug you with a ?SN error.

Go ahead and RUN the three-line program that is now stored in memory. Type:

```
RUN (ENTER)
```

The Computer will clear the Display (that's what line 5 tells it to do), then print a greeting (according to line 10), and then ask you a question (line 20).

```
HI! I'M YOUR TRS-80 COMPUTER!
MASTER, WHAT'S YOUR NAME?
READY
>
```

Well, what are you waiting for? Type in `MY NAME IS. . .` followed by your name, and press (ENTER)

```
?SN ERROR
READY
>
```

Hmmm. Another one of those. But I know this one's spelled right! (Anyway, how would the Computer know the difference?)

Then why doesn't the Computer accept your name as an answer to its own question?

The Hard Truth

The Computer isn't smart enough. IT CAN ONLY DO WHAT YOU TELL IT TO DO. You told it to ask the question, but you didn't tell it to accept your answer. When the Computer finished your program and displayed the message:

```
READY
>
```

it was expecting you to enter another instruction (like RUN, CLS, LIST, PRINT, etc). It wasn't prepared to learn your name.

(Stop a minute and let this idea sink in: THE COMPUTER CAN ONLY DO WHAT YOU TELL IT TO DO. *Every line of every program you write should be written with this in mind.* Don't ever expect your Computer to "know what you mean". It won't. You've got to use BASIC to spell out exactly what you want done.)

In a later chapter, we *will* "teach" the Computer to learn your name, and even to carry on a friendly conversation with you.

Someone has said that a Computer is just an idiot. A fast idiot, but an idiot nonetheless. Don't tell your TRS-80, though. After all, you don't yet know if it (he? she??) has feelings!

Another side of this rule: YOU have got to know exactly what you want done before you can communicate it to the Computer.

Subtracting from the Program (Deletion)

There'll be many times when you not only want to add a line to an existing program, but subtract (or delete) one as well.

Do you still have your program?

LIST it and see (if not, take a minute and type it in).

```
5 CLS
10 PRINT "HI! I'M YOUR TRS-80 COMPUTER!!"
20 PRINT "MASTER, WHAT'S YOUR NAME?"
```

Now let's say you need to delete line 10 so the program consists of just lines 5 and 20.

It's simple. Type:

```
10 (ENTER)
```

Then LIST your program.

What happened? Well, whenever TRS-80 encounters a line number that has already been used in an existing program it deletes the first line and replaces it with the second. And, in this case, TRS-80 replaced line 10 with *nothing*. (You can correct mistakes the same way. Just type the line number and the corrected statement and (ENTER). Later on, you'll learn how to delete a single letter or word by EDITing it instead.)

How to Tell the Computer, "Forget it!"

With what we've covered so far, you can already write some interesting programs. But before you do, you've got to get rid of that one that's now in memory. (Type LIST (ENTER) to see that it's still there.)

Remember how we started the chapter? Type in the command that erases everything . . .

```
NEW (ENTER)
```

After the dust settles, look at the Display.

```
READY
>
```

So you cleared the Display, but what about the program? It may still be hiding in memory somewhere.

Well, how can you prove that the program is gone? Command TRS-80 to tell you what's in there. Remember how?

Type LIST (ENTER). What did it list? Nothing! That means there isn't any program in memory.

UFO #3-4.
NEW tells the Computer to erase any program it has stored in memory. To use that program again, you'll have to re-enter it.

Now that you've eliminated the old program, you can put another one into memory. Here's where you get to do some creative programming.

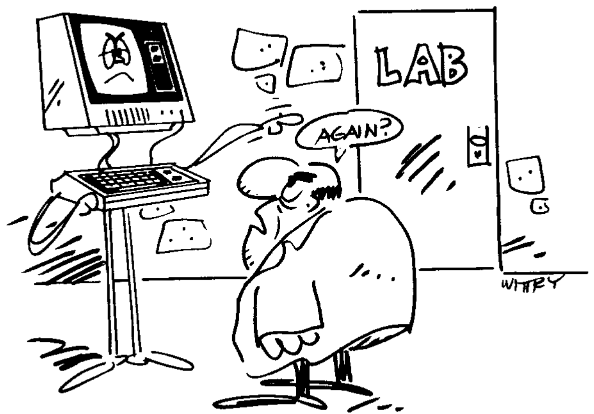
Back to the Laboratory, Igor!

Program (Do It Yourself #3-1). Write a program that clears the Display, prints a line of asterisks, and then prints the message, "I LIKE STARS". Use the blank lines below to write down your program. (Notice we've already supplied the line numbers. All you have to do is add the necessary BASIC statements.)

10 ____

20 ____

30 ____



After you've written out the program, check it for misspellings, punctuation, etc. Then carefully type the program into the Computer, pressing **ENTER** at the end of each line. List your program and re-type any lines which contain errors.

Now RUN it.

Did it work? If so, then it's time to tell your family and friends that you're involved with a Computer. 'Cause once you've succeeded in writing and running your own computer program, the relationship can get very "heavy". It may be quite a while before you want to re-join the everyday world. And *someone's* got to make sure you eat and sleep!

Of course we counted our line numbers by tens! What good's a one in these days of inflation??

Seriously, there *is* a very practical reason for counting by tens, fives, twenties, or hundreds—*anything* but ones. You'll see why in the next chapter.

If It Didn't Work, Relax.

It may be a few minutes before you have to say goodbye to normality. The most likely error message will be:

```
?SN ERROR IN 10
READY
10
```

We've used line number 10 just as an example but TRS-80 will display the number of the line which contains the error.

If this happens, simply press **ENTER** and the Computer will return with a

```
READY
>
```

Now LIST the program, and look closely at the line which the Computer says contains an error. There's probably a spelling error, missing punctuation, or extra punctuation.

Once you've located the error, simply re-type the line correctly. First type the line number, then the statement. When you've got it right, press (ENTER). Now the corrected line will replace the old line, since both share the same line number—and physics tells us that no two bodies can occupy the same space at the same time...

Later on, we'll tell you about a much niftier way to correct errors, using a built-in Edit function. Imagine — you'll be the editor-in-chief of an important micro-computer daily!

List the corrected program (LIST (ENTER)) to be sure everything's okay.

Now RUN the program.

As a Last Resort:

If it still doesn't work, look up the answer to Do It Yourself #3-1 in the Appendix.

Remember, our answer is just one of *many* ways to do it.

Program (Do It Yourself #3-2). Write a program to print the simple fractions 1/2, 1/3, 1/4, 1/5, 1/6, 1/7, 1/8, and 1/9, in decimal form. (For example, .5, .333333, etc.) If you want to get fancy, make the Computer print an equation for each fraction, as in:

```
1/2=.5
```

Write down the program in the space below. Use the first line in your program to clear the display, and the remaining eight lines to print out the answers. This time *you* pick the line numbers. The only rule is: make sure to number the lines from lowest to highest.

____

____

____

____

____

____

____

____

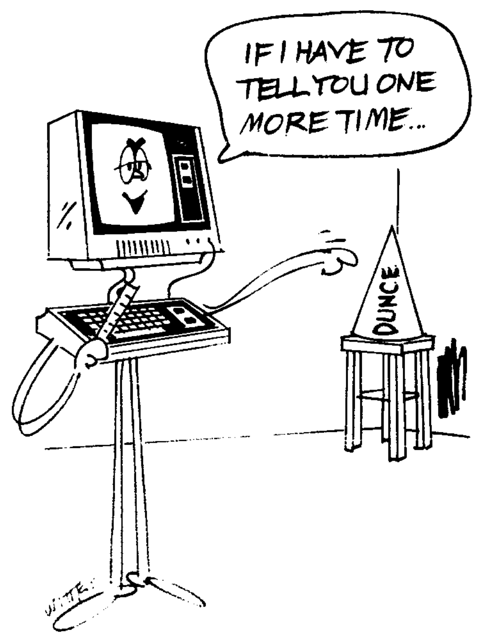
____

Double-check what you've written down. Type NEW to erase any old program, then type in the program, pressing ENTER at the end of each line. List the program, re-type any incorrect lines, then RUN it.

If the Computer says you have an error, try to handle it by yourself. After all, we've already told you how *once* (after Do It Yourself #3-1), and *you* don't need to be told *twice*, do you?

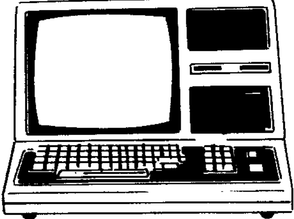
Here's a good chance to practice SAVEing your program on tape. (Check your Owner's Manual for specific directions on how to do this.)

Anyway, this is your first real program and you'll probably want to put it in a scrapbook somewhere along with your bronzed baby-shoes and that old high school yearbook.



☑ Chapter Checkpoint #3

1. Programs are nothing more than re-usable ______________.
2. The one thing programs, laundry, and groceries have in common is that they can all be ________ed.
3. Every line in your program must begin with a ______ number.
4. In order to execute a program you first have to:
   a. JOG
   b. RUN
   c. FIND A REPUTABLE HIT MAN
5. To delete one line from your program, you must re-type the ________ ______ and press (ENTER).
6. Choose the best method for inserting a new line into an existing program.
   a. type in NEW and start all over
   b. hire an out-of-work programmer
   c. type in a new line number between two existing lines; then type the addition, then press (ENTER).



Part II
BASIC Training

- Statements • Shortcuts
- Editing • Math Functions
- Arrays • Strings

Chapter

Four

BASIC Training (Crew-Cut Optional)

How're you feeling? Sharp as a sponge and fresh as a tack? Great — because it's time to get down to the basics of BASIC, and that means . . . new words, ideas, insights, discoveries, and even a little creative thinking! Maybe you ought to run down to the local diner first, and have a Brain-Food Plate (that's Harvard beets and boiled fish).

Okay, Here Goes . . .

So far, we've used the terms **instruction, command,** and **statement** to refer to words that your TRS-80 understands. But now let's clean up our language a bit so we can get on with the more interesting and practical aspects of programming.

From now on, we'll use the word **statement** to mean **a complete BASIC instruction.** Most statements can be used in the immediate mode (without line numbers) as well as in programs. The statements we know so far are: PRINT and CLS.

We'll use the word **command** to describe those BASIC statements that you would normally **use outside of a program,** in the immediate mode. For example, RUN, LIST and NEW are commands. For special effects, you can put them in programs (see margin). But normally you wouldn't.

BASIC Statements—A Survey

There are several kinds of statements you can use to tell TRS-80 what to do. The major kinds are:

Name-it Storing information for instant recall when you need it (we'll cover some of these in this chapter)

Three Abnormal Programs

We said that commands are normally used outside of programs, in the immediate mode. For those of you who are fascinated by the abnormal, here are some strange critters.

Self-Destruct

Type in the following program.

```
10 NEW
```

RUN it. Now try to list it.

Pride will get you nowhere

```
10 LIST
```

This program thinks it's the greatest invention since the battery-powered banana-peeler.

A Compulsive Runner

Look out for this one.

```
10 PRINT "I LIKE TO RUN"
20 RUN
```

Type RUN to start this program on its endless journey. When you want your Computer back, hold down the (BREAK) key until the Computer displays the message:

```
BREAK IN ##
READY
>
```

The BREAK message means, "I was about to do line ## when you forced me to take a break."

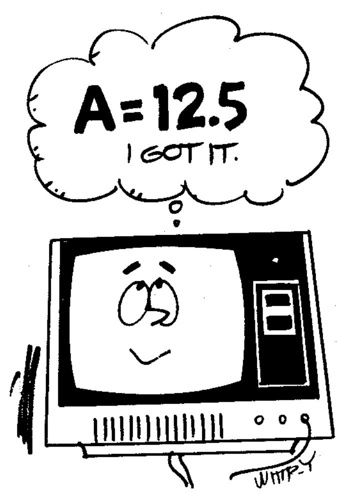
Input/Output Getting information in and out (I/O for short.)

Graphics Doing special things to the Display.

Program Flow Changing the order in which program lines are executed.

Declare-it Changing the Computer's "assumptions".

Name-it Statements



The statement:

A=12.5

tells the Computer to copy the number 12.5 into a place in memory, and to name that place "A". *(Well, it's not Acapulco, but at least it's easy to spell!)* Then, whenever you want to recall that number, you simply refer to it by its name, "A".

Go ahead, if you haven't already, type in the statement,

```
A=12.5 (ENTER)
```

Not too exciting a result, was it? But something *did* happen inside the Computer. To prove it, tell the TRS-80 to print the "contents" of A, by typing:

```
PRINT A (ENTER)
```

You just told the Computer to look in memory for a place called A, and tell you what is stored there.

What's In a Name?

Just what is this thing called "A"? We've said it names a place in memory, and that this place is now storing the number 12.5. But why bother? Why not just refer to the number 12.5 and skip the name?

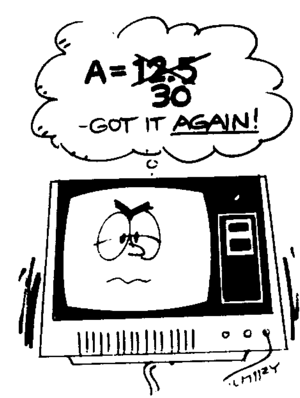
Well, suppose 12.5 is the number of goldfish in the Average American Goldfish Family (AAGF). This number is bound to change, depending on prevailing

conditions in the bowls and ponds across the land. And whatever that number is — 12.5, 13, 5 or 50, we would like to store it where we can find it.

That's where A comes in. Because A doesn't have to contain 12.5. *We can put any number there*. For example, when the number of goldfishes in the AAGF zooms up to 30, we can type:

```
A=30 (ENTER)
```

to tell the Computer to replace 12.5 with 30.



Type:

```
PRINT A (ENTER)
```

to see that the contents of A really *has* been changed to 30. Since its contents can vary, we call "A" **a variable.**

No More Food for Thought, Please

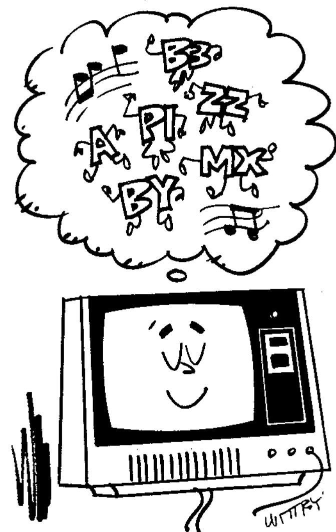
"Contains" and "contents" make us think of milk cartons, cereal boxes and other food-and-drink containers — which tends to get us off the subject. So we use the words "equals" and "value" instead: A *equals* 30, the *value* of A is 30, etc.

UFO # 4-1.
The simplest name-it statement looks like this:

variable = number

Such a statement puts the *number* into a place in memory, and names that place *variable*. For convenience, read "A= . . ." as, "Set A equal to. . ."

How Many Variables? (Can Dance in the Head of a TRS-80)



We chose the letter "A" as a variable name purely by chance. There are hundreds of other names you can choose for variables, and **each different variable can contain a different number.**

For example,

```
A1=1000
C =33
J3=49
XX=33
```

are all okay as name-it statements. In each case, **the number on the right of the equals sign is stored in the variable on the left.**

UFO # 4-2.
Variable names must begin with one of the letters A through Z and may be followed by another letter or one of the digits 0 through 9.

For example, the following are all acceptable and distinct variable names:

A	A1	AB	AZ
B	BA	BO	BY
C	C1	CB	CA

The only restriction is that you can't use one of BASIC's "special words" in a variable name. (We'll cover these later, but for the time being, just don't use TO, IF, or ON as variable names.)

Variable names can be longer than two characters, but only the first two will be used by BASIC. For example, SU and SUPER are the same to BASIC.

Quick—Name Four Famous Variables!

Just to be sure you've got the idea, find four acceptable variable names in the list below:

AM	1Y	LO	Z	0E
8Q	5	WC	1A	17

Got all four? Good. Check the list in the margin to be sure you got them right.

If you didn't get all four, re-read UFO 4-2 and the examples above.

Answer:

AM LO Z WC

Why Use Variables?

Having numbers in memory means you can do all kinds of number-juggling and crunching without having to refer directly to the number.

Type in the following:

```
A=1 (ENTER)
B=10 (ENTER)
C=1234 (ENTER)
```

You have now stored the three numbers in memory. (If you're not convinced, use our friend PRINT just to be sure.)

Now try these Print statements:

Don't forget to press (ENTER) after each statement.

```
PRINT A+B
PRINT C-B+A
PRINT B*C
PRINT C/B
```

Now you've performed several calculations, but your original three numbers are still in memory. . . for use again.

From One Variable to Another...

You just PRINTed several calculations on the Display. But suppose you want to save the **results** inside the Computer? Easy — just save the results in another variable. For example:

```
D=A+B
```

This statement says: "Add the value of A to the value of B, and copy the result into variable D".

Go ahead and PRINT D. While you're at it, PRINT A and B as well, to see that they are unchanged.

UFO #4-3.
The most general name-it statement has the form:

variable = expression

The Computer "solves" or evaluates the *expression*, and puts its value into the *variable* you named.

An expression is a sequence of variables, math symbols like + (plus) and − (minus), and numbers.
The following are all arithmetic expressions:

```
B+C
1+A
3*A/7
```

1 (any simple number can be used wherever an expression is allowed)

Take that, Miss Rutabaga!

The statements

```
A=1
A=A+1
```

make perfect sense to the TRS-80. Now what would your old grade school teacher have thought of that? What do YOU think of it? Try to predict what happens to variable A.

Check your conclusion by typing in the statements:

```
A=1 (ENTER)
A=A+1 (ENTER)
```

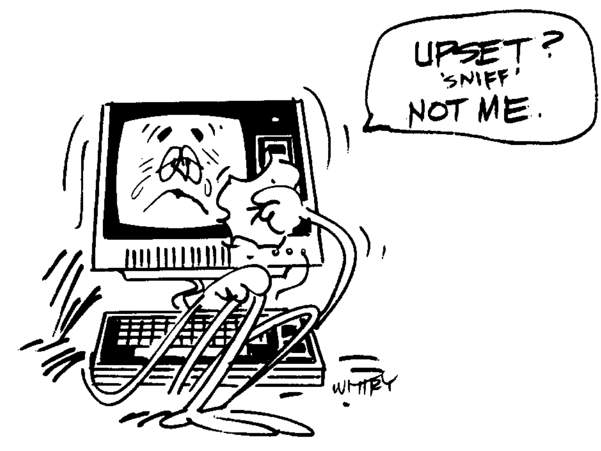
and then printing the value of A:

```
PRINT A (ENTER)
```

As you've probably discovered, when the same variable name is used on both sides of the equals sign, the old value of the variable is used in computing a new value.

Poets Have More Fun...

By now you know that the TRS-80 is a whiz with numbers — it can add, subtract, multiply, divide . . . and conquer them. (Every school kid should keep one in his pocket.) But what does the TRS-80 do for fun? Logarithms? quadratic roots? surds???



Well, why not give the old number-cruncher a tease by storing something clever in one of those variables? Type:

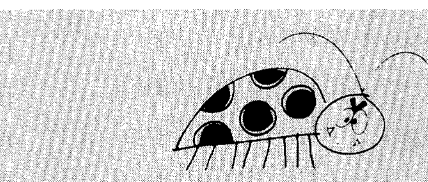
```
N1= "THE TRS-80 IS A SQUARE"
```

and press (ENTER).

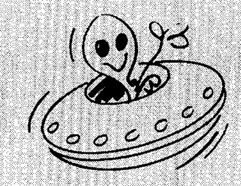
```
?TM ERROR
```

There — just as you expected. Instead of taking a little good-natured kidding, the Computer gets all upset and accuses you of violating its Trade Mark!

Well, let's not jump to conclusions. Look up TM ERROR in Appendix C. (You might as well get familiar with that section now, because you'll be running into these error messages more and more.)



If you looked the error message up, you know it means, "Type Mismatch Error". You tried to assign a string of characters (or **string** for short) to a variable which can only contain numbers. That's kind of like trying to put a square peg into the proverbial round hole — only harder.



UFO #4-4.
There are two kinds of variables — **numeric** and **string**. Numeric variables can store numbers; string variables can store sequences of characters, or "strings".

"THE TRS-80 IS A SQUARE" is a string value; it can't be added, subtracted, or otherwise "numberized" into submission. And it can't be stored in a numeric variable like N1!

Here are some other examples of string values:

"JOHNSON, B. Q. 3838 WINDING WAY"
"RED YELLOW BLUE GREEN ORANGE BLACK BROWN PURPLE"
"OCTOBER 12-------B'S BIRTHDAY"
"17 + 18 = 35"
"1 INCH = 2.54 CENTIMETERS"
">>>>>----->>>>>"

String values are almost always given inside double-quotes, so you and BASIC know exactly where the values begin and end.

With strings like these, who needs connections? If *only* you could store them somewhere...

Cheer up; you're about to stumble onto something big...

The Almighty $

They say money can buy just about anything ... so why not add "a little something" to our last name-it statement. Type in the following line (don't forget to add the $ after N1):

```
N1$= "THE TRS-80 IS A SQUARE"
```

and press **ENTER**.

Now type:

```
PRINT N1$ (ENTER)
```

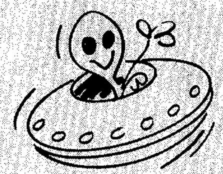
Money talks!

Now that you've oiled the TRS-80's palms a bit, see if it'll take another one. Type:

```
N1$ = "TRS-80 IS A SHAMELESS (ENTER)
HUSTLER" (ENTER)
PRINT N1$ (ENTER)
```

You can also read the $-symbol as a shorthand for "string". So, read "N1$= . . ." as "N1-string equals . . ."

Getting the idea?



UFO #4-5
Adding $ at the end of any variable name makes it a **string variable.** String variables can store any sequence of characters, including letters, punctuation symbols, numerals, and even spaces. (Everything, that is, except another set of quotation marks.)

Here are some examples of string variable names:

A1$ D$ LO$ KK$ B$ ZX$

More Good News

Adding the $ to a variable name actually creates *an entirely new variable* — and you can still use the old one (without the $) for storing numbers. For example, you can have:

```
A= 1001
```

while

```
A$="FRIED CHICKEN 1.98"
```

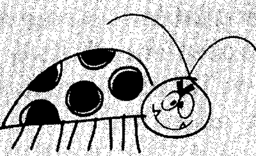
Input/Output Statements (I/O)

Remember when you programmed TRS-80 to ask your name, but it wouldn't accept your answer? Well, it's time to educate your Computer past the talking parrot level and teach it some manners by combining a name-it, a graphic, an output and now an input statement.

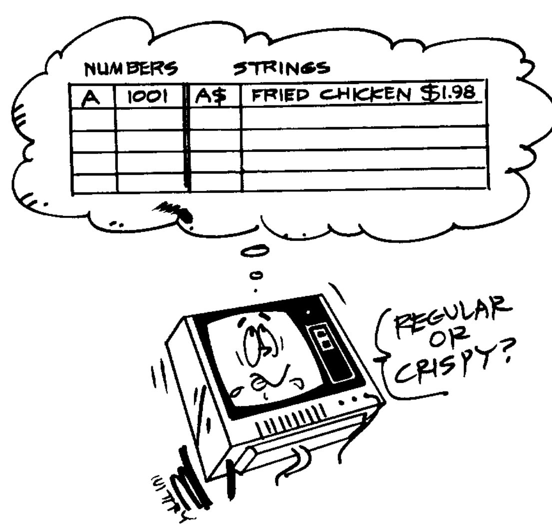
Type NEW, then enter this program.

```
10 CLS
20 PRINT "HI! I'M YOUR TRS-80 COMPUTER!!"
30 PRINT "WHAT'S YOUR NAME?"
```

Now, the trick is to tell the Computer to input your name, and to store it in a variable.



Whoa partner! Did you happen to get bugged while storing string values? If you did, it was probably an ?OS error which means you ran out of string space. (And if you didn't, consider yourself lucky and forge ahead.) If you did, type `CLEAR 100` (ENTER) and try again. More later on. . .



What kind of variable — string or numeric? Well, contrary to rumor, computers have not made obsolete all those good old names like DON, RICK, SYLVESTER, and PATRICIA. So we'll need a variable that can store **alphabetic info . . .** a **string** variable. How about N$?

Okay, so we've got a "place", N$, for your name. Now how do we make the Computer input your name into N$?

Here's another case where BASIC is like English. Just say, "Input my name!", by typing:

```
40 INPUT N$ (ENTER)
```

Hold it — don't type in your name yet. Remember, line 40 is part of a program stored in memory. The Computer won't actually *execute it* until you type RUN.

But first let's add another line to the program, to make the Computer show you it really has learned your name.

Type:

```
50 PRINT "HELLO ";N$;"!!" (ENTER)
```

Now list the entire program, making sure everything's okay. Retype any lines which contain mistakes. Got it right now? Then go ahead and RUN the program. (RUN (ENTER).)

```
HI! I'M YOUR TRS-80 COMPUTER!!
WHAT'S YOUR NAME?
?
```

The ? means the Computer has reached line 40, and now it's waiting for you to type in your name and press (ENTER). Go ahead and speak up! For example, if your name is "MOE", type:

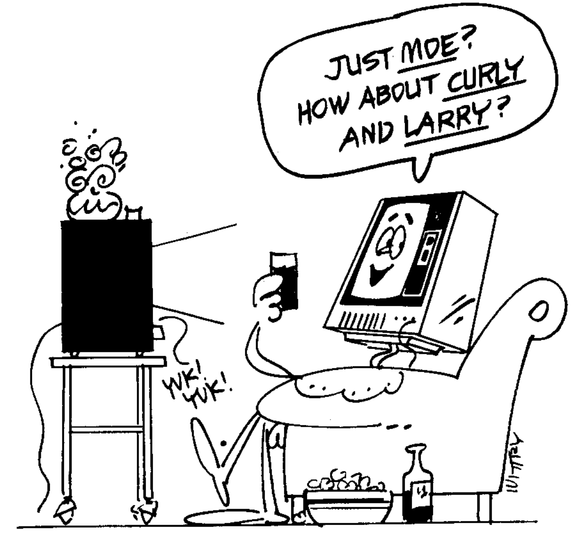
```
?MOE (ENTER)
```

Here's how lines 40 and 50 work:
When the Computer reaches line 40, it stops and waits for your answer. Just as soon as you press (ENTER), the name you typed in is stored in the variable N$.

It loses a little in the translation, but don't worry, to the TRS-80, it's pure poetry!

Hope you don't mind putting words in your Computer's mouth — 'cause that's the only way to get it to say much!

Be sure to type in a space right after HELLO and before the second ". You'll see why in a few minutes.



Line 50 tells the Computer to print the "canned" hello message, including a comma and a blank at the end; then to print the value of N$; and then to add a couple of !! marks at the end of your name. If you left out the blank space after the comma, the Computer's greeting would have looked something like this:

```
HELLOMOE!!
```

See how important the space is?

Program (Do It Yourself #4-1). Add to the "good-manners" program so the Computer next asks your age in years, and then tells you exactly how many days old you'll be on your next birthday (give or take a few days for leap years).

Hint: You need to make the Computer input your age and store this number in a variable (string or numeric?). If you're not sure where to go from here, then think about how *you'd* do it. "If I'm Y years old, how many days old will I be on my *next* birthday?"

Use these lines to write out D.I.Y. #4-1.

☑ Chapter Checkpoint #4

1. A complete BASIC instruction is called a ________. We've covered five of these. Can you name two of them? ________ and ________.
2. Which of the following is **not** a variable name?
   a. AL
   b. B
   c. 1L
   d. C2
3. TRUE FALSE (Circle one)
   Numeric variables store numbers.
4. TRUE FALSE (Circle one)
   String variables store string. (Of course not! But what do they store?)
   ____
5. I/O, a very important abbreviation, stands for ________.
6. A/An (Expression) (Graphic) is a sequence of variables, math symbols, and numbers. (Circle one)

Chapter

Five

BASIC Training, Continued

Remember Do It Yourself #3-2? That's where you wrote a nine-line program to print simple fractions in decimal form. No need to enter it again. Just look at this listing:

```
10 CLS
20 PRINT 1/2
30 PRINT 1/3
40 PRINT 1/4
50 PRINT 1/5
60 PRINT 1/6
70 PRINT 1/7
80 PRINT 1/8
90 PRINT 1/9
```

Repetitious, isn't it? *(How's it ever going to figure my taxes at this rate?)* Well, take heart: modern computer science has come up with A Better Way...

Using name-it statements, and a few other simple tools, you can do a whole lot more with just *five* program lines. Look at this sleek little beauty:

```
10 CLS
20 A=1
30 PRINT 1/A
40 A=A+1
50 GOTO 30
```

At the front of this book, we said the Computer's language was a lot like English. Well, go ahead... *read* through the program, using normal English in place of the BASIC words.

No, it's not Pulitzer-prize material, but don't let that discourage you. Perhaps it'll help to add a few imaginary exclamation points as you read through the program...GOTO!!!

Here's one reading of the program:

"(10) Clear the Display!

(20) Store into A the number 1.

(30) PRINT the result of 1 divided by the current value of A.

(40) Give A the new value A+1. (In other words, increase A by 1.)

(50) Go back to line number 30!" (Notice GOTO is **one** word.)

Whew! Now you see why the TRS-80 uses BASIC instead of English!

Try out the program. First be sure there's no program already in memory. (Remember how to do that?. . . Tell the Computer you want to do something NEW. . .)

Type NEW (ENTER) to clear out any program in memory.

Then type in lines 10 through 50 very carefully. After you've finished, LIST the program and double-check each line. If you find any errors, retype the line so everything's perfect.

Remember to press (ENTER) at the end of each line. We'll be leaving that off more and more, because you're probably in the habit by now.

RUN the program.

Whoa!! Did you see .5, .333333., .25, etc. go by? Press the (BREAK) key to stop the program and get another > on the bottom line of the Display.

"Think your way" through the program; perform each step, then go on to the next line. You'll find that there isn't any end to it! It just keeps going back to lines 30, 40 and 50.

In programming "circles," this is called a "loop."

Print Punctuation, Revisited

One reason the Display is scrolling faster than you can read it is that TRS-80 is printing just one item per line. You can't use a "print-list" in this program since the items are computed one at a time.

Remember when we introduced the semi-colon? We said it told TRS-80 that there's another item to be printed, but that's only part of the story.

Suppose you put a semi-colon at the **end** of a "print list", and didn't follow it with any more items to be printed. For example:

```
30 PRINT 1/A;
```

(Make this change in your program and RUN it.)

Now the computer will PRINT as many items as possible on a single line before beginning a new line. Here's what's happening:

UFO #5-1

When TRS-80 finishes PRINTing an item or a list of items, it automatically returns the cursor to the beginning of the next line. That way, each following PRINT will start at the beginning of the line. (This is usually desirable.)

But you can tell TRS-80 not to start a new line after a PRINT, simply by putting a semi-colon at the end of the print list. (We call this a "trailing semi-colon".) The Computer will then leave the cursor at that position and start printing there the next time it PRINTS.

Printing in Columns

Whenever you use trailing semi-colons in a PRINT list, TRS-80 prints the items one after the other, as close as possible. It does insert a single space after each number, and a space in front of positive numbers. (If the number is negative, a minus sign goes in the leading space.)

In your present program, that leaves you with a somewhat confused-looking display. No self-respecting typist would organize a table of numbers that looked like that; columns would look so much better . . .

Which leads us to that other handy piece of print punctuation, the comma.

Type in this **immediate** line:

```
PRINT 1/2, 1/3, 1/4, 1/5, 1/6, 1/7, 1/8, 1/9, 1/10
```

and then this one:

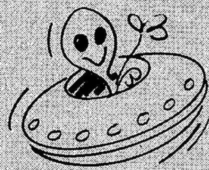
```
PRINT "ZONE 1", "ZONE 2", "ZONE 3", "ZONE 4"
```

Now change your program to use a trailing comma instead of semi-colon:

```
30 PRINT 1/A,
```

and RUN.

TRS-80 is quite a typist after all (to say nothing of a math wiz!).



UFO #5-2

You can use commas in a print list instead of semi-colons. A comma tells the Computer to skip over to the next "print zone" before continuing the list. The four print zones start at columns 1 (left margin), 17, 33, and 49.

Trailing commas work like trailing semi-colons in that both prevent the cursor from automatically returning to the beginning of the next line after a print. But when a comma is used, printing continues at the next print zone rather than at the next available position.

Note: If the cursor is already at the beginning of a print zone when the comma is encountered, it will advance to the beginning of the next print zone before printing the next item.

PRINT punctuation is just one of many tools available for making neat, orderly display output. We'll cover the other tools in a later chapter. For the time being, we can do quite a lot simply using commas and semi-colons.

Now for the Pause that Refreshes

There's a simple way to tell the Computer to pause so you can read the Display. Here's how:

After you type RUN (ENTER), quickly hold down (SHIFT) and @ together, until the Display stops scrolling upwards. When you want the Computer to continue, press any key. To pause the execution again, use (SHIFT) @ again.

Go ahead and try it — and be quick with the (SHIFT) @ keys! RUN and pause it a few times.

UFO #5-3.

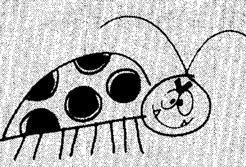
During program execution, you can force the Computer to pause by pressing (SHIFT) @. Press any key to continue execution.

Model 4 users: You can also press (F1) to pause program execution.

By the Way, Who Was that Masked Man? (GOTO)

GOTO was your first program flow statement. The TRS-80's normal routine is to do one program line after another until the last line has been executed. But line 50 says, "Forget the normal sequence and go back to line 30."

UFO #5-4.
GOTO *line number* is an "unconditional branch" statement. It tells the Computer to skip to the specified line, no matter where it is in the program.



Be careful with "unconditional branches" because they sometimes sprout ?UL (or "undefined line") errors. Usually, the Bug just wants to let you know that you told TRS-80 to send a line "somewhere," but that "somewhere" can't be found in your program.

The Big IF...

What if you don't want to play quick-draw with the (SHIFT) @ keys? Or what if you don't care to see every fraction from 1/2 to 1 millionth and beyond?

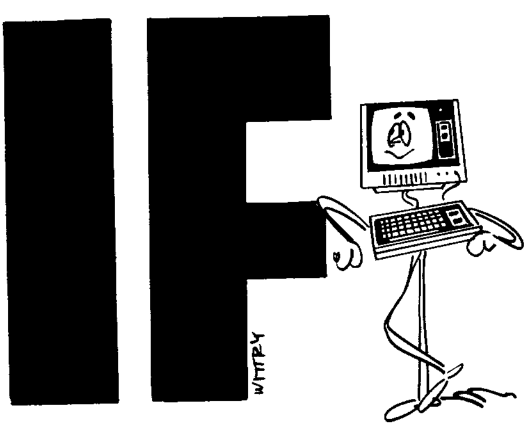
There's another "smarter" statement you can use in this case. Add the following line to the program. (Be sure the computer is READY. Press (BREAK) if it's not.)

```
45 IF A=10 THEN END
```

List the entire program. It should look like this:

```
10 CLS
20 A=1
30 PRINT 1/A
40 A=A+1
45 IF A=10 THEN END
50 GOTO 30
```

Fix any errors by retyping the incorrect line.
Now RUN it.



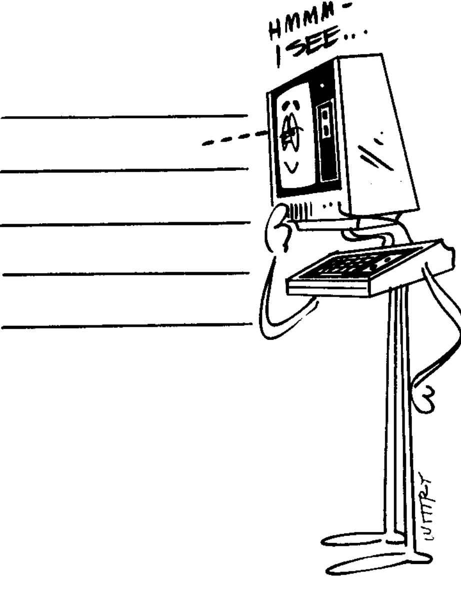
Notice that we've inserted the new line between lines 40 and 50. Now do you see why it's a good idea to "spread" the program line numbers? If the program lines were numbered 1, 2, 3, 4, 5, you couldn't insert a line!

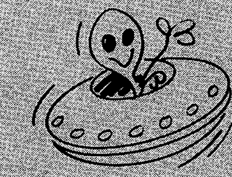
How's that for convenience!

Reading Between the Lines

Read line 45, "If A equals 10, then end the program right now — don't go any further." When A equals 10, that means we've already printed 1/9, so it's time to end the program.

What if A does not equal 10? The IF. . . THEN. . . statement doesn't say it, but the Computer knows what to do in this case: it *proceeds directly to the next line* in the program and ignores what comes after THEN. . .





UFO #5-5.
IF *test* THEN *action* is a "conditional branch" statement. If the *test* turns out okay, then the Computer will continue on to the action. Otherwise, the *test* fails and the Computer will go directly to the next line in the program, ignoring the *action*.

In our line 45 above, the test is "Does A = 10?" Here "A = 10" is not a name-it statement. The variable A is *not* given the value 10. Instead, the Computer simply tests A to see whether it already contains (equals) the number 10.

The action in line 45 is END. We'll have more to say about END later on. . . but for now, let's just say it's a great way to end a program "before its time"!

Program (Do It Yourself #5-1). Modify the program so it works out these fractions: 1/10, 1/11, 1/12, . . ., 1/19. **Hint:** Change the starting value of A (line 20) and the cutoff value of A (line 45).

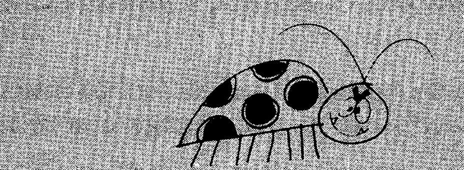
Go Ahead and Talk My Ears Off

After that last UFO, you need someone to talk to. Someone who'll understand (or pretend to).

Look no further — your Computer's ready and willing. Just erase whatever program is in memory (with NEW) and type in this one:

```
10 CLS
20 PRINT "HI, I'M YOUR TRS-80 THERAPIST!!"
30 PRINT "WHAT'S YOUR NAME?"
40 INPUT N$
50 PRINT "WHAT'S ON YOUR MIND, ";N$;"?"
60 INPUT N$
70 PRINT "VERY INTERESTING. . . TELL ME MORE."
80 GOTO 60
```

LIST the program, check it carefully, and RUN it. Now go ahead and spill out whatever's on your mind. . . only keep each message short (50 letters or less) and don't use any punctuation.



If you type in too long a message, you'll get an OS error, which means Out of String Space: the message you typed in was too long. For the time being, keep the messages under 50 characters. Later, we'll show you how to set aside more string space for longer (and more revealing) messages for the therapist.

When you've said your peace, press (BREAK) to get your Computer out of this endless loop. Otherwise it'll never stop asking for more and more juicy details.

Program (Do It Yourself #5-2). Add a line to the What's on Your Mind Program so that when you type, "THAT'S ALL", the session comes to an abrupt END.

Hint: Take a look at line 45 from the fractions program:

```
45 IF A=10 THEN END
```

That's the kind of test you need to make, only the variable is N$ and the value you're looking for is "THAT'S ALL".

Now where do you put the extra line? Well, read through the program, and decide where you want "the doctor" to check your message.

Compare your program with ours in Appendix A.

Prompted INPUT

As you've discovered by now, one of the most common operations is inputting from the keyboard to a program. Typically, the program asks a question (referred to as a prompt), and you type in a response. For example, type in this program:

```
10 PRINT "WHAT IS YOUR NAME";
20 INPUT N$
30 PRINT "HELLO"; N$; "!"
```

Notice we don't put a question mark in the PRINT message of line 10; that will be supplied by the INPUT statement. Also notice the trailing semi-colon at the end of line 10; that prevents the cursor from dropping down to the next line before printing the "?".

First RUN this program so you'll be able to compare it to a new approach we're about to use.

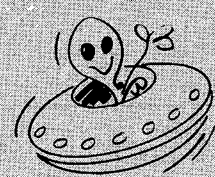
We're going to combine lines 10 and 20 into one statement, called "prompted INPUT". DELETE line 10, and change line 20 to:

```
20 INPUT "WHAT IS YOUR NAME"; N$
```

Go ahead and RUN the program, to see how the new line 20 does the work of the old lines 20 and 30.

Prompted input can be used to input several variables:

```
20 INPUT "NAME AND AGE (NAME, AGE)"; N$, AG
30 PRINT N$; "IS"; AG; "YEARS OLD."
```



UFO #5-6.
The INPUT statement can print a prompting message before stopping to wait for your input. The general form for prompted input is:

INPUT *"prompt message"; variable list*

where *variable list* contains one or more variable names, all but the last followed by a comma. *Prompt message* must be enclosed inside quotes, and the semi-colon after *prompt-message* is required.

More Ways to Test a Number

In all our IF test THEN action statements so far, we've tested for equality. There are times, however, when we're not interested in whether two numbers are equal, but rather in whether one is larger than the other. Remember the fractions program?

```
10 CLS
20 A=1
30 PRINT 1/A
40 A = A + 1
50 IF A = 10 THEN END
60 GOTO 30
```

Suppose we want to end the program when 1/A becomes smaller than 0.0125. In other words, we don't want to print out any fractions smaller than .0125. The test we need here is, "Is 1/A less than .0125? If it is, then end the program."

"<" and ">"

The two symbols, < and >, can be used to make neat designs on the Display. But, as veterans of the New Math and the Old Algebra already know, they can also be used to describe a very basic (no pun intended) relationship between two unequal numbers: which of the two has a greater value.

The < and > signs are used exclusively in tests:

```
IF A < 10 THEN 20
IF A > 10 THEN END
```

"<" means "is less than"; ">" means "is greater than."

As a memory aid . . . In both cases, the smaller side of the symbol is closer to the smaller of the two quantities; and the larger side, closer to the larger quantity.

<, > and = can be combined to produce a full vocabulary to describe any relationship between two numbers.

UFO #5-7.

BASIC has six special symbols for describing the relation between any two numbers: <, =, >, and various combinations of these three:

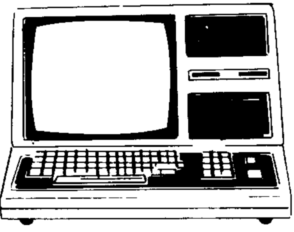
Symbol	Meaning
<	Is less than
=	Is equal to
>	Is greater than
<= or =<	Is less than or equal to
>= or =>	Is greater than or equal to
<> or ><	Is not equal to

These are known as "relational operators". They can be used inside any test, for example, IF *test* THEN *action*.

Program (Do It Yourself #5-3). Revise the simple fractions program so that it exits the loop when 1/A is less than or equal to .0125.

☑ Chapter Checkpoint #5

1. IF you want to use a "conditional branch" in your program, THEN you must include an ________ statement.
2. Is TRS-80 (<), (>), or (=) other microcomputers? (Circle one but please don't hurt TRS-80's feelings!)
3. Although you'll probably get tired before TRS-80 will, you can press [ ] [ ] to give it a rest (i.e., a pause in the program).
4. If you want TRS-80 to skip a specific line, you must tell it where to go by using ________.
5. The two types of PRINT punctuation are ________ and ________.
6. "Prompted INPUT" allows you to take a PRINT message and an INPUT line and:
   a. have tea
   b. combine them
   c. unconditionally branch them
7. To end a program before TRS-80 reaches the last line, use this BASIC statement:
   a. END
   b. THAT'S ALL FOLKS
   c. BREAK



Chapter

Six

Tricks and Treats

You've earned the right to some of BASIC's luxuries—the little things that make the Computer easier and more fun to use. The longer your programs get, the more necessary some of these "extras" will become.

"?"

When you want to PRINT something (which is quite often, right?) you don't have to type the word PRINT—you can use the ? as an abbreviation. Try these immediate lines:

```
?"THIS IS AN ABBREVIATED PRINT STATEMENT"
? 1234/100
```

You can even use the abbreviation inside a program: when the program is listed, you'll discover that BASIC replaced ? with the full word PRINT! For example, type in these lines:

Don't forget to press (ENTER) at the end of each line!

```
10 ? "HOW DO YOU INTERRUPT AN ENDLESS PROGRAM?"
20 GOTO 10
```

RUN the program. Press (BREAK) to interrupt it. Now LIST it. See what happened to the abbreviation? Also notice that the ? inside quotes wasn't mistakenly changed to PRINT.

UFO #6-1.
When typing in an immediate line or a program line, you can use ? as an abbreviation for the keyword PRINT. As long as ? appears where PRINT would normally appear, the Computer will interpret it correctly.

Multi-Statement Lines

Take a look at this program (you don't need to type it in):

```
100 CLS
200 INPUT "GIVE ME A NUMBER, ANY NUMBER"; N
300 PRINT
400 PRINT "HMMM. . . VERY INTERESTING. . ."
```

We chose increments of 100 just for variety — could have used ones, tens, etc. Line 300 is there just to separate the Computer's response from the number you type in.

In a case like this, the line numbers hardly matter at all—you can describe the sequence without referring to the line numbers: "Do the statements in order, from first to last."

When line numbers aren't important, why use them? Of course, a BASIC program requires that every line have a number — but not every individual statement. You can put several statements into one program line. All you have to do is separate the statements with colons. Here's the same program, using two multi-statement lines:

```
100 CLS: INPUT "GIVE ME A NUMBER, ANY NUMBER"; N
200 PRINT: PRINT "HMMM. . . VERY INTERESTING. . ."
```

We can even pack this entire program into a single line:

```
100 CLS: INPUT "GIVE ME A NUMBER, ANY NUMBER"; N: PRINT:
  PRINT: "HMMM. . VERY INTERESTING. . ."
```

Remember, we've had to break the line in this manual, but *you* simply type in the letters and spaces without regard to "wrap-around." Your display will look different from this example.

When you type in a line this long, you'll notice that it takes up two lines on the display. The cursor automatically "wraps around" to the beginning of the next line.

Why Use Multi-Statement Lines?

The primary reason is to save memory, so you can get more actual program into your Computer. Think of line numbers as "overhead"; they do not contribute to the actual computing power of a given program.

Another reason is to group together two or more simple, related operations. For example, assigning initial values to variables, clearing the screen and printing a message.

How Many Statements Can You Put in a Single Program Line?

The number of statements doesn't matter. The limit is set by the number of characters (letters, numbers and spaces) in your line. BASIC program lines can contain up to 255 characters. You can type in up to 240 characters directly, and sneak in an extra 15 with a technique we'll describe in the next chapter.

How Many Statements SHOULD You Put in a Single Program Line?

You may notice that our last program line, containing four distinct statements, is a little hard to follow. This is one reason for not packing too many statements into a single line.

Another limitation for multi-statement lines is that you cannot GOTO the middle of a line—only to the beginning. Take a look at this program:

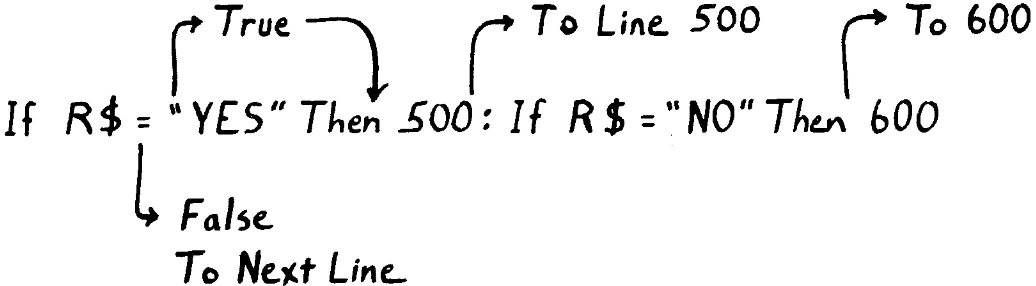
```
100 INPUT "YES OR NO"; R$
200 IF R$ = "YES" THEN 500
300 IF R$ = "NO" THEN 600
400 GOTO 100
500 PRINT "THAT'S BEING POSITIVE!"
550 GOTO 100
600 PRINT "WHY SO NEGATIVE?"
650 GOTO 100
```

In this program, the line numbers are important in describing the sequence of operations: if you type "YES", control goes to line 500, then 550, then 100; if you type "NO", control goes to line 600, 650, then 100; etc.

What would happen if you combined lines 200 and 300 into a single line?

```
200 IF R$="YES" THEN 500: IF R$="NO" THEN 600
```

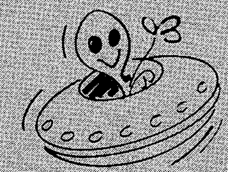
Look at the flow diagram of our new line:



All roads lead **away** from the second statement, IF R$ = "NO" THEN 600. There is no way for that statement to be executed!

Program (Do It Yourself #6-1). Use multi-statement lines wherever possible in the following program. Your final program should use five line numbers. Be careful not to create any "unreachable" statements (statements that can't ever be executed because the program will always branch around them)!

```
100 INPUT "YES OR NO"; R$
200 IF R$ = "YES" THEN 500
300 IF R$ = "NO" THEN 600
400 GOTO 100
500 PRINT "THAT'S BEING POSITIVE!"
550 GOTO 100
600 PRINT "WHY SO NEGATIVE?"
650 GOTO 100
```



UFO #6-2.
You can put several BASIC statements in a single program line. Put a colon after each statement except the last one:
statement: statement: statement

Automatic Line Numbering

Here's a way to reduce the amount of typing you have to do. First erase the previous program, then type:

```
AUTO (ENTER)
```

The Computer will respond by displaying a line number, followed by the cursor:

```
10
```

Now type in CLS and press (ENTER). The Computer will respond by displaying another line number:

```
20
```

Type:

```
PRINT "THIS IS GOING TO BE A SHORT PROGRAM"
```

and press (ENTER) at the end of the line. The Computer will display the next line number:

```
30
```

We want to stop entering lines, so press (BREAK). The Computer will display a:

```
READY
>
```

Now LIST the program.

The Computer provided the line numbers and all you had to do was type in the actual BASIC statements.

How come the Computer started at line 10 and counted by tens? These are called default values (what the Computer provides if you don't ask for anything else). You can specify any starting line number and any increment between lines.

UFO #6-3.
The AUTO command turns on an automatic line numbering function. Its general form is:

AUTO *startline, increment*

where *startline* is any line number from 0 to 65529, and *increment* is the amount to add to each line number to get the next line number. If *startline* is omitted, 10 is used. If *increment* is omitted, 10 is used.

For example:

```
AUTO 100, 50
```

will give you line numbers beginning with 100 and increasing by 50 each time you press (ENTER)

```
100, 150, 200, 250...
```

During AUTO line numbering, the Computer provides the line number; all you do is type in the program statements. Each time you press (ENTER), it stores the line and provides the line number for the next line. Press (BREAK) to get out of auto-numbering at any time. The current line will not be stored.

If, during AUTO line input, the Computer brings up a line number that is already in use, it will display an asterisk immediately after the line number. For example:

```
30*
```

indicates there is a line 30 already. If you do not wish to change this line, press (BREAK).

So You Didn't Get Your Ph.D in Speed-Reading?

What happens when your program is too long to list on the display all at once? The lines scroll off the top of the screen before you have a chance to read them.

If you've had this happen before, maybe you figured out that (SHIFT) @ will pause the listing, and (BREAK) will stop it.

Well, there's an even better way: you tell the Computer exactly which lines to list.

Command	Tells TRS-80 to list
`LIST 100`	Line 100 only
`LIST 100-110`	Lines 100 through 110
`LIST -190`	Lines up through 190
`LIST 200-`	Lines 200 and higher
`LIST.`	The last line entered or executed.

UFO #6-4.
The general form for LIST is:
  LIST *startline-endline*
where *startline* specifies the lowest-numbered line to list, and *endline* specifies the highest-numbered line to list. If *startline* is omitted, the lowest-numbered line in the program is used; if *endline* is omitted, the highest-numbered line in the program is used.

A period "." can be substituted for either *startline* or *endline*. The period signifies the "current line", i.e., the last line entered or executed.

To list a single line, type:
  LIST *line*
where *line* is the number of the desired line.

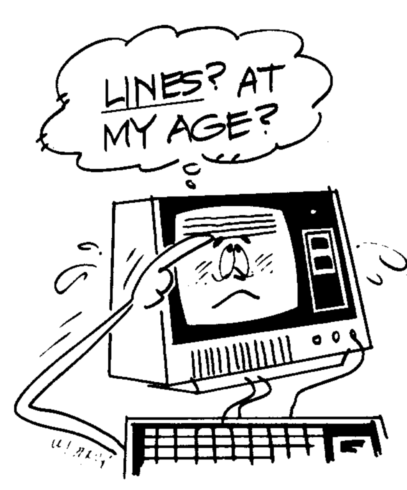
How to Get Rid of Those Unwanted Lines

Suppose you've just typed in a 40-line program, and you realize that there are about 12 unnecessary lines in the middle of the program. How do you delete the unwanted lines? Well, you could replace each one with an empty line. For example, if the lines are numbered 100, 110, 120, . . ., 210, you could type:

```
100 (ENTER)
110 (ENTER)
```

etcetera, all the way to:

```
210 (ENTER)
```



There's a simpler way. Type:

```
DELETE 100-210
```

TRS-80 will automatically delete all lines in the specified range.

UFO #6-5.
To delete a single line or a range of lines, use the DELETE command. Its general form is:

DELETE *startline-endline*

where *startline* is the lowest-numbered line to delete, and *endline* is the highest numbered line to delete. *endline* must actually be used in the program; *startline* need not be. If *startline* is omitted, the lowest numbered line in the program is omitted. (The *startline-endline* is referred to as "line range".)

To delete a single line from a program, type:

DELETE *line*

where *line* is the number of the line to be deleted. This line number must be used in the program, or you'll get a UL (undefined line) ERROR.

Let's See...Now Why Did I Write that Program Line?

When you enter a lengthy program, using a lot of variables and some complicated logic, you may be creating a problem for yourself.

Suppose you want to modify the program later...you're going to have to go through and discover what every variable and every step was for.

You could write this information down on paper, but that's not necessary. You can actually store it right inside the program! All it takes is a simple BASIC statement, REM (short for remark).

Try typing in this line:

```
REM PRINT "THIS IS A REMARK"
```

Nothing happens! That's because REM tells BASIC to ignore the rest of the line. Try putting it in a program line:

```
10 REM... THIS IS A SAMPLE REMARK STATEMENT
```

REM is often helpful following a BASIC statement, as in:

```
20 Y = Y + B: REM INCREMENT Y BY B
```

There's a handy abbreviation for REM—it's the single apostrophe, (SHIFT) (7) on your keyboard.

You use it like this:

```
30 X = X + 1 'ADD ONE TO THE COUNTER
```

Notice there's no colon separating the two statements. The colon is "built-in" to the ' abbreviation.

Note: To use REM or ' in a multi-statement line, you must make it the last statement in the line—because TRS-80 will ignore whatever follows REM.

UFO #6-6.

The REM ("remark") statement tells TRS-80 to ignore the rest of the line. Use it to add explanations to your program listing. The general form for REM is:

REM *comment*

where *comment* can be any information you choose.

The single-quote ' serves as an abbreviation for :REM. This is especially useful at the end of multi-statement lines.

☑ Chapter Checkpoint #6

1. What is another way you can command TRS-80 to PRINT?
   a. TYPE
   b. type "GET BUSY"
   c. ?

2. Which punctuation mark helps you create multi-statement lines? __________.

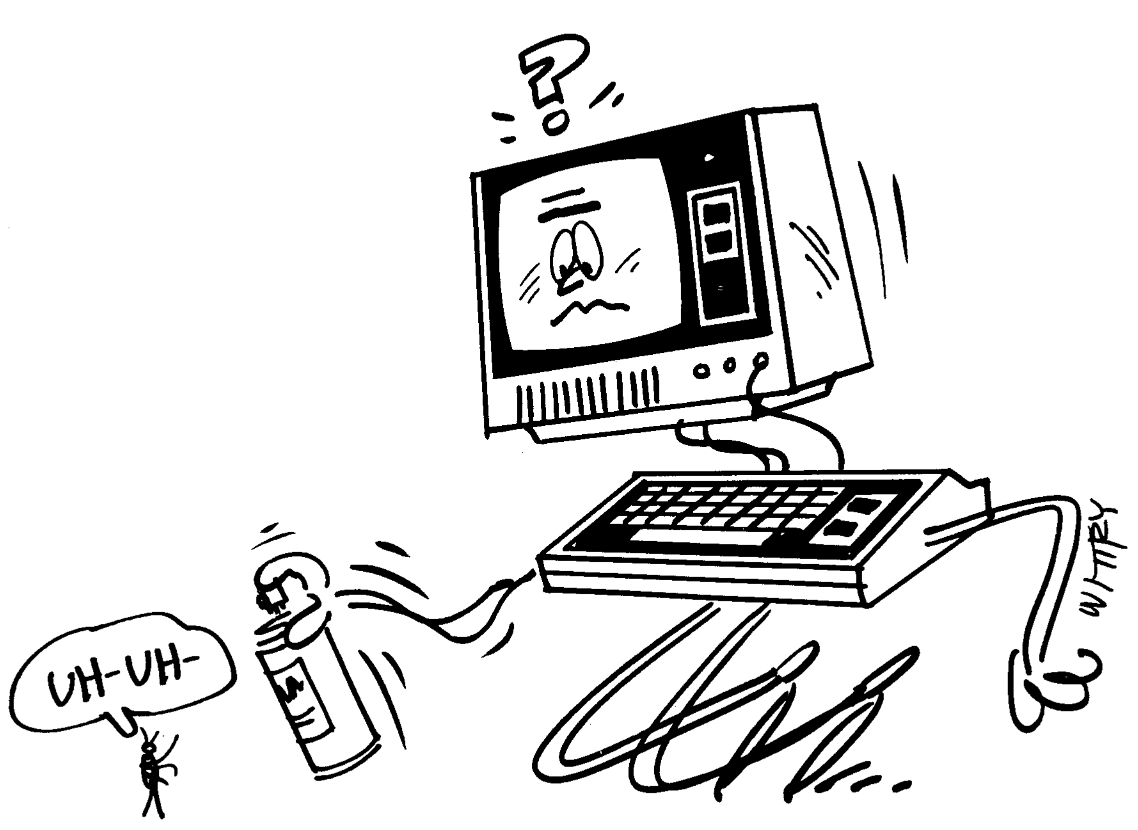
3. Since TRS-80 never sleeps, REM obviously doesn't refer to its Rapid Eye Movement. Instead it is short for __________.

4. You can easily save time by using the AUTOMATIC LINE NUMBERING.
   a. What command must you type to use this function? __________
   b. What key must you press to get out of it? (__________).

5. Unwanted lines are the bane of the programmer's existence. To get rid of them, type __________ followed by the offending line(s) number.

6. By typing __________ and a specific line number or line-range, you can easily list a particular line or line-range.

Chapter



Seven

Don't Kill That Bug—Edit It!

Your TRS-80 has another very special feature. Like AUTO and DELETE, this feature is provided to save you time when you're creating and modifying a program. It's called the Edit Mode.

The Edit Mode lets you change the contents of a program line quickly and efficiently so you don't have to re-type the whole line just because of a simple typing error.

Before starting an edit session, be prepared for some surprises.

Many keys are given special functions — as soon as you press one of these keys the Computer does what that key tells it to. *You don't have to press* (ENTER) *to be "heard"* in the Edit Mode. So, to get the most benefit (and the least confusion) from this chapter, we suggest you follow exactly the steps given.

The Edit Mode is like most other tools — to see it work, you've got to give it something to work on. So we're going to make up a series of program lines badly in need of editing. Go ahead and type in this line exactly:

```
100 THE GREEAT INTENTION SINCE THEEE PRE-SLICED BANDANNA
```

There are two ways to start editing this line: you can try to RUN it, in which case BASIC will immediately throw you into the Edit Mode for line 100, or you can use a special command, EDIT, to tell BASIC to edit the line.

Type:

```
EDIT 100
```

You may have entered the Edit Mode already. For example, after hitting a syntax error, say, in line 10, BASIC would display this:

```
?SN ERROR IN 10
10
```

In Chapter 3 we told you to press (ENTER) and retype the whole line carefully if something like this should happen. What we didn't tell you was that you had just entered. . . the Edit Mode. (At that point, it might have seemed more like the twilight zone!)

meaning "I want to edit line 100." BASIC will display the line number followed by the cursor:

```
100▒
```

You are now in the Edit Mode. YOU are the editor.

In Model III Computers, the cursor is a blinking block. In Model I's, it is an underline. We're going to show the cursor in this chapter, because its position is crucial during edit operations.

UFO #7-1.
BASIC has an Edit Mode to allow correction of program lines. When BASIC encounters a syntax error, it will automatically go into the Edit Mode so you can fix the error. You can also enter the Edit Mode by typing:

EDIT *line number*

where *line number* specifies the line you want to edit. You can substitute a period for a number; then BASIC will start editing the last line entered or executed. While you are editing a line, BASIC displays a "working copy" of the line, showing all the changes you've made.

Cursing (Cursor-ing?) Through the Line

The position of the cursor is very important in the Edit Mode, because any editing changes you make will take effect at the cursor position. Right now the cursor is sitting at the beginning of the line — on the first character.

You can advance the cursor through the line simply by pressing (SPACEBAR). Each time you press (SPACEBAR), the cursor displays another character in line 100. You aren't changing the line, just looking at it.

Go ahead and try it. Press (SPACEBAR) over and over until the entire line is displayed:

```
100 THE GREEAT INTENTION SINCE THEEE PRE-SLICED BANDANNA▒
```

Pressing (SPACEBAR) now has no effect. The cursor can't go any further — it's reached the end of the line. (Remember, you aren't changing the line yet, just displaying it.)

Now let's backspace the cursor. Use the (←) key, just as you do in the **immediate mode.** But you still aren't changing the program line itself.

You can use ⊖ to back the cursor all the way to the beginning of the line:

```
100▒
```

In short, you can use (SPACEBAR) and ⊖ to position the cursor wherever you want it.

Listing the Line

To list the entire line in its current form, simply press (L). The line will be displayed, and a new "working copy" will be started.

UFO #7-2.
In the Edit Mode, use (SPACEBAR) and ⊖ to position the cursor. Neither key has any effect on the line in the Edit Mode. Use them to position the cursor to the point at which you need to edit the line.

Press (L) to display the entire current line and start a new working copy.

Deleting a Character

Let's get rid of those extra letters in the line. Start by positioning the cursor to the first E in "GREEAT".

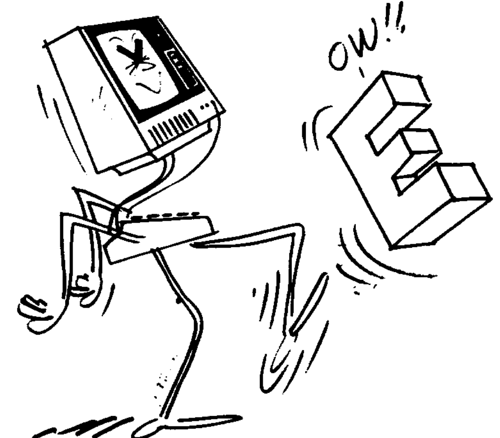
```
100 THE GR▒
```

Now you need to tell TRS-80 to delete the character immediately following the cursor. The (D) key serves this function. Go ahead, press (D). The deleted character will be displayed inside ! !:

```
100 THE GR!E!▒
```

The exclamation marks have not been added to the line — they are simply displayed so you'll know exactly which character was deleted.

DON'T MOVE THE CURSOR NOW!



It wouldn't mess up the actual program line to use (SPACEBAR) or (←), but it would make the displayed line a little confusing, because of the exclamation marks that aren't really there.

Just to be sure the character has been deleted, list the entire line by pressing (L) twice.

Now let's delete the two extra E's in "THEEE". After positioning the cursor, you could press (D) twice, but there's an easier way. You tell BASIC to delete the next two characters, by typing (2)(D). Now the ! marks will surround the "EE".

```
100 THE GREAT INTENTION SINCE THE!EE!▒
```

Press (L) twice to see the line in its current form.

Now use the delete function to change "BANDANNA" to "BANANA". After making the deletions, press (L) twice. Your line should look like this:

```
100 THE GREAT INTENTION SINCE THE PRE-SLICED BANANA
```

Changing a Character

Now we're going to change incorrect characters into correct ones. For example, we want to change "INTENTION" to "INVENTION".

First position the cursor to the "T" in "INTENTION":

```
100 THE GREAT IN▒
```

Changing a character requires two keystrokes. First you press (C), then you press the key representing the new character you want. In this case, you will need to press (V). Think of the key sequence this way:

Keystroke	**Meaning**
(C)	"I want to change the next character."
(V)	"Here is the new character."

Go ahead and try.

Nothing seems to happen after you press (C). But the Computer is waiting for you to tell it what new character to use. But when you press (V), the V is displayed where T used to be:

```
100 THE GREAT INV▒
```

Press (L) to list the remainder of the line and start an updated working copy:

```
100  THE GREAT INVENTION SINCE THE PRE-SLICED BANANA
100  ▒
```

If you want to change two characters at once, you could type:
(2) (C) *two new characters*
In general, the change function works like this:
(n) (C) *n new characters*

Inserting Characters

So far we know how to delete characters and how to change characters. But what happens when there's something missing in the line, like the "EST" after "GREAT"? Deleting or changing characters won't help here — we've got to insert some.

Inserting characters is quite different from deleting or changing them. Once you have started inserting characters, the special function keys like (D) and (C) no longer have a special function. The keyboard (with a few exceptions) acts as it does when you are typing in a line for the first time. You simply type in the characters to be inserted.

When you are through inserting characters, you use a special sequence to "escape" from this insert mode and re-enter the normal Edit Mode.

Note: While you are in the insert mode, pressing (SPACEBAR) does not simply move the cursor, as in the normal Edit Mode: it actually inserts spaces into the line.

Here is a summary of how you insert characters:

Keystroke	**Meaning**
(I)	"I want to start inserting at the current cursor position."
new characters	"Put in these characters."
(SHIFT) (↑)	"I'm ready to stop inserting characters."

Now we are ready to try inserting the letters "EST" after "GREAT". First use (SPACEBAR) and (←) to position the cursor to just after the "GREAT":

```
100  THE GREAT▒
```

Now press (I) to start the insert mode. Nothing will happen yet — but whatever you type will be inserted into the line at the current cursor position.

Now type in the letters to be inserted:

```
100  THE GREATEST▒
```

Before escaping from the insert mode, let's try making a mistake. Type in the letter Z:

```
100  THE GREATESTZ▒
```

You can correct this mistake easily — simply use ⊖ to backspace and erase the "Z":

```
100  THE GREATEST▒
```

Each time you use ⊖, another character is erased from the line — so be sure not to backspace over the good part!

Your line should look like last one listed above. Now you're ready to escape from the insert mode into the Edit Mode. Press (SHIFT) and hold it down. While holding down (SHIFT), press (↑). This is the escape sequence. Release both keys and you will be back in the Edit Mode.

List the remainder of the line (press (L)):

```
100  THE GREATEST INVENTION SINCE THE PRE-SLICED BANANA
100  ▒
```

How to End an Edit Session

Now that you've made all the necessary changes, it's time to get back to the

```
READY
>
```

prompt. There are several ways to do it. The easiest way is simply to press (ENTER).

Go ahead and do it. Then type:

```
LIST 100
```

to see the corrected line:

```
100  THE GREATEST INVENTION SINCE THE PRE-SLICED BANANA
```

. . . which describes the Edit Mode quite well.

UFO #7-3.
The three BASIC editing commands are:
n (D) for deleting *n* characters
n (C) to change a character (the next *n* keys after (C) to replace the current *n* characters)
(I) to start inserting characters. Press (SHIFT) (↑) to leave insert mode and return to the normal Edit Mode.
Model 4 Users: You can also press (F2) to escape from the insert mode.

Project (Do It Yourself #7-1). Type in the following line exactly:

```
10 PRINT "HI!! I'M YOUR TRS-80 MICROCOMPUTER": GOTO 100
```

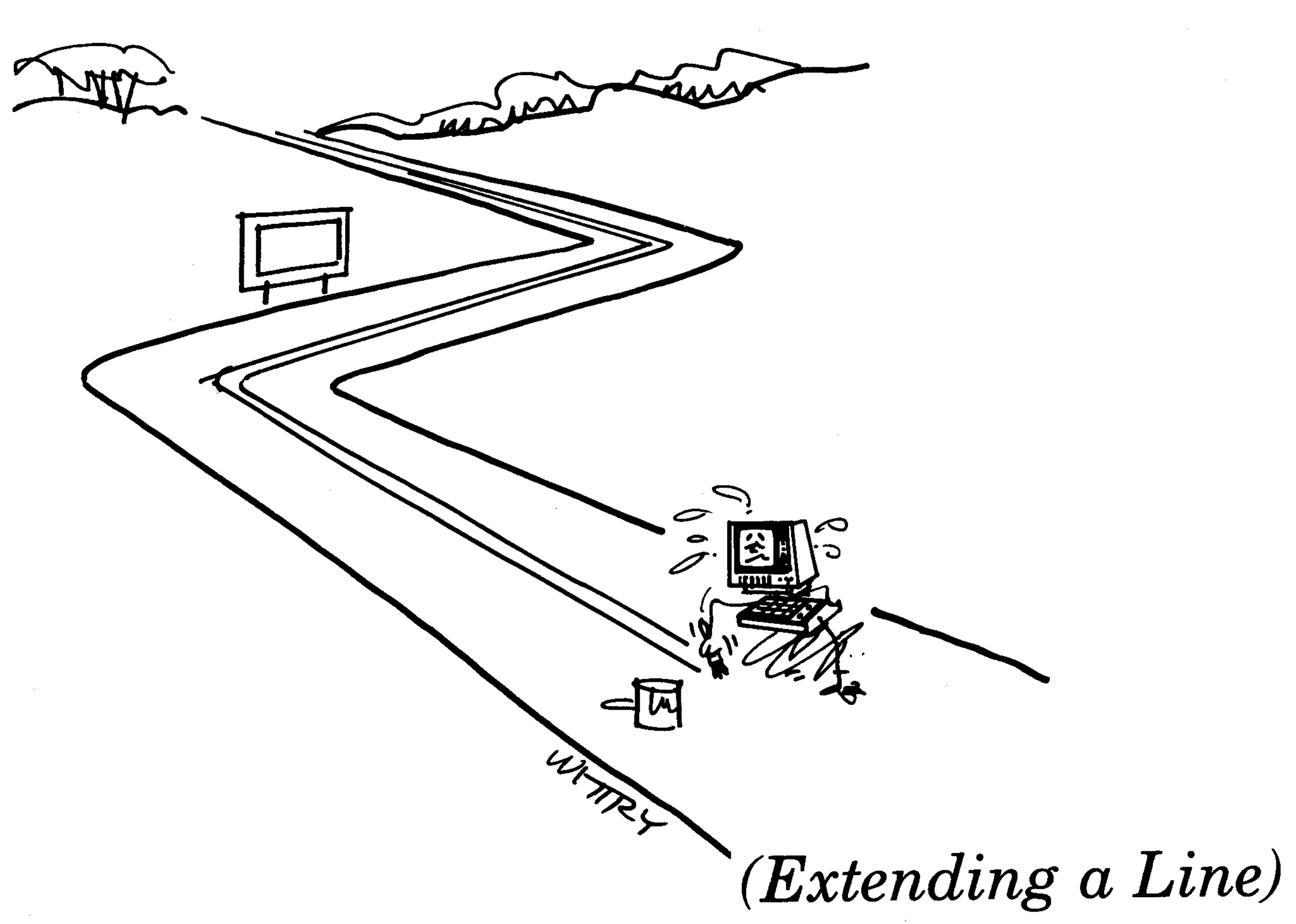
Then use the Edit Mode to change the line to this:

```
10 PRINT: PRINT "HELLO, I'M YOUR TRS-80!": GOTO 10
```

☑ Chapter Checkpoint #7

1. If you left a letter out of a word, you may insert the letter in the Edit Mode by pressing ▢ followed by the letter. To stop inserting, you press ▢ ▢.
2. The TRS-80 automatically goes into the Edit Mode whenever it encounters a ________ error.
3. You can get rid of unwanted characters in the Edit Mode by typing in the number of characters to be deleted, then pressing ▢.
4. TRS-80 will change a character in the Edit Mode when you:
   a. Type CHANGE
   b. Press (C) followed by the new character
   c. Circle the incorrect character in red ink

Chapter



(Extending a Line)

Eight

Advanced Editing

As an introduction to the Edit Mode, we showed you just the bare essentials. Now that you've got the feel for this mode, you're ready for the really good stuff!

Faster Cursor, Faster

First type in this line, just to give us something to work on.

```
100 PRINT "THE BETTER THE DATA THE DEADER THE BATTER"
```

Now go into the Edit Mode:

```
EDIT 100
100▒
```

Suppose you want to move the cursor over 20 spaces. You could press (SPACEBAR) 20 times. But there's a much faster way. Simply tell TRS-80 how many spaces to move over. Before pressing (SPACEBAR), type in the number 20. Then press (SPACEBAR):

```
100 PRINT "THE BETTER TH▒
```

The cursor skipped over 20 characters at once!

You can skip backwards, too. With the cursor positioned as in the last listing, type in the number 19 and then press ⊖.

```
100 P▒
```

Since you went forward 20 spaces and back 19, the net effect was to move forward one space.

UFO #8-1
You can advance and backspace the cursor by any number of characters, from one to 255. Simply type in the desired number of characters before you press (SPACEBAR) or ⊖. If the cursor gets to the end or beginning of the line before taking as many steps as you specified, it will stop at the limiting position.

Extending a Line

Often you simply want to add to the end of a line or "extend" it. There's a special edit function for just this purpose. It's called the extend-line function, and you start it by pressing (X) in the Edit Mode.

The extend line function is exactly like the insert (I) function, except that it lets you start inserting at the end of the line instead of at the current cursor position. Otherwise, everything we said about the insert function and the insert mode applies to the extend-line function.

Back in an earlier chapter, we mentioned that a program line can contain up to 255 characters—if you use the edit mode to cram in the last 15. As an illustration of the extend-line function, we'll demonstrate this technique.

In the immediate mode, type in a very long line. In fact, type until the TRS-80 won't take any more! Maybe your line looks something like this:

```
100 PRINT "THIS LINE WOULDN'T BE POSSIBLE WITHOUT THE EXTEND-
  LINE OR INSERT-CHARACTER FUNCTION OF THE EDIT-MODE, BECAUSE
  WHEN YOU TYPE IT IN, YOU CAN'T GET PAST THE 240TH CHARACTER,
  HOWEVER, WITH THE EDIT MODE, YOU CAN CRAM IN THE LAST 15
  CHARA▒
```

The line-endings on your display will be different from those in this example.

Now press (ENTER).

To get in an extra 15 characters, type:

```
EDIT 100
```

Now BASIC displays:

```
100▒
```

Press (X) and the Computer will display the entire line. Ours looks like this:

```
100 PRINT "THIS LINE WOULDN'T BE POSSIBLE WITHOUT THE EXTEND-
   LINE OR INSERT-CHARACTER FUNCTION OF THE EDIT-MODE. BECAUSE
   WHEN YOU TYPE IT IN, YOU CAN'T GET PAST THE 240TH CHARACTER.
   HOWEVER, WITH THE EDIT MODE, YOU CAN CRAM IN THE LAST 15
   CHARA▒
```

The cursor is at the end of the line and you are ready to start inserting. Type in an additional 15 characters. Ours will look like this:

```
100 PRINT "THIS LINE WOULDN'T BE POSSIBLE WITHOUT THE EXTEND-
   LINE OR INSERT-CHARACTER FUNCTION OF THE EDIT-MODE. BECAUSE
   WHEN YOU TYPE IT IN, YOU CAN'T GET PAST THE 240TH CHARACTER.
   HOWEVER, WITH THE EDIT MODE, YOU CAN CRAM IN THE LAST 15 CHAR-
   ACTERS. PRESTO!!▒
```

Now you're ready to leave the insert mode, so use the escape sequence:
(SHIFT) (↑)

Press (L) twice to list the line and start the cursor over at the beginning of the text. Then end the editing session.

Remember how to end an editing session? You just press (ENTER). Well, instead of using the escape sequence to get out of the insert mode, you could have used the same method. TRS-80 would return you to the READY mode, with all your corrections in effect.

UFO #8-2.
To insert text at the end of a line, use the extend function. To start this function, press (X), and then type the text to be inserted. To escape from the insert mode, press (SHIFT) (↑).

Extend-line function is just like the insert function, except that it starts inserting at the end of the line.

That's One Hack of an Editor!

In another common situation, you want to eliminate the last portion of a line; you may also want to insert some new text at that point. The hack function (started with the (H) key) serves both purposes. Like (X), the hack function puts you in the insert mode.

For example, type in the following line:

```
100 PRINT "THIS MESSAGE IS OUT OF DATE"
```

You want to change the message to: "THIS MESSAGE IS OUT OF THIS WORLD"

The hack function takes effect at the current cursor position, so start editing the line, and position the cursor like this:

```
100 PRINT "THIS MESSAGE IS OUT OF ▒
```

since that's where you want to hack and insert. Now press (H). Nothing will happen on the Display, but the Computer has deleted the rest of the line, and is waiting for some new insert text. Type in the new text, until the display looks like this:

```
100 PRINT "THIS MESSAGE IS OUT OF THIS WORLD"
```

To stop inserting and end the edit session, press (ENTER).

UFO #8-3.
To "hack away" the remainder of the line and begin inserting at the current cursor position, use the hack function. To start this function, press (H), and then type in the new text, if any. Press (SHIFT)(↑) to escape and return to the normal Edit Mode or (ENTER) to end the edit session.

Searching

There's a still more impressive way to position the cursor. You simply tell the Computer to search for a specific character in the line; it will advance the cursor until it finds the next occurrence of the character or reaches the end of the line.

Here's the key sequence to search for a character:

Keystroke	**Meaning**
(S)	"I want to search for a character."
character	"Here's the *character*."

For example, type in this new line (press ⬇ after "THROW,"):

```
100 PRINT "WHEN AJAX STRIVES SOME ROCK'S VAST WEIGHT TO THROW,
  THE LINE TOO LABORS AND THE WORDS MOVE SLOW. (POPE)''
```

Now start editing the line (EDIT 100):

```
100 ▒
```

We want to position the cursor to the "V" in "VAST". So type (S) (V) and the Computer will display:

```
100 PRINT "WHEN AJAX STRI▒
```

That's the first occurrence of "V". Type (S)(V) again to get the next one. The Computer will display:

```
100 PRINT "WHEN AJAX STRIVES SOME ROCK'S ▒
```

If you wanted to make a change now, you could (but we don't—who could improve on that line, anyway?).

Searching for the Nth Character

The search function normally finds the next occurrence of the specified character. But there's a way to skip over to the *n*th occurrence. You just type the number *n* before pressing (S). For example, press (L) to start a new working copy of line 100:

```
100 PRINT "WHEN AJAX STRIVES SOME ROCK'S VAST WEIGHT TO THROW,
  THE LINE TOO LABORS AND THE WORDS MOVE SLOW. (POPE)"
100 ▒
```

Now type: (2) and then (S) then (V) which means "find the second occurrence of "V". The Computer will display:

```
100 PRINT "WHEN AJAX STRIVES SOME ROCK'S ▒
```

UFO #8-4.
To search for a certain character in a line, and position the cursor to that character, use the (S) function. The general form is:

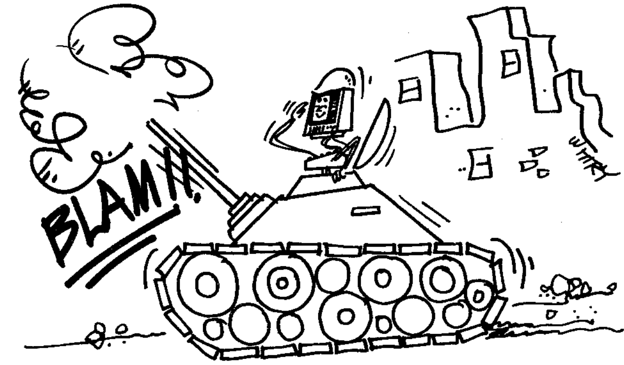
n (S) *c*

where *c* is the character you are searching for, and *n* tells TRS-80 to find the *n*th occurrence of *c*. If *n* is omitted, the Computer will find the next occurrence of *c*.

If TRS-80 reaches the end of the line before finding the *n*th occurrence of *c*, it positions the cursor at the end of the line.

Note: The search always starts at the character following the current cursor position. Therefore, if the cursor is positioned on character *c*, and you tell the Computer to find character *c*, it will skip over the current position of *c* and look in the following positions.

Search and Destroy (Kill)



When you want to save the first part of a line and get rid of the last part, you use the hack function. But what about the opposite situation: you want to delete everything up to a certain point in the line, and save the rest.

The kill function does it. You simply press (K) followed by the first character to be saved. For example, type in this line:

```
100 CLS: INPUT "PRESS ANY KEY"; X
100 ▒
```

We want to get rid of "CLS: ". Start editing the line (EDIT 100).

```
100 ▒
```

We want to kill everything up to the letter I. So type:

(K)(I)

BASIC will display the deleted text inside exclamation marks:

```
100!CLS:! ▒
```

Notice that TRS-80 also positions the cursor on the letter I. Now press (L) once to list the remainder of the line; press (L) again to list the line in its latest form:

```
100 INPUT "PRESS ANY KEY"; X
100 ▒
```

Press (ENTER) to stop editing.

You can tell Kill to delete everything up to the nth occurrence of a character, as well (remember how this works with the search function?). Type in this line, and start editing it:

```
100 PRINT "LINE 1": PRINT "LINE 2": PRINT "LINE 3"
EDIT 100
```

Let's delete everything up to the third PRINT statement. If Kill is like search, we'd use something like this:

(3)(K)(P)

Go ahead and try it. . .

Whoa!! Everything's wiped out!

```
100 !PRINT "LINE 1": PRINT LINE 2": PRINT LINE 3"!_
```

Remember how search starts looking after the current character? Kill works the same way. It was sitting on the first P; so it didn't count this one. You told it to find the third P and kill everything in front of it. It never found a third P. For this reason, it deleted everything up to the end of the line.

Press (ENTER) to stop editing, then retype line 100, and start editing it again:

```
100 PRINT "LINE 1": PRINT "LINE 2": PRINT "LINE 3"
EDIT 100
100 ▒
```

Since the cursor is sitting on the first P, it won't be counted when it searches for the nth P. So you must tell the Computer to kill everything up to the second P, by typing:

(2)(K)(P)

The Computer will display:

```
100 ! PRINT "LINE 1": PRINT LINE 2": !
```

Now press (ENTER) to end editing. List the line:

```
100 PRINT "LINE 3"
```

is all that's left.

UFO #8-5.
The Kill function deletes everything up to a specified occurrence of a specified character. If the character is not found, the entire line is deleted. The form for the kill function is:

n (K) *c*

where *c* is the first character to be saved (everything in front of it will be deleted), and n tells TRS-80 to find the nth occurrence of *c*. If *n* is omitted, the Computer looks for the first occurrence of *c*, after the current cursor position. If there is no nth occurrence of *c*, the Computer deletes the entire line after the cursor. If you press (ENTER) instead of *c*, TRS-80 also deletes the entire line after the current cursor position.

Starting Over Again

Remember the example where you killed the entire line, instead of just the first two statements? TRS-80 expects you to make some mistakes, and therefore provides a means of recovering from them. It's called the restart function; to use it press (A).

Restart cancels all changes you've made since starting the Edit Mode, and starts a new working copy of the line. Try the kill-function example again, but this time, press (A) for "Again" after you mistakenly delete the entire line. Then press (L) twice to see the restored line.

Press (ENTER) to leave the Edit Mode.

Note: Once you end an edit session by pressing (ENTER), the changes you have made go into effect. The only way to undo them is to re-type the line.

UFO #8-6.
The restore function cancels all editing changes, restores line in its original form, and positions the cursor at the beginning of a new working copy. To restore a line, press (A).

Two More Ways to End an Edit Session

Suppose you start editing a line, then suddenly realize you don't want to edit that line at all. You could press (A) and then press (ENTER). But there's a single function for this purpose. It's called the Quit function. To use it, press (Q). All editing changes you made will be canceled, and TRS-80 will put you in the READY mode.

Remember our demonstration of the restore function? To try out the Quit function, follow the restore example. But when you're ready to cancel the changes, press (Q) instead of (A).

A Special Use for the Quit Function:

You know the Computer puts you into the Edit Mode when it encounters a syntax error. Usually this is desirable. . . might as well correct the error now, right?

But, whenever you edit a program line, all variables are cleared (and so are a bunch of other information areas). Suppose a program had been churning away for 10 minutes before it reached the syntax error. You might want to examine some of the variables first, and correct the error later. In such a case, you should press (Q) as soon as TRS-80 starts editing the line. This will put you in the Immediate Mode with all variables intact so you can examine them with "?" (PRINT).

End

Pressing (ENTER) at any time ends the editing session. Pressing (E) will do the same thing, except when you're in the insert mode (started with I, H, or X functions).

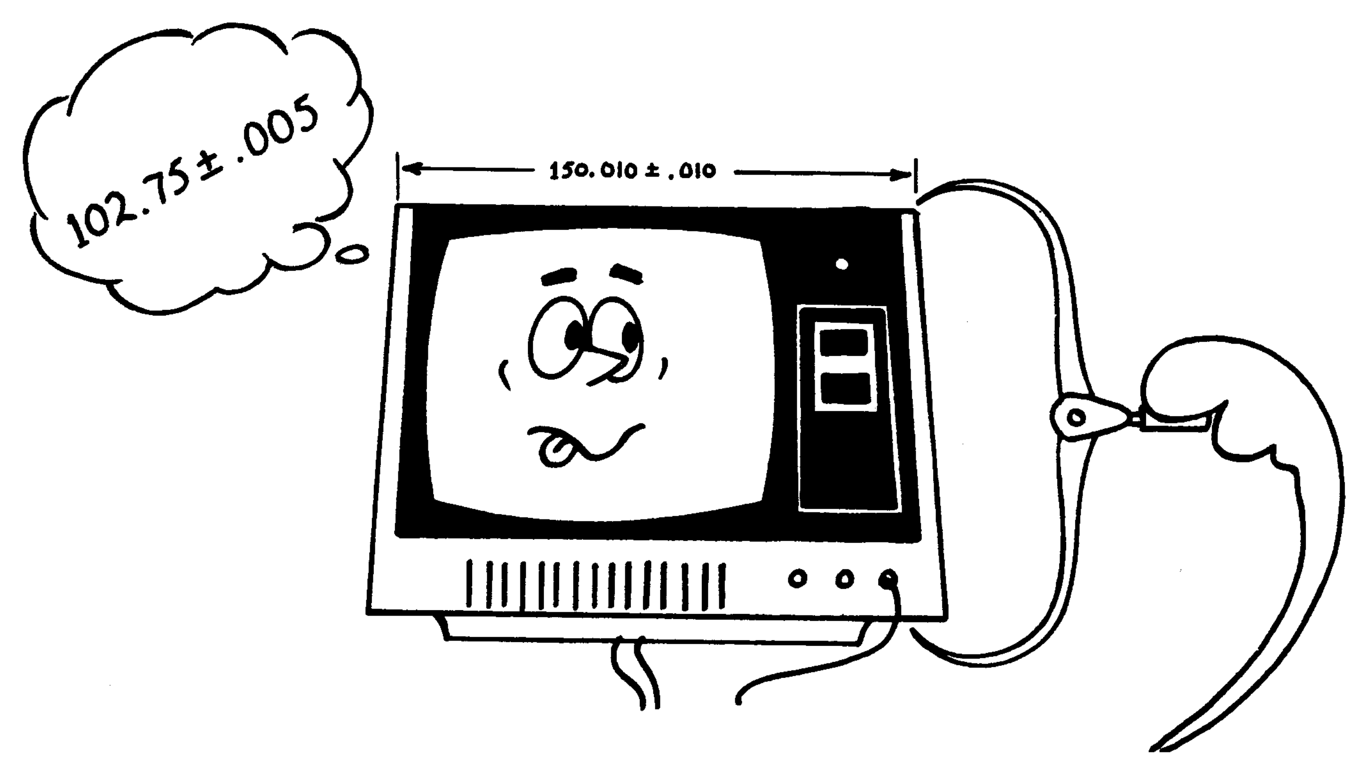
UFO #8-7.
There are three ways to end an edit session:

(ENTER)	at any time ends an editing session and effects all your changes.
(E)	ends an editing session except when you're in the insert mode (I, H, X)
(Q)	ends an editing session and cancels all changes made. Does not work in the insert mode (I, X, H)

Chapter Checkpoint #8

1. In the Edit Mode, you can skip the next 10 characters by pressing ( ) ( ) ( )
2. To Hack, or cut away the end of the line, just press ( ). You can then begin inserting characters. Press ( )( ) to return to the stop inserting characters.
3. To add characters onto the end of a line (extend it), press ( ). To stop inserting, press ( ) ( ) again.
4. The key-sequence (3) (S) (I) tells TRS-80 to:
   a. prepare Three Sandwiches Immediately
   b. delete the next three "I"s
   c. search for the third "I"
5. Can you show one of the three possible ways to end an edit session?

Chapter



Nine

A Precision Machine

Up to now we've used numbers rather off-handedly, without stopping to think how the Computer works with them. At this rate, the old math muscle might just shrivel up and die from lack of use — we'll be depending on the computer for everything!

This chapter deals with numbers. For those who don't enjoy numbers, we've tried to keep things entertaining anyway. But if you start to bog down, feel free to do some skimming.

For instance, in the program that printed out simple fractions, you didn't tell the Computer how many decimal places you wanted to see, or where to put the decimal point. To work out the fractions on paper, these are things *you* would have to consider.

For example, in working out the long-division problem:

```
      127.2
 11 / 1400.0
      11
      --
       30
       22
       --
        80
        77
        --
         30
         22
         --
          8
```

you have to decide how many decimal places to include in the answer. You can stop at 127.3 (rounding up the last digit), 127.2727, or you can fill up the line with repeating decimals: 127.272727272727272727272727272727272727272727272727. Or you may not be interested in the fractional part at all — in which case you can stop with 127.

But when you tell the Computer to

```
PRINT 1400 / 11
```

it will always produce the result:

```
127.273
```

which is accurate to six digits.

How does TRS-80 know where to stop dividing? It's all a matter of precision.

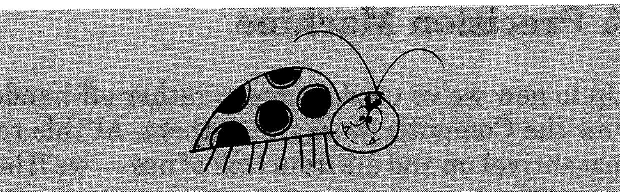
TRS-80 assumes that you're a fair-minded person — not one of those pin-headed, nose-to-the-grindstone people who wants that answer correct down to the last digit — even if it takes a lifetime to get it onto paper.

On the other hand, it doesn't mistake you for one of the What — Me Worry? types who are happy if the answer is accurate to one or two decimal places.

To give us this moderate degree of accuracy, your Computer uses what we call "single-precision" numbers.

UFO #9-1.
Single-precision means accurate to seven significant digits. Any number with seven or fewer digits can be handled as a single-precision value. All numeric variables are assumed to be single-precision unless you change their type (We'll show you how later in the chapter).

Note: When a single-precision number is PRINTed, only six digits or fewer are displayed.



Proceed with Caution! ?FC errors ("illegal function calls") can be a common annoyance throughout this chapter.

For example, all the following numbers can be handled as single-precision:

1 3.14159 1.234567 −38000000 0.00125

since none has more than seven significant digits.

You can verify this by RUNning the following program:

```
10 INPUT "TYPE IN A NUMBER"; X
20 PRINT X; "IS STORED IN YOUR COMPUTER."
30 PRINT
40 GOTO 10
```

When the program asks you to, type in the single-precision numbers listed above.

Notice what happens to 1.234567. It gets rounded to six digits.

That's because the PRINT statement does this rounding automatically — even though the full seven digits are retained internally.

Did you try −38000000? Then you got the rather strange-looking message:

```
-3.8E+07 IS STORED IN YOUR COMPUTER.
```

Similarly, typing in the number 0.00125 gives the message:

```
1.25E-03 IS STORED IN YOUR COMPUTER.
```

Here's where our concept of significant digits comes in handy. You see, your Computer is thrifty (and wise, too!) with its digits, especially when it comes to very small and very large numbers. In such cases, it only PRINTs out the significant digits, with a special notation telling you how many leading or trailing zeroes are left out. The −3.8E+07 means "move the decimal point over seven places to the **right** to get the true value." The symbol 1.25E−03 means "move the decimal point over three places to the **left** to get the true value."

RUN the program and try typing in numbers in this format (we'll call it E-format from now on). For example, input these values:

1.23E−3 1E5 −.10E8 1234567E−7

Significant digits? Those are all the digits in a number except for leading and trailing zeroes. For example,

```
10000000
```

contains one significant digit followed by seven trailing zeroes;

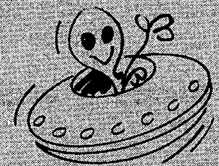
```
10000010
```

contains seven significant digits followed by one trailing zero;

```
0.0008
```

contains only one significant digit.

(This is our own definition for the term "significant digits." In other books, it may have a slightly different meaning.)



UFO #9-2.

Positive single-precision numbers greater than or equal to 1,000,000 or smaller than 0.01 are printed out in exponential format (E-format):

x.yyyyy E *zz*

Negative numbers smaller than −999999 or larger than −.01 are handled the same way.

The most significant digit *x* (i.e., the left-most non-zero digit) is printed to the left of the decimal point, and the rest of the number *yyyyy* to the right of the decimal point. E *zz* stands for "times 10 to the *zz* power". If the original number was between −1 and +1, then *zz* will be a negative number. To summarize, *x.yyyyy* E *zz* stands for the number:

$x.yyyyy * 10^{zz}$

Exercise (Do It Yourself #9-1).

Write these numbers in E-format:

1 1234.56 1000.1 0.000123 −300.0031

Write these E-format numbers in ordinary format (we call it "floating decimal" since the decimal point moves around as necessary):

1.1E1 8.7654321E8 1E6 3141593E-6 0.123456E6

If you want to mimic the PRINT statement, don't print out more than six significant digits (round the least significant digit).

Let's Get Our Data Together

As a final demonstration of single-precision and E-format, we're going to write a program that prints out a pre-determined set of numbers. Instead of using a bunch of name-it and PRINT statements, like:

```
X=1.2345
PRINT X
X=-1000.10
PRINT X
```

we're going to include all the data in a new type of statement called DATA. Then we're going to read these numbers with a new statement called READ.

RUN this program:

```
NEW
10 READ X
20 PRINT X
30 DATA 1.234567
```

The READ statement takes a value from the DATA statement and stores it in X. It doesn't matter where the DATA statement is in the program — READ will always find it. The DATA statement by itself does nothing; its function is only fulfilled when a READ statement accesses it.

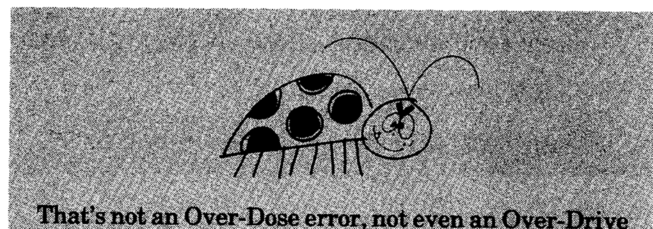
The DATA statement becomes more useful when we make it store several values, and then READ the DATA repeatedly. **Each time a READ is performed, the next value in the DATA list is used.**

Make these changes to the program:

```
27 GOTO 10
30 DATA 1.234567, 1000005, 1234.56, 8000.10
```

RUN it. You'll see the data printed out and then . . .

```
?OD ERROR
```


That's not an Over-Dose error, not even an Over-Drive error. The program simply ran Out of Data. It tried to READ after all the DATA items had already been used.

To prevent this from happening, we often put a special end of data or "EOD" marker at the end of the DATA list; before READing DATA, we check to see if the EOD has already been reached.

Make these changes in the program:

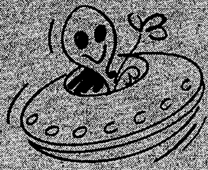
```
15 IF X=0 THEN PRINT "END OF DATA": END
30 DATA 1.234567, 1000005, 1234.56, 8000.10, 0
```

Now RUN it. Line 15 tests for the EOD. If it hasn't been read yet, the program reads another value.

We'll be saying some more about READ and DATA later, but we've got enough information to make our demonstration of single-precision and E-format. So change line 30 in your program as follows:

```
30 DATA 1.1E5, 12345E-5, 3141593E-6, -100.1001E3, 700.1E-3,
  1.234567, 1.234565, 1.234564, 0
```

RUN the program and see how the Computer interprets and displays all those values. Notice the rounding in the last three values printed.



UFO #9-3.
The DATA statement lets you store data in a program. Each item except the last must be followed by a comma. You can have as many DATA statements as you wish in a program; the Computer will treat them as if all the items were in a single list.

The data can only be read by a READ statement. The items will be read into the READ variable one at a time, starting at the first item in the first DATA statement.

The READ statement can contain a list of variables, in which case enough values are read to satisfy every item in the list. If READ runs out of data before satisfying all of its variables, an OD error will occur.

The formats for READ and DATA statements are:

READ *variable*, *variable*, *variable*...

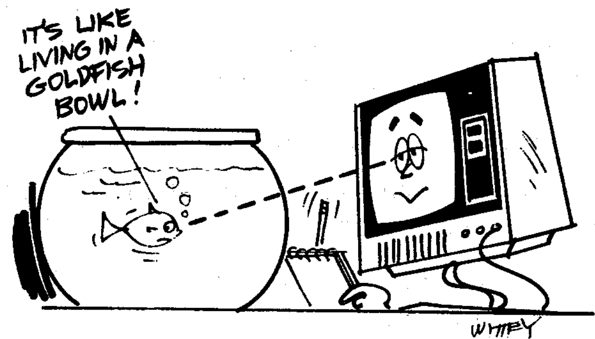
Every *variable* except the last is followed by a comma.

DATA *item*, *item*, *item*...

Every *item* except the last is followed by a comma.

The Goldfish Problem: A Case History

Despite our remarks about nose-to-the-grindstone and What-Me-Worry types, there are times when you:
a) Need more than seven digits, or
b) Need whole-number answers only.
The Goldfish Problem illustrates both situations.



Suppose you are studying the Goldfish Problem mentioned in Chapter 4. You want to compute the size of the Average American Goldfish Family (AAGF). This number will be used by the U.S. Department of GHEW (pronounced "few") for determining how much money goes to goldfish programs.

Out of 250,000 qualifying goldfish families nationwide, GHEW has surveyed 25, and come up with the following data. Each number gives the size of one of the families surveyed:

```
13, 17, 18, 11, 50, 12, 18, 10, 2, 23,
33, 81, 77, 66, 32, 11, 19, 18, 33, 1,
25, 16, 14, 13, 33
```

We are going to need this data in the program that solves the goldfish problem, so we need to store it in. . . DATA statements, of course. Go ahead and type in the following program lines (First erase any old program from your Computer):

```
100 DATA 13, 17, 18, 11, 50, 12, 18, 10, 2, 23
110 DATA 33, 81, 77, 66, 32, 11, 19, 18, 33, 1
120 DATA 25, 16, 14, 13, 33, -1
```

Notice the −1 added after the last DATA item. We're going to use this to tell when we've reached the end of the data.

Now we need to write a program that:

1. Reads and adds together all the data to get a total (we'll store the total in variable TTL).
2. Computes the average size family, using the formula:
   AVG=TTL/25

Add these lines to the program:

```
130 TTL=0
140 READ GF
150   IF GF=-1 THEN 180
160   TTL=TTL+GF
170 GOTO 140
180 AVG=TTL / 25
190 PRINT "THE AVERAGE FAMILY CONTAINS"; AVG; "GOLDFISH"
```

What's TTL? Sounds like an abbreviation for "total", but isn't three characters too long for a variable name?

Surprise — just for your convenience, you can make variable names as long as you like, so they sound more like the quantities they represent. But keep in mind two very important points:

1. Only the first two characters of the name will be significant to TRS-80. For example, TTL and TT are the same variable as far as the Computer is concerned.
2. You cannot use one of BASIC's reserved words inside a variable name. For example, TOTAL cannot be used as a variable name, since it contains the reserved word TO. See the list of Reserved Words in the Appendix.

Read through the program and try to determine what's going on. We're using our new statements, READ and DATA.

Notice the program repeats lines 140-170 until the last item (−1) is READ. As each item is read, it is added into the variable TTL, which keeps a running total of all the data.

When the last item is read, the program breaks out of the loop (line 150) and transfers to line 180. Line 180 computes the average by dividing the total by the number of items counted. Line 190 prints out the results.

Now RUN the program.

```
THE AVERAGE FAMILY CONTAINS 25.84 GOLDFISH
```

Sounds like a government study, doesn't it? Sure feel sorry for that fraction of a goldfish. GHEW just might have to set up a commission to study that problem . . .

Wait — there is a simple, painless way to eliminate it (the problem, not the fish). It's called a Software Solution.

No, that's not a delicate-fabric cleaning agent!

Add a % after every variable name in the program. (Don't ask why, just do it!) You will have to change lines 130, 140, 150, 160, 180 and 190 as follows:

The Edit Mode's Insert Command will come in handy here!

```
130 TTL%=0
140 READ GF%
150   IF GF%=-1 THEN 180
160   TTL%=TTL%+GF%
170 GOTO 140
180 AVG%=TTL% / 25
190 PRINT "THE AVERAGE FAMILY CONTAINS"; AVG%; "GOLDFISH"
```

Now RUN it again.

```
THE AVERAGE FAMILY CONTAINS 25 GOLDFISH.
```

That's better! Now what were those % signs for?

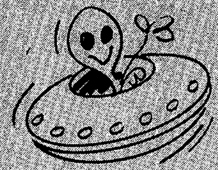
Remember The Almighty $? It changes an ordinary variable into a string variable. Well, % performs a similar function — it changes the nature of the variable.

Just to give you some more ideas about %, type in the following lines in the **immediate** mode (don't change the program that's in memory):

```
A%=1.334
B%=100.0001
C%=-1.5
D%=-300.1
PRINT "THE NUMBERS ARE STORED AS"; A%, B%, C%, D%
```

So, 1.334 gets stored as 1; 100.001 as 100; −1.5 as −2; and −300.1 as −301.

See the pattern? The positive numbers have their fractional portions chopped off, returning only whole numbers. . . but what about the negative numbers? Hmmm.



UFO #9-4.
Adding % at the end of a variable changes its type from single-precision to a different type called "integer." An integer type variable can only store whole numbers between -32768 and 32767. When you store a non-whole number in an integer variable, the largest number not greater than the original number is stored.

For positive numbers, this means that only the whole-number portion is stored; for negative numbers, the next smaller whole number is stored.

A%, B%, C%, and D% are all integer variables. They are completely separate from the single-precision variables A, B, C and D.

To prove it, type in these lines (immediate mode):

```
A%=100
A=3.141593
PRINT A%,A
```

Program (Do It Yourself #9-2). Our solution to the problem of the fractional goldfish isn't really statistically correct. In effect, we "rounded" all our fractional values *down*. We should have rounded them to the *nearest* integer value, e.g., 26.4 rounds to 26, while 26.5 rounds to 27. This latter procedure is known as "4/5 rounding."

There's a neat trick to accomplish 4/5 rounding. Before chopping off the fractional part of a positive number, we add .5 to it. Watch this:

26.4+.5=26.9 which converts to integer 26
26.5+.5=27.0 which converts to integer 27

Can you think of a way to change line 180 in the program so that we accomplish 4/5 rounding?

Compare your answer with ours in Appendix A.

GOLDREC and CLEPOBO

Now that we've computed the size of the AAGF, we can use this information to see how much money goes to two very important goldfish programs.

GHEW will place $1.01 annually in the Goldfish Recreational (GOLDREC) Fund, and $1.000000098765 in the Clean Ponds and Bowls (CLEPOBO) Fund for each goldfish in the land.

We need to add steps to the Goldfish program to accomplish the following:

1. Multiply the size of the average family by the number of qualifying families (250,000) to get the total goldfish population:
   POP=AVG*250000
2. Multiply the total population by 1.01 to get GHEW's annual contribution to GOLDREC:
   GOLDREC=POP*1.01
3. Multiply the total population by 1.000000098765 to get GHEW's annual contribution to CLEPOBO:
   CLEPOBO=POP*1.000000098765

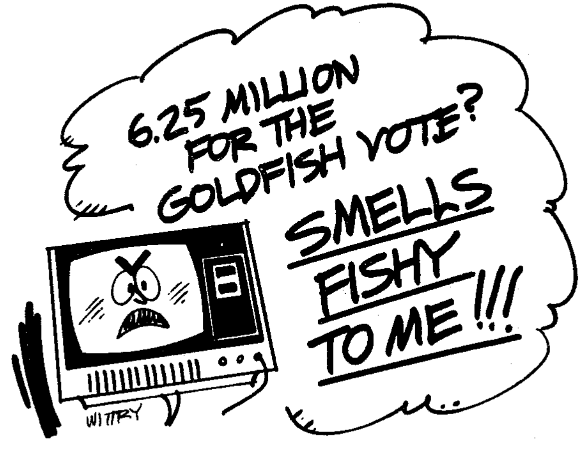
Add these lines to the program:

```
182 POP=AVG%*250000
184 GOLDREC=POP*1.01
186 CLEPOBO=POP*1.000000098765
200 PRINT "THE ESTIMATED GOLDFISH POPULATION IS"; POP
210 PRINT "THE ANNUAL ALLOCATION TO GOLDREC IS $"; GOLDREC
220 PRINT "THE ANNUAL ALLOCATION TO CLEPOBO IS $"; CLEPOBO
```

Now RUN it.

```
THE ESTIMATED GOLDFISH POPULATION IS 6.25E+06
THE ANNUAL ALLOCATION TO GOLDREC IS $6.3215E+06
THE ANNUAL ALLOCATION TO CLEPOBO IS $6.25E+06
```

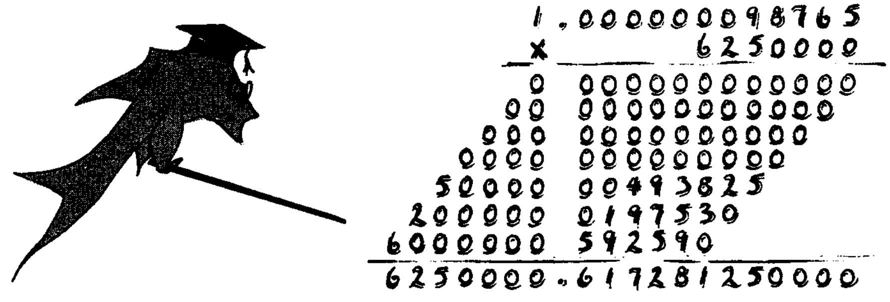
Now that you can read exponential format, you realize that $6.3215 million goes to GOLDREC, and $6.25 million to CLEPOBO. We should have some very happy goldfish . . .



There may be a few unhappy taxpayers out there, but GHEW is there to serve goldfish!

A Question of 62¢

One upstart goldfish in an Eastern goldfish school is upset. He figures that with an estimated goldfish population of 6.25 million, and a CLEPOBO budget of $6.25 million, that's exactly $1.00 for each goldfish in the land. But the CLEPOBO grant allows $1.000000098765 for each goldfish. Taking a pointed shell in his mouth, he begins scratching away on the sandy floor of his pond. The work is arduous, but as he nears the end of the problem, his eyes begin to gleam with a sharklike ferocity. Here's what he finds out:



Rounding up to the nearest penny, he confirms his suspicions — the amount budgeted for CLEPOBO is short by 62 cents! A whole year's tuition at the Goldfish School of Rodeo in West Texas!

Our upstart goldfish brought the matter to the attention of the CLEPOBO authorities, and they puzzled over his calculations all day long. . .

The Heart of the Problem

Remember that single-precision numbers can hold only seven significant digits; extra digits are simply ignored. Now look at line 186:

```
186 CLEPOBO=POP*1.000000098765
```

CLEPOBO is a single-precision variable, but we are asking it to store the product of POP*1.000000098765. The constant contains 13 significant digits, and the product is bound to contain even more than that. *But CLEPOBO can only hold the seven most significant digits* of this product.

GHEW is simply going to have to revise the fudge-factor for the CLEPOBO budget to a single-precision number. What a lot of red tape that will require. . . unless we can come up with more precision in our program.

Let's try a little experimentation in the **immediate** mode. Type:

```
PRINT 6250000*1.000000098765
```

The Computer will display the result:

```
6250000.61728125
```

which looks awfully similar to what our upstart goldfish got.

So much easier than scratching it out in the sand!

But how did the computer give us all those digits? The answer contains no fewer than 15 of them. (No wonder the 62 cents got lost!)

It's going to take a few pages to explain, but for now, think of it this way: The **immediate** PRINT didn't involve storing values in single-precision variables. So TRS-80 didn't have to follow the limitations of single-precision variables. . . so we got a lot more precision in our answer. . . for a fuller explanation, read on!

A Software Solution (Again)

Change lines 186 and 220, by adding "#" to the end of the variable name CLEPOBO:

```
186 CLEPOBO#=POP*1.000000098765
220 PRINT "THE ANNUAL ALLOCATION TO CLEPOBO IS $"; CLEPOBO#
```

Now RUN the program.

```
THE ANNUAL ALLOCATION TO CLEPOBO IS $6250000.61728125
```

Which solves the Goldfish Problem, and ends our case study. (Man and fish working together. . . for a better world.)

Now, what was that # all about?

UFO #9-5.
Adding # at the end of a variable changes it from single-precision to double-precision. It can then store up to 17 significant digits. Constants with eight or more digits are also treated as double-precision.

Note: When the Computer PRINTS a double-precision value, it displays a maximum of 16 digits. The 17th digit is used for rounding purposes.

For example, all of the following are double-precision variables:

CLEPOBO# A1# PI# Z1# L# ZZZZ#

Remember, only the first two characters of any variable name are used by the Computer — CLEPOBO# is the same double-precision variable as CL#.

To further investigate integer, single-precision and double-precision, RUN this program:

```
NEW
10 INPUT "TYPE IN A NUMBER"; X#
20 X=X#
30 IF X#<-32768 OR X#>=32768 THEN X%=0: GOTO 50
40 X%=X#
50 PRINT "INTEGER", "SINGLE-PREC", "DOUBLE-PREC."
60 PRINT X%, X, X#
```

Notice line 30. It's there to make sure you don't try to assign a number to integer variable X% which is too large in either "direction" (positive or negative) to be stored in an integer. Remember, integer variables can only represent numbers between −32768 and 32767.

When the program asks you to, try typing in these numbers:

12345.678901234567
-300.6666666
1.6666666666666666 (16 sixes)
9999999999999999 (16 nines)
10000000000000000 (16 zeroes)
.01
.009

Did you try entering 10000000000000000? Then you got a number in what looks like E-format, with one Difference:

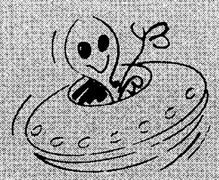
```
DOUBLE-PREC.
1D+16
```

Similarly, when you type in 0.009, you see:

```
DOUBLE-PREC.
9D-03
```

Now try turning the tables and typing in these values as INPUT to the program:

123456789D−9   300.1D10   .33D1



UFO #9-6.
Double-precision numbers greater than or equal to 10000000000000000 or smaller than 0.01 are printed out in double-precision exponential format (D-format):

x.yyyyyyyyyyyyyyy D *zz*

Negative numbers smaller than −999999 or larger than −.01 are handled the same way.

The most significant digit *x* (the left-most digit) is printed to the left of the decimal point, and the rest of the number *yyyyyyyyyyyyyyy* to the right of the decimal point. D *zz* tells stands for "double-precision times 10 to the *zz* power". If the original number was between −1 and +1, then *zz* will be a negative number. To summarize, *x.yyyyyyyyyyyyyyy* D *zz* stands for the number:

$x.yyyyyyyyyyyyyyy * 10^{zz}$

Another way of looking at it is: for positive *zz*, move the decimal point to the right *zz* places; for negative *zz*, move the decimal point to the left *zz* places.

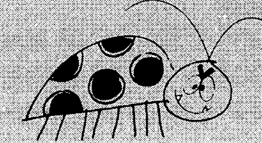
A Double-Precision Curve Ball

Now that you've got your math muscle back in shape, how about flexing it just a bit?

Program (Do It Yourself #9-3). Write a program that:
1) Stores the fraction 1/3 in a double-precision variable
2) Multiples it by 3
3) Prints out the answer in all its double-precision glory.

Check it over and then RUN it.

So you always thought 1/3*3=1, not 1.000000028902322! Well, you learn something new every minute with a computer.



If you got 1/3*3=1, either
a) You didn't use double-precision variables, or
b) You very cleverly sidestepped the problem we're trying to create.

If a) is true, go back and try it again using double-precision (#) variables. If b) is true, go back and try it again — not so clever this time! After all, how can you learn from your mistakes if you don't make any!

To see the accepted answer and the Clever Sidestep, see the answers in the Appendix.

You Can't Squeeze Double-Precision Out of a Single-Precision Number

When the Computer executes the line

```
A#=1/3
```

it first computes the result of 1/3, and then stores the result in A#. But, in dividing 1 by 3, it doesn't know you want it to store the result in a double-precision variable. It assumes you are only interested in a single-precision result. Therefore it computes 1/3 accurate to seven digits only. When this value is placed in A#, the extra precision (10 more digits) consists of. . . numeric garbage!

Just to make the situation look even worse (so our solution will look that much better), type in these immediate lines:

```
A#=1.9
A=1.9
PRINT A#, A
```

Single-precision is more accurate than double!?

#&*||&!!! That Double-Precision

Erase the program in memory, and RUN this one:

```
10 A#=1/3#
20 B#=A#*3
30 PRINT A#; "*3="; B#
```

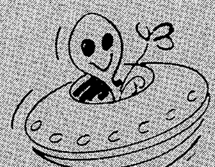
The Computer will display:

```
.333333333333*3=1
```

This works because the # after 1/3 tells your Computer you want double-precision.

Try these lines:

```
A#=1.9#
PRINT A#
```



UFO #9-7.
Adding # at the end of any number makes that number double-precision. If any number in a +, −, *, or / operation is double-precision, then the operation is done in double-precision, producing a double-precision result. Any constant with eight or more significant digits is also stored as double-precision.

For example, all of the following operations will produce double-precision results:

```
1#+3.14159    -300.1#-1    11/12#
```

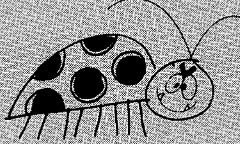
The results of these operations can be stored in double-precision variables with 17 digits of accuracy.

Program (Do It Yourself #9-4). Write out a program that continues printing out the sequence:

5, 25, 125, 625, etc.

Make the answers accurate to 17 significant digits.

Check your answer against ours in the Appendix.



Did the program stop with an OV error? Remember what that means? . . . Overflow. At a certain point in the calculations, the Computer attempted to generate a number that exceeded its limits for numbers.

For single- and double-precision numbers, the value of the number must be less than 1.7E38 and greater than −1.7E38. (You can actually sneak in some more digits after the 1.7; for example, 1.711E38 can be stored without an OV error.

If the number gets very close to zero (between −1.7E−38 and +1.7E−38), it will be stored as zero.

☑ Chapter Checklist #9

1. If you can convert these to exponential form, TRS-80 will give you an E for Effort!

   a. −38000000
   b. 1111111

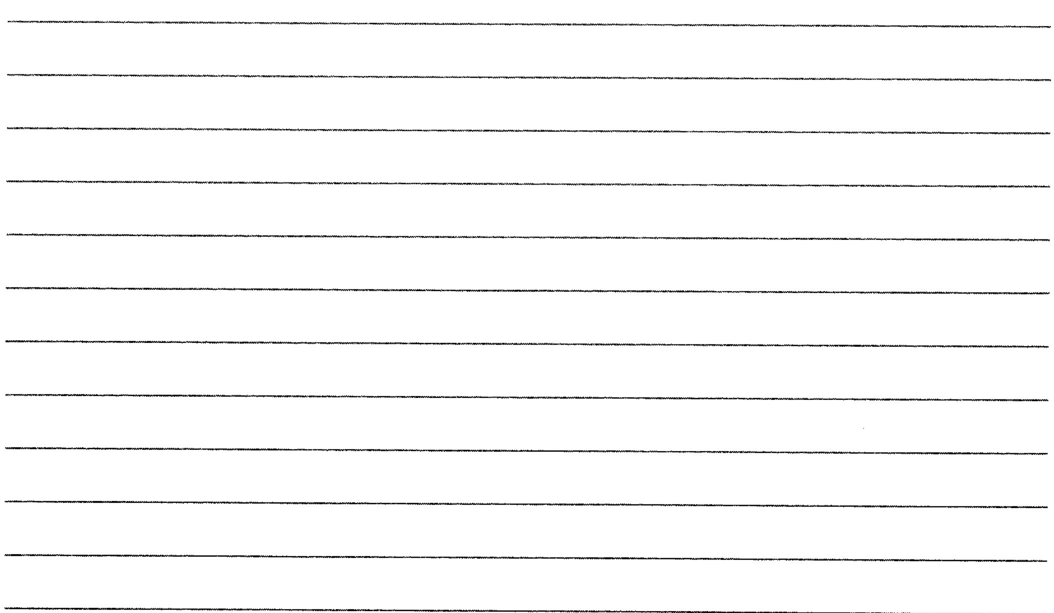
2. Which of the following are **single-precision** constants?

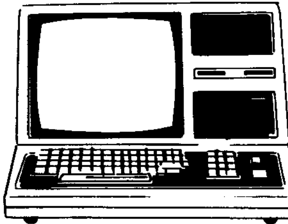
   a. 3.14159
   b. 1.234567
   c. 1.1000000125

3. Can you choose the incorrect DATA statement?

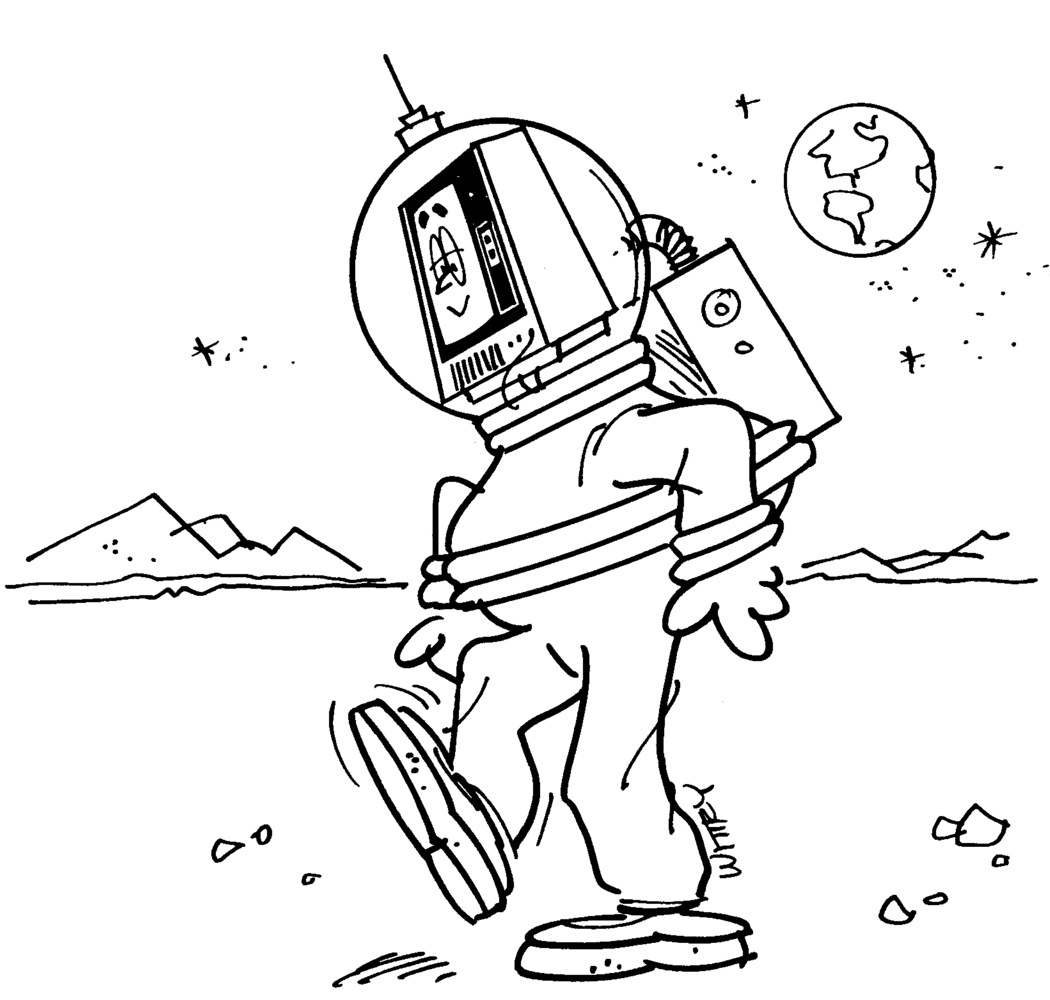
   a. DATA 1.34, 1005, .679
   b. DATA 1.34 1005 .679

4. By adding "#" to a variable, you can change from single-precision to ________-________.





Chapter



Ten

A Medium-Sized Step Forward

The last program uses repetitive multiplication to compute the terms in the series:

5^1 5^2 5^3 5^4 5^5

It works fine if you're interested in every term in the series, but is a roundabout way to get any particular term. For example, to compute 5^4, you must multiply three times to get the answer.

TRS-80 Has a Better Idea. . .

See the ↑ key? Use it when you type in these lines:

```
PRINT 5 ^ 4
PRINT 5 ^ 11
```

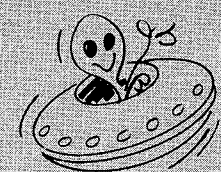
Pressing ↑ may generate a "↑," "^," or a "[." It won't matter what character is displayed. The function will be the same.

Program (Do It Yourself #10-1). Rewrite Do It Yourself #9-4 to print out same series, using the ↑ symbol.

Compare your answer with ours in the Appendix.

Notice anything different in the output? (Look at the terms greater than 999,999. Hmmm. Repetitive multiplication isn't obsolete after all!

The numbers greater than 999,999 now appear in E-format, with only the six most significant digits displayed.



UFO #10-1.
The ↑ operator indicates exponentiation—raising one number to another power. The general form is:

x ↑ y

The result of exponentiation is always single-precision, even if one or both of the operands (*x* and *y*) are double-precision.

The absolute value of a number is its magnitude without regard to whether it's negative or positive. For non-negative x, the absolute value of x equals x. For negative x, the absolute value of x equals -1*x.

The last program? You remember—Do It Yourself #9-4 where you had TRS-80 continue to print out the sequence:
5, 25, 125, 625. . .
to 17 significant digits.

Note for Model III Users. Remember that your ↑ will display a [(left bracket) but still functions like an up-arrow.

Nth Roots and Other Strangers

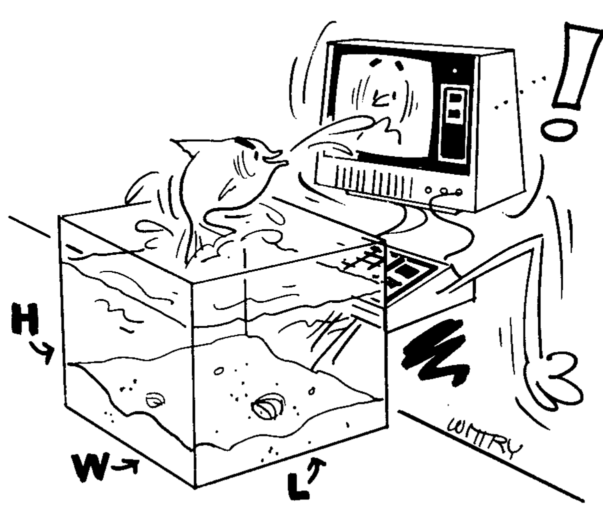
You know how to compute the result of

5^{11}, 5^8, 5^5

To generalize, you can compute 5^n. But what about the square root of 5? 10th root? *n*th root? How do you compute them?

Suppose we're designing a cube-shaped goldfish bowl (what else?) to meet GHEW requirements that for every goldfish, 371.08 cubic inches (6080.9 cubic centimeters) of water are necessary.

We have 10 fish, so we'll need 3710.8 cubic inches (60809 cubic centimeters) of water. So what should the dimensions of our bowl be? Here's a plan for the bowl:



The *n*th root of 5 is the number which, when multiplied times itself *n* times, produces the result 5. In other words, if

$\sqrt[n]{5} = x$

then

$x^n = 5$

For example,

$\sqrt[2]{16} = 4$

and $4^2 = 16$

The formula for the volume of such a container is:

$V = height * width * depth$

or to be more official,

$V = h * w * d$

Since all sides h, w and d are equal in a cube, we can rewrite our formula like this:

$V = h * h * h = h^3$

This looks like a case for nth (actually cube) roots!

$\sqrt[3]{V} = h$

Program (Do It Yourself #10-2). Solve the goldfish cube problem by trial and error. Determine the dimensions of the smallest container that will hold enough water for our goldfish:
a) To within 1 inch or 1 centimeter.
b) To within 1/4 inch or .25 centimeter.

____

____

____

____

____

____

Did you remember to allow a little extra space (for the goldfish)?

Back from the Shadows

Well, it's time to dust off the old exponentiation operator. (No—it can't return double-precision, but GHEW specifications are always single-precision.)

Think back to that dusty chalkboard on which your childhood monsters came to life as algebraic formulas and equations. Remember this one?

$$\sqrt[n]{V} = V^{1/n}$$

For example,

$$\sqrt[3]{27} \quad 27^{1/3}$$

Got any ideas? Now write a one-line program that positively leaps to the answer. Check your answer by raising it to the third power (cubing it).

The cubed result should be 371.08 inches or 6080.9 centimeters. Does your answer check out? If it does, you pulled another clever one. If it's not—you're about to learn an important fact of life:

Hint: Plug the volume (371.08 inches or 6080.9 cubic centimeters) and the number of sides (3) into the equation (substitute for V and n).

```
PRINT 371.08 ^ 1/3
```

Purple Elephants Make Double Agents Snicker, or Parentheses Even Make Difficult Arithmetic Simple (PEMDAS)

There are two operations in the expression

```
371.08 ^ 1/3
```

exponentiation and division. How does the computer decide which to do first? If it does exponentiation first, the calculation will be 371.08 ↑ 1, and this result will be divided by 3.

If it divides first, the first calculation will be 1/3, and this result will become the exponent for 371.08. Which comes first, ↑ or /?

Back to the Laboratory, Igor!

Exercise (Do It Yourself #10-3). Find out the sequence in which all the arithmetic operators (+ − * / and ↑) are done. Find out if it matters which operation comes first in the line. Write down a pecking order, or "hierarchy" as its called in mathematics.

(Hierarchy—that's worse than monarchy . . . I always knew those math-heads were reactionaries!)

Write down the hierarchy here:

Now how do you get the result you want? Here's an indirect solution:

```
A=1/3
PRINT 371.08 ^ A
```

Here's a shorter way:

```
PRINT 321.08 ^ (1/3)
```

UFO #10-2.

When TRS-80 evaluates expressions involving two or more operations, it uses the following hierarchy:

1. Operations within the same level of parenthesis are evaluated first.
2. Within a given level of parentheses, operations are performed in this order:
   ↑
   * and /
   + and −
3. If * and / are used in the same level of parenthesis, whichever comes first is done first. The same is true of + and −.

Program (Do It Yourself #10-4). Add parentheses to make this line print out a value of 7776.

```
PRINT 6 ^ 5 ^ 4 / 3 + 2 - 1
```

Modern Living with Square Roots and Natural Logs

You know how to get the *n*th root of any number. It always takes two operands (the x and $\frac{1}{y}$ in our UFO #10-1 example). But for those ever-popular square roots, TRS-80 has an easier way. It's called a built-in function. Try this program.

```
10 INPUT "ENTER A NUMBER"; N
20 PRINT "THE SQUARE ROOT IS"; SQR(N)
30 GOTO 10
```

Whistlestop Tour (Do It Yourself #10-5). Replace line 20 in the above program with each of the following lines (one at a time), and observe the results:

```
20 PRINT "THE NATURAL LOG IS"; LOG(N)
20 PRINT "THE SIGN-COMPONENT IS"; SGN(N)
20 PRINT "THE ABSOLUTE VALUE IS"; ABS(N)
20 PRINT "THE LARGEST WHOLE NUMBER IS"; INT(N)
20 PRINT "THE LARGEST INTEGER IS"; CINT(N)
20 PRINT "THE WHOLE-NUMBER PORTION IS"; FIX(N)
20 PRINT "THE SINE OF"; N; "DEGREES IS";
   SIN(N*1.745329E-2)
```

There's a complete table of built-in math functions in your TRS-80 BASIC Language Reference Manual. Most of them will be introduced and used in the following chapters.

For these lines, try the following input values:

LOG: 1, 10, 2.71828
SGN: −100, 0, 100
ABS: −300.1, 0, 300.1
INT: −300.1, 12344.5, 65000.1
CINT: −300.1, 12344.5, 65000.1(OV ERROR is normal)
FIX: −300.1, 12344.5, 65000.1
SIN: 30, 60, 90, 120, 180

Don't #!%$ that Number — Declare It!

Back in Chapter 4, we said there were five types of BASIC statements:

- Input/Output
- Name-it
- Graphics
- Program Sequence
- Declare-it

We've sampled every one of these except the last. Now that you're able to write programs using integers, single- and double-precision numbers, and strings, the declare-it statements will come in handy.

When you want to change a variable from single-precision to another type, you already know you can add one of the symbols # (double), % (integer) or $ (string). These are called "type-declaration tags."

With certain declare-it statements, you can define whole groups of variables as any of these data types. For example, the statement:

DEFINT I

tells TRS-80 that from now on, any variable starting with the letter I is automatically assumed to be integer, unless you add a #, $, or ! symbol at the end. Try this program:

```
10 DEFINT I
20 I1=1.2345
30 IH=33.3333333333
40 I1!=1.2345
50 IH#=33.333333333
60 I=-5.5
70 I$="THIS IS A STRING VALUE"
80 PRINT I1, IH, I1!, IH#, I, I$
```

Notice that now TRS-80 assumes I1, IH and I are integer variables, but that you can change this by adding the type declaration tags at the end of the variable name.

Oh—we forgot to tell you about !. It's just like #, % and $, except it tells BASIC that the variable is single-precision. You never need it unless you've defined a variable to be some other type.

Why did I = −5.5 print out as −6?

When a negative number with a fractional part is stored in an integer variable, the next lower whole number is used.

Refer back to UFO #9-4 for details on how numbers are stored in integer variables.

UFO #10-3.

There are three statements for changing the way TRS-80 determines what type a given variable is.

- DEFINT ("define-integer") makes variables starting with the specified letter integer variables.
- DEFSNG makes variables starting with the specified letter single-precision.
- DEFDBL makes variables starting with the specified letter double-precision.
- DEFSTR makes variables starting with the specified letter string variables.

The letters to be changed can be specified in a list, with a comma after every letter but the last; or in a range, with a hyphen between the first and last letter. For example:

```
DEFINT I, J, K,
DEFDBL A-D
DEFSTR S, W-Z
```

Adding a type declaration tag (#, !, %, $) always overrides the type assigned to a variable.

Declare-it statements should always be placed at the beginning of the program, before values are assigned to the variable.

The Old Shell Game

Try this program:

```
10 DEFINT A: DEFDBL B: DEFSTR C
20 A=1000: B=1.234567890123456: C="FIND ME!"
30 DEFSNG A-C
40 PRINT "A="; A
50 PRINT "B="; B
60 PRINT "C="; C
70 PRINT "NOW WHERE DID THE ORIGINAL DATA GO?"
```

Now you see why DEF statements belong at the beginning of the program, before values are assigned!

Exercise (Do It Yourself #10-6). Figure out a way to print out the original contents of A, B, and C. (Hint: Type declaration tags always override the "assumed" type of a variable.)

☑ Chapter Checkpoint #10

1. With TRS-80's help, can you perform these math operations?

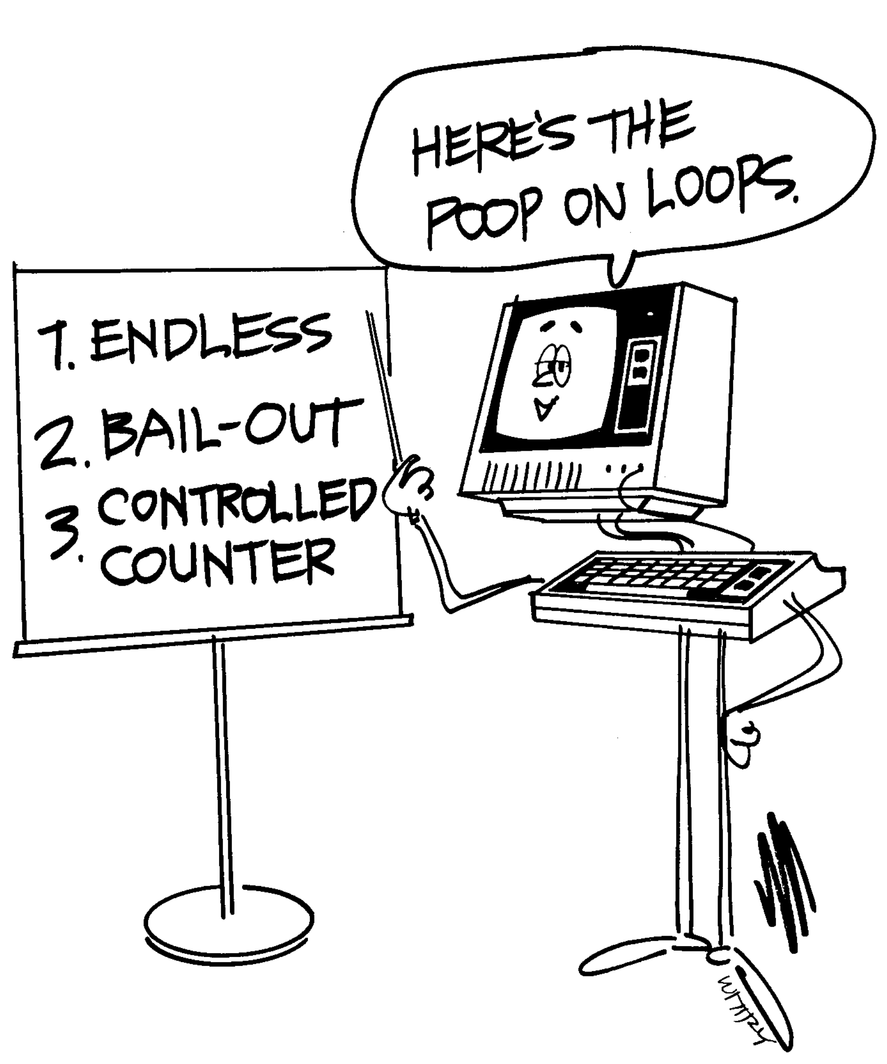
   a. square 73.2
   b. cube 16
   c. raise 43 to the 9th power

2. Again with TRS-80's assistance, check your above answers.

3. Which of the following is NOT a statement for changing the way BASIC decides what type a given variable is?

   a. DEFINT
   b. DEFISH
   c. DEFDBL
   d. DEFSTR

Chapter



Eleven

From Simple Loops Do Mighty Programs Grow . . .

We humans generally abhor repetition, finding it boring, tedious, frustrating, and wasteful. Machines, of course, thrive on it, and Computers are no exception.

When a Computer performs a sequence of operations over and over, we call this a **loop.** For example, type in this program.

```
10 A=1
20 PRINT 1/A
30 A=A+1
40 GOTO 20
```

But wait — before we RUN it, let's introduce a special Computer command designed to trace the flow of a program, i.e., to show which line is being executed at any given moment. It's called TRON (short for "trace on"). Go ahead and type in this command:

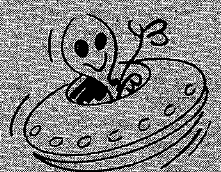
```
TRON
```

Now RUN the endless loop program again. See how each time TRS-80 starts a new line, it prints out the line number inside the <>.

Often in a program that's not working right, the trace will help you find out where it's going wrong.

The trace will remain on until you turn it off with TROFF (short for "trace off"). Simply press (BREAK) to stop the program, then type in the command:

```
TROFF
```



UFO #11-1.

To follow program flow, you can turn on TRS-80's trace function. Type:

```
TRON
```

Then BASIC will display the number of the current line, inside "<>" You can use TRON inside a program or in the **immediate** mode. It will remain on until you turn it off with:

```
TROFF
```

TROFF can also be used in a program or in the **immediate** mode.

There are three kinds of program loops:

1. Endless loops
2. Bail-out loops
3. Controlled counter loops

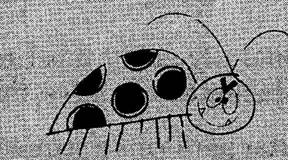
The simplest is the **endless loop,** in which a GOTO statement causes the loop to be executed over and over. Such loops are useful for continuous displays, as in:

```
10 PRINT "I LIKE TO RUN",
20 PRINT
30 GOTO 10
```

Often we use a very tight endless loop, to make the Computer stick at one point in a program without returning to the READY mode:

```
20 PRINT "PRESS <BREAK> TO GET CONTROL OF YOUR COMPUTER"
30 GOTO 30
```

The second kind of loop, **bail-out,** repeats itself until a certain condition is present or a test is passed. Once again, the looping is provided by a GOTO statement; the testing is done by an IF . . . THEN statement. Typically, you cannot tell in advance how many times such a loop will be executed before the "bail-out" occurs.



You can have a big problem with ?UL errors ("undefined lines") whenever you use loops, so always watch where you send that line.

Be sure to try running this program with the trace on (type TRON before you run it). Turn it off (TROFF) to see it run without all the line numbers.

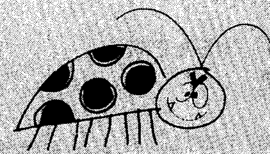
```
10 A$="I LIKE TO RUN"
20 PRINT A$
30 INPUT "DO YOU WANT TO QUIT (Y/N)"; R$
40 IF R$="Y" THEN END
50 GOTO 20
```

Notice that you can't predict how many times the loop will be executed, since that is determined by whoever is using the program.

You'll find bail-out loops useful whenever you input data from the keyboard or from DATA statements. That way, the loop doesn't need to know in advance how much data is coming. In the goldfish population analysis program, we used a bail out loop similar to this one:

```
5 K=0: TTL=0
10 READ A
20 IF A=-1 THEN 50
30 TTL=TTL+A: K=K+1
40 GOTO 10
50 PRINT "TOTAL="; TTL, "AVERAGE="; TTL/K
60 DATA 23, 45, 50, 30, 100, 10, -1
```

RUN the program.



Notice that you can add or remove DATA items, as long as the last one (and only the last one) is a −1. If you forget the −1, the program won't know when to stop reading data; this will cause an OD (out of data) error to occur.

Notice also that the program keeps track of how many items are read in; this count is stored in K. Each time another value (other than −1) is read in, K is increased by 1.

Program (Do It Yourself #11-1). Write a program that:

1. Accepts numbers from the keyboard until you type in a 999
2. PRINTS out the total and the average of the numbers.

Program (Do It Yourself #11-2). Use a controlled loop to find and print out the largest whole number n such that $n*n*n$ is less than 1110.

By the way, you could find such a number with a single statement, using the ↑ operator. Try it.

The last kind of loop, the **Controlled Counter**, is very useful when you know in advance how many times a loop needs to be repeated. Such loops feature a counter variable, an **initial value**, an **increment**, and a **limit value**.

At the beginning of the loop, the counter is set to its initial value. Each time the loop is executed, the counter variable is incremented and tested against the limit value. When the limit value is reached or exceeded, the program exits from the loop.

GOTO and IF . . . THEN can be used to form a controlled counter loop:

```
10 A=1
20 PRINT 1/A
30 A=A+1
40 IF A=10 THEN END
50 GOTO 20
```

A is the counter variable; 1 is the starting value; 1 is the increment, and 10 is the limit value. The program repeats lines 20, 30 and 40 until A=10.

By using negative increments ("decrements"), you can make the loop count down from a larger number to a smaller number, as in:

```
10 A=1000
20 A=A-1
30 PRINT A
40 IF A=0 THEN END
50 GOTO 20
```

which counts down from 999 to 0.

Or you can count by tens, tenths, or any other increment:

```
10 A=0
20 A=A+100
30 PRINT A
40 IF A=1000 THEN END
50 GOTO 20
```

which prints out the sequence 100, 200, . . , 1000.

Programs (Do It Yourself #11-3, 11-4, and 11-5). Write programs using GOTO and IF . . . THEN to:

11-3. Print all the even numbers from 0 to 256

11-4. Count from 0 to 10 by tenths (use an increment of 0.1), printing out every counter value.

Increment — that's Computer talk for "increase". When we increment a variable, we add something (the increment) to it.

What if the increment is smaller than zero? We can call it a "negative increment", a term that's popular with some employers. (A "wage cut" thus becomes a "negative increment".)

Negative increments are usually known as "decrements". When we decrease a counter variable, we "decrement" it.

Try RUNning this program with the trace on (TRON). Notice that the last line executed is line 40.

11-5. Count from 0 to 59 and then start over at zero . . . over and over, printing out every counter value.

Do It Yourself #11-5 consists of a controlled counter loop inside an endless loop. The inner loop is said to be "nested" inside the outer loop.

Automatic Controlled Counter Loops

In controlled loops, we know in advance how many times each loop is to be executed. Now wouldn't it be nice if we could tell TRS-80 in advance how many times to perform a loop? Then we could skip the incrementing and testing steps, and let the Computer keep count.

Well, your TRS-80 hasn't let you down yet, has it? So type in this program:

Be sure to try it with trace on, too (TRON ENTER).

```
NEW
10 FOR SC=0 TO 59
20 PRINT "SC="; SC
30 NEXT SC
```

Before running the program, try reading through it. What do you think it will print?

RUN it, then read through it again, this time with feeling!

Here's another one:

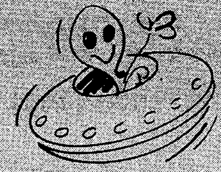
```
NEW
10 PRINT "THIS WILL TAKE ABOUT 10 SECONDS . . ."
20 FOR T=1 TO 3300
30 NEXT T
40 PRINT "THAT'S ALL"
```

If you try this one with trace on, it will take much longer than 10 seconds. That's because it has to do the extra work of printing all those line numbers on the display.

Note: If your "timer" is off, just try different values for the limit of T (i.e., instead of 3300).

How about counting down? Try this:

```
NEW
10 FOR I=100 TO 0 STEP -1
20 PRINT I,
30 NEXT I
```



UFO #11-2.
TRS-80 has a pair of statements that together form an "automatic" controlled counter loop:

FOR *counter=initial value* TO *limit value*

and

NEXT *counter*

counter is the numeric variable that holds the count.

initial value is the number where the counting starts.

limit value is the number where the counting stops.

The increment for counting will be 1, unless an alternate form of FOR statement is used:

FOR *counter=initial value* TO *limit value* STEP *increment*

increment is the number that will be added to *counter* each time the loop is repeated. When increment is **less** than zero, the loop counts **down** instead of **up**.

Suppose we have the following program loop:

```
FOR I=1 TO 10
NEXT I
```

then here's a summary of how the loop is executed.

1. The first time the Computer hits
   FOR *counter* = *initial value* TO *limit value* STEP *increment*, it sets *counter* equal to *initial value*. (I=1)
2. Execution then proceeds until it reaches NEXT *counter*
   Each time NEXT is reached, the *increment* is added to the *counter* (I=I+1)

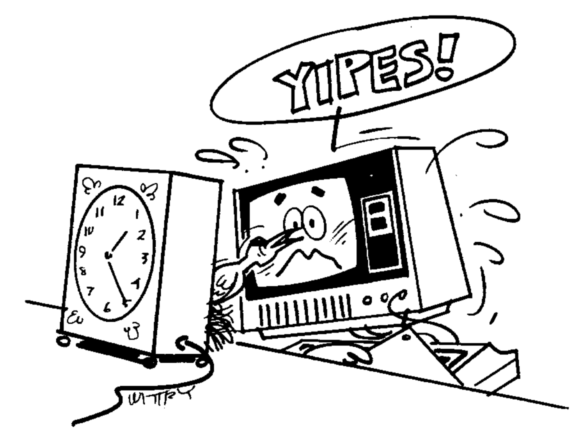
   counter is then compared with *limit value*. If it equals or exceeds *limit value* (I>=10) the loop ends and the program continues with the statement following NEXT.

   But if the counter has not yet reached *limit value* (I<10), the program loops back to the first statement following FOR.

Program (Do It Yourself #11-6, and 11-7). Revise D.I.Y.'s #11-3 and 11-4 to use FOR and NEXT instead of GOTO and IF... THEN.

Hey — There's a Nest in That Clock!

Remember the program (D-I-Y #11-5) to count from 0 to 59, and then start over? That one involved "nested loops" — one loop inside another one. Here's the same program, using a FOR/NEXT loop nested inside a GOTO loop.




```
10 FOR SC=0 TO 59
20     PRINT "SC=";SC
30 NEXT SC
40 GOTO 10
```

Lines 10, 20 and 30 form the nested loop; line 40 is executed once each time SC passes 59.

RUN the program. It simply counts from 0 to 59, over and over. That's what the second-hand on a stopwatch does. If only we could slow it down . . .

Well, what's the easiest way to slow down a watch? Hold the second hand, of course. Well, the "second-hand" in this program is the counter variable SC. So how do you "hold" a counter variable?

Hmmm.

How about inserting a one-second (approximate) pause between lines 20 and 30? That way, the Computer won't increment SC until a second has passed.

Try adding these lines:

```
25 FOR K=1 TO 330
27 NEXT K
```

Now the Computer will have to count from 0 to 330 before going on to increment the seconds. That should slow it down a bit.

RUN the program. If you have a watch with a second-hand, you can fine-tune the stopwatch program. If the seconds are ticking too fast, slow down the loop in line 25 by using an upper limit larger than 330. If the seconds are ticking too slowly, speed up the loop by using an upper limit smaller than 330.

Here's a listing of the resident program:

```
10 FOR SC=0 TO 59
20 PRINT "SC="; SC
25 FOR K=1 TO 80
27 NEXT K
30 NEXT SC
40 GOTO 10
```

Notice the program consists of a FOR/NEXT loop within a FOR/NEXT loop within a GOTO loop. Look closely at the sequence of the NEXT K and NEXT SC statements. Why does NEXT K come first? Well, because we want K to reach its final value before SC is incremented.

It's essential that the innnermost loop be completely contained inside the middle loop. In other words, when you open one loop inside another loop, you must close the inner loop before closing the outer loop:

Right!

```
10 FOR J=1 TO 3
20     FOR J2=1 TO 2
30     NEXT J2
40 NEXT J
```

Wrong!

```
10 FOR J=1 TO 3
20     FOR J2=1 TO 2
30 NEXT J
40     NEXT J2
```

RUN the "wrong" program; you'll get the error message, ?NF ERROR IN 40. NF is short for NEXT WITHOUT FOR.

The Computer gave you this message because you closed the outer loop (the J-loop) before closing the inner (J2) loop. The Computer assumed you meant to close all inner loops at the same time. So when it reached line 40, the J2-loop was no longer open.

Program (Do It Yourself #11-8). The stopwatch program only counts seconds. Add another controlled counter loop to make it count minutes, as well. When the minutes pass 59, reset them to zero.

Hint: Each time you have to reset the seconds, increment the minutes. You'll end up with an innermost FOR/NEXT loop, a middle FOR/NEXT loop, and an outermost GOTO loop:

Seconds counter — innermost loop
Minutes counter — middle loop
Minutes reset to zero — outer loop

Remember to close inner loops before outer loops!

So now you've got a computerized Stopwatch . . .

But I wanted a clock!

Roll Over, Big Ben

The following program will make full use of nested loops to provide an hours : minutes : seconds "clock".

```
10 INPUT "ENTER THE CURRENT HOUR"; HR
20 INPUT "ENTER THE CURRENT MINUTE"; MN
30 INPUT "ENTER THE CURRENT SECOND"; SC
40 FOR HR=HR TO 12
50     FOR MN=MN TO 59
60       FOR SC=SC TO 59
70          PRINT HR; ":"; MN; ":"; SC
80          FOR T=0 TO 330
90          NEXT T
100        NEXT SC: SC=0
110    NEXT MN: MN=0
120 NEXT HR: HR=1
130 GOTO 40
```

Often we'll indent the lines that are "nested" inside a FOR-NEXT loop, just to emphasize the fact.

Program (Do It Yourself #11-9). Change the clock program so that it keeps "military time", i.e. uses a 24-hour clock cycle.

Bad Idea (Do It Yourself #11-10). The very first program in this chapter showed an endless loop using GOTO.

```
10 PRINT "I LIKE TO RUN",
20 PRINT
30 GOTO 10
```

Make an endless loop using FOR/NEXT instead of GOTO. Ordinarily, this would be considered a "bug" in your program, but doing it will teach you something about how FOR/NEXT loops operate.

Bad Idea (Do It Yourself #11-11). Here's another educational mistake. Remember the bail-out program?

```
10 A$="I LIKE TO RUN"
20 PRINT A$
30 INPUT "DO YOU WANT TO QUIT (Y/N)"; R$
40 IF R$="Y" THEN END
50 GOTO 20
```

Suppose we write a FOR/NEXT version, by adding line 15 and changing line 50:

```
15 FOR I=1 TO 100
50 NEXT I
```

RUN the program. Type:

```
Y(ENTER)
```

when you're ready to quit. Seems to work fine, right? . . . Well, not exactly. You see, the loop never was properly ended. As far as the Computer is concerned, the loop hasn't yet been completed. It's waiting for another NEXT I statement so it can increment and loop again, until I reaches 100.

What we have here is a "pending FOR/NEXT loop", and in a program, this can easily cause problems and confusion. For example, type in this **immediate** line:

```
NEXT I
```

You've given the Computer the NEXT I it was waiting for, and now you're back in the program at line 30!

There is a way to "bail out" of a FOR/NEXT loop without leaving a pending loop. When you're ready to bail out, simply set the counter equal to the limit value, and go directly to the NEXT statement. that way, you exit the loop naturally and prematurely. For example, change line 40 in the resident program:

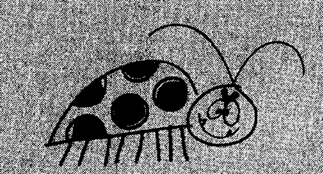
```
40 IF R$="Y" THEN I=100
```

Now RUN it, type Y (ENTER) to quit, then in the **immediate** mode test the Computer by typing:

```
NEXT I
```

You should get the message:

```
?NF ERROR
```



This simply indicates that you've issued a NEXT without first opening the loop with FOR. This proves that the program's loop was properly "closed".

UFO #11-3.
In general, you should only use FOR/NEXT loops to provide controlled counter functions. For endless loops, use GOTO; for bail-out loops, IF . . . THEN.

If you insist on bailing out of a FOR/NEXT loop, do it by setting the counter equal to the last value and executing the NEXT statement normally.

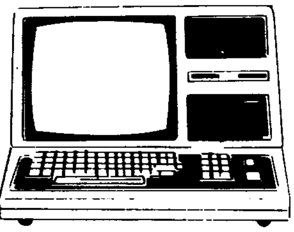
☑ Chapter Checkpoint #11

1. See if you can match the correct statement with the proper program loop.

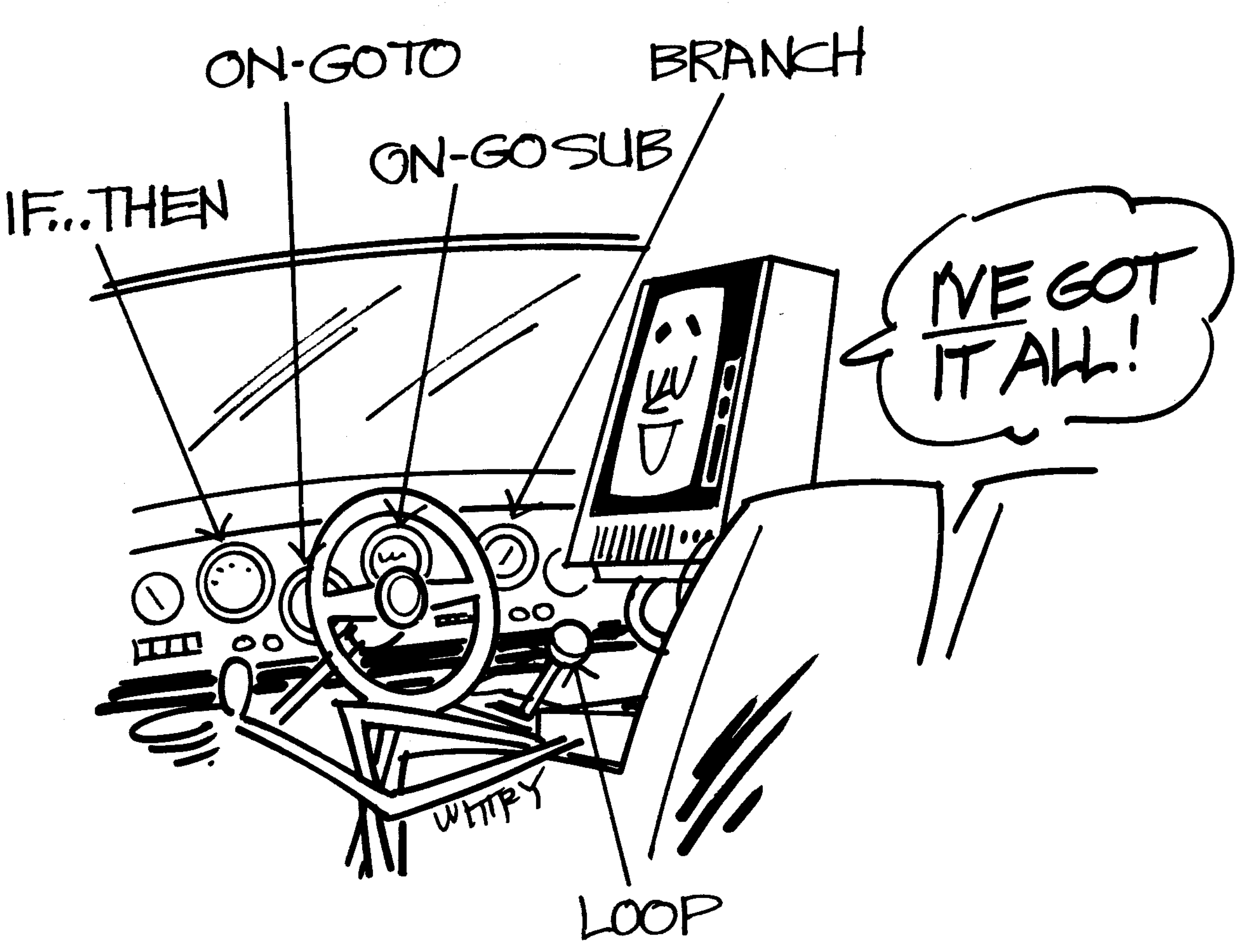
LOOP	STATEMENT
a. Endless	1. FOR/NEXT
b. Bail-out	2. GOTO
c. Controlled Counter	3. IF . . . THEN

2. TRS-80 will help you trace a program's flow if you use the command ________.

3. An ________ loop will be executed over and over until you press (BREAK).



Chapter



Twelve

Automatic Transmission with Power Steering

You now have the basic mechanism for running a program through nested loops, unconditional and conditional branches, dry creek beds, and performing many other stunts.

But good old TRS-80 has some additional features related to program sequence. Call them luxuries, creature comforts, or whatever. You'll find they make programming simpler by reducing the number of statements required for a given task.

We're going to cover them quickly in this Chapter, and then put them to use later.

IF... THEN... ELSE

You can add the word, ELSE, to IF *test* THEN *action*. This will allow a single program line to handle both cases—when the test passes and when it fails.

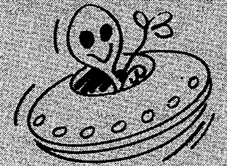
For example, this three-line program:

```
10 INPUT "ENTER TWO NUMBERS"; A, B
20 IF A<B THEN PRINT A; "<="; B : END
30 PRINT B; "<"; A
```

can be compressed into two lines using ELSE. DELETE line 30 and CHANGE line 20:

```
20 IF A<=B THEN PRINT A; "<="; B: ELSE PRINT B; "<"; A
```

Here's how TRS-80 interprets line 20:
"If A is less than or equal to B then PRINT A <= B;
Otherwise (ELSE) PRINT B< A."



UFO #12-1.

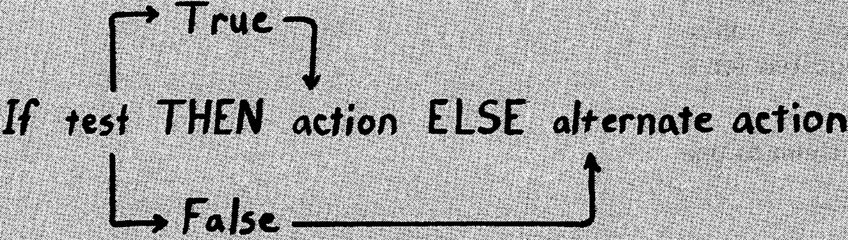
IF . . . THEN . . . ELSE lets you make a test and take either of two possible actions, depending on the outcome. Here's the general form:

IF *test* THEN *action* ELSE *alternate action*

test is a relational expression like A<B.
action is a BASIC statement or a line number.
alternate action is another statement or line number.

If the test is true, TRS-80 performs the action and then drops down to the next line in the program. If the test is false, your Computer skips over to alternate action and performs it.

Here's an illustration:



The alternate action can be any BASIC statement—even another IF. . . THEN. . . ELSE statement. For example:

```
10 INPUT "ENTER TWO NUMBERS"; A, B
20 IF A < B THEN PRINT A; "<"; B: STOP: ELSE IF B < A THEN PRINT B;
  "<"; A: STOP
30 PRINT A; "="; B
```

Translate the new line 20 this way: "If A is less than B then PRINT A<B and stop; but if B is less than A then PRINT B<A and stop. If neither test passes, then go on to the next line."

Multiple Branches

Consider a program that starts out like this:

```
10 PRINT "TYPE 1 FOR SALES REPORT"
20 PRINT "TYPE 2 FOR P & L STATEMENT"
30 PRINT "TYPE 3 FOR A PAYROLL LISTING"
40 INPUT C
```

After you type in a value for C, the program must find out which option you selected. These lines would work:

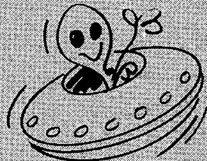
```
50 IF C=1 THEN 1000
60 IF C=2 THEN 2000
70 IF C=3 THEN 3000
80 GOTO 10
1000 PRINT "SALES REPORT . . .": END
2000 PRINT "P & L STATEMENT . . .": END
3000 PRINT "PAYROLL LISTING . . .": END
```

RUN the program. Under what conditions will line 80 be executed? What would happen if you omitted line 80? Try it.

Lines 1000, 2000, and 3000 are just dummy lines—there to illustrate a point. The sales report program would start at 1000, P & L at 2000, etc.

Now DELETE lines 50-70, and type in this one:

```
50 ON C GOTO 1000, 2000, 3000
```



UFO #12-2.

ON. . . GOTO is a multi-way branch statement, allowing you to branch to one of several line numbers, depending on the test value. Its general form is:

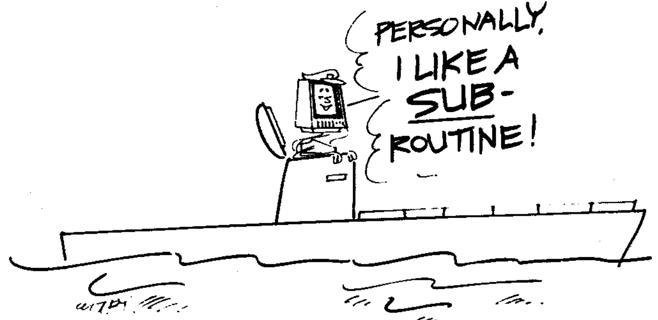
ON *test value* GOTO *line number list test value* is a numeric variable or expression whose value is between 0 and 256. The line numbers in *line number list* are separated by commas.

If test value equals 1, TRS-80 branches to the first number in the line number list; if test value equals 2, the Computer branches to the second line number in the list, etc.

If *test value* equals zero or exceeds the number of elements in the list, control falls through to the next line in the program.

Here's an illustration:

```
            =3 ↱ GOTO line-ref 3
            =2 ↱ GOTO line-ref 2
            =1 ↱ GOTO line-ref 1                =n ↱ GOTO line-ref n
ON test value GOTO line-ref 1, line-ref 2, line-ref 3, . . .
      ↳ = Ø or > the number of line-refs
                                      ↳ GOTO next line
```



Routines and Subroutines

Both these terms refer to sequences of instructions which perform a fairly specific function for a main program. They differ in how they are entered and in what happens when they are completed.

Actually, the "routine" is just a programming concept created to help you organize programs better. We are using the term for the sake of contrast with subroutines. The "subroutine" in BASIC is more than a concept; it refers to a specific set of BASIC statements.

Routines are accessed by branch-type statements. For example:

```
IF A<B THEN 100 ELSE 200
```

can access routines starting at 100 or 200.

```
ON A GOTO 1000, 2000, 3000
```

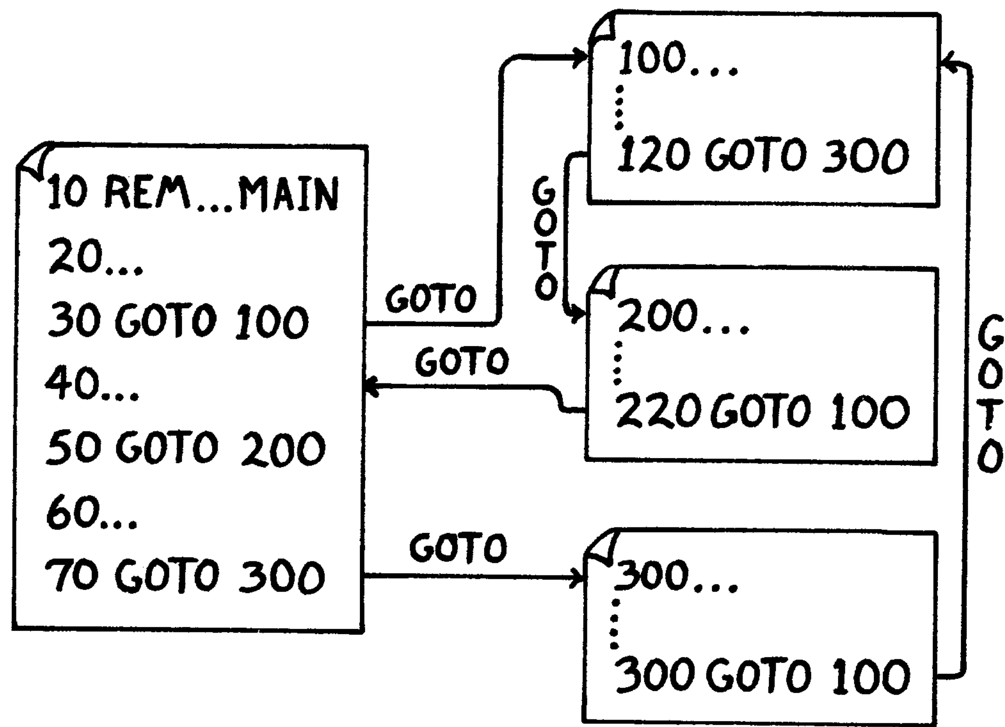
can access routines starting at 1000, 2000 and 3000.

At the end of a routine, we place another branch instruction to get back to the main program or to some other routine. For example:

```
10 PRINT "TYPE 1 FOR SALES REPORT
20 PRINT "TYPE 2 FOR P & L STATEMENT
30 PRINT "TYPE 3 FOR A PAYROLL LISTING
40 INPUT C
50 ON C GOTO 1000, 2000, 3000
80 GOTO 10
1000 PRINT "SALES REPORT . . .": GOTO 10
2000 PRINT "P & L STATEMENT . . .": GOTO 10
3000 PRINT "PAYROLL LISTING . . .": GOTO 10
```

Lines 10-80 constitute the main program; routines are at lines 1000, 2000 and 3000.

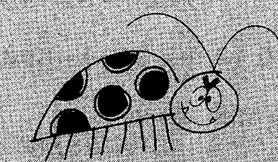
Here's an illustration of how routines work:



Subroutines are accessed exclusively by a special BASIC statement, GOSUB. They differ from routines in that they always end with a RETURN statement, which transfers control to the statement following the GOSUB.

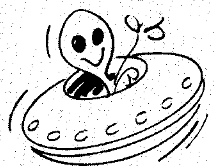
Try this program:

```
NEW
10 PRINT "SUBROUTINE DEMONSTRATION"
20 S=0
30 GOSUB 1000
40 S=S+1
50 PRINT "SECONDS=";S
60 GOTO 30
1000 PRINT "EXECUTING DELAY SUBROUTINE . . ."
1010 FOR I=1 TO 330
1020 NEXT I
1030 RETURN
```



Line 60 is very important; without it, TRS-80 would crash into the subroutine uninvited by a GOSUB. This is always an error; when the Computer hits the RETURN statement, it will give you an RG (Return without GOSUB) error.

Lines 10-60 form the main program; lines 1000-1030 form the subroutine.



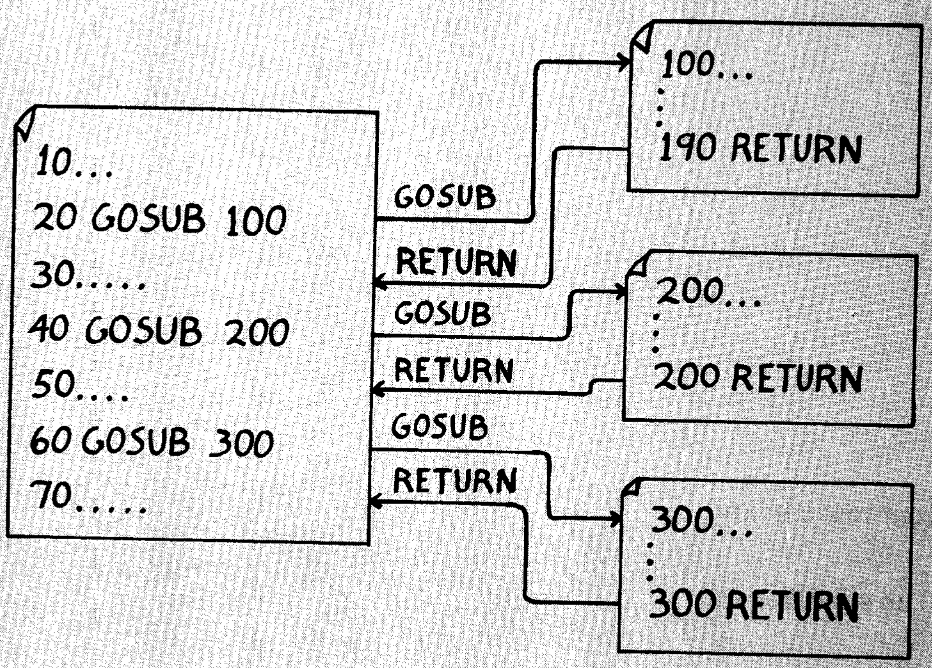
UFO #12-3.

GOSUB transfers control to a subroutine; RETURN ends the subroutine and returns control the statement following the GOSUB.

Two very important rules about subroutines:

1. Use END statements or GOTO statements directly ahead of the subroutine to ensure that BASIC can enter the subroutine only via a GOSUB.
2. Every subroutine must end with a RETURN.

Here's an illustration of a main-program/subroutine structure:



When the subroutine ends, it always returns control to the statement following the GOSUB.

Multiway Subroutine Calls

Remember ON. . . GOTO? Quite a powerful little statement, once you understand it. It allows us to branch to any of several routines, depending on a test value.

The ON. . . GOSUB statement is very similar. It lets us call any of several subroutines, depending on a test value.

Try this program:

```
NEW
10 PRINT "TYPE 1 FOR SQUARE ROOT TABLE"
20 PRINT "TYPE 2 FOR CUBE ROOT TABLE"
30 PRINT "TYPE 3 FOR FOURTH ROOT TABLE"
40 INPUT C
50 ON C GOSUB 100, 200, 300
60 GOTO 10
100 FOR I=1 TO 50
110 PRINT I; SQR(I),
120 NEXT I
130 RETURN
200 FOR I=1 TO 50
210    PRINT I; I ^ (1/3)
220 NEXT I
230 RETURN
300 FOR I=1 TO 50
310    PRINT I; I ^ (1/4),
320 NEXT I
330 RETURN
```

Exercise (Do It Yourself #12-1, #12-2, #12-3).

#12-1 Diagram an ON. . . GOTO statement using A, B, and C for your routines and arrows for the branches.

#12-2 Diagram an ON . . . GOSUB statement using A, B and C as subroutines. (Don't forget to include GOSUB and RETURN!)

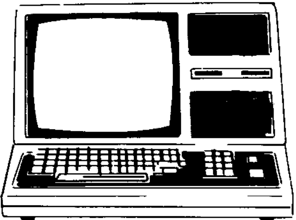
#12-3 Diagram the "Square Root, Cube Root, and Fourth Root Table" program that you just ran.

☑ Chapter Checkpoint #12

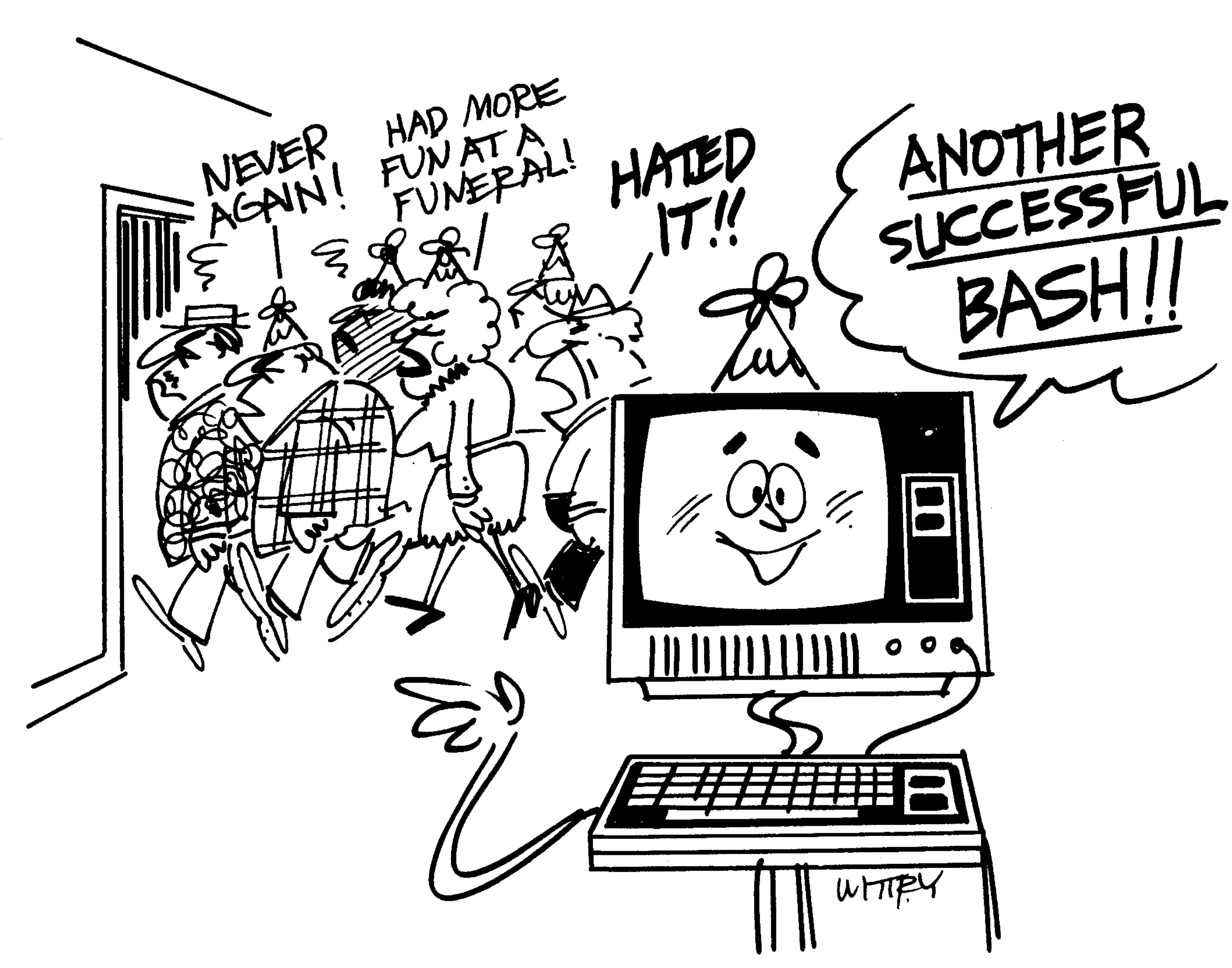
1. If you want to make a test and take either of the two possible outcomes, then you can add ________ to the IF. . . THEN statement.

2. The statement ON. . . GOTO will allow you to use:
   a. a new language
   b. multiple branches
   c. branch water

3. You must enter every subroutine with ________ and exit from it with ________.



Chapter



Thirteen

Automatic Party Planner

We're having a party for six rather difficult individuals. Each has strong feelings about one or more of the others.

1. JOHN won't come unless SUE's there.
2. KARL won't come unless MARY's there.
3. RICK won't come without ELLA.

So far, no problem. Just invite the three couples and have plenty of refreshments. But wait, there are complications:

4. ELLA won't come if SUE is present.
5. MARY can't stand RICK or ELLA.
6. SUE won't come unless KARL or RICK is present.

Well, that's one problem you'll never have with TRS-80. Computers will go to any party, anywhere, and they don't care *who* is there!

A quick look at 4 and 5 tells us we can't have all six people to the same party. We're going to have to make up some trial invitation lists and see which ones satisfy all six requirements.

Sounds like a boring task. . . hmmm. . . that often means it's a good job for a computer.

Boring. . .? Maybe that's why poor TRS-80 never gets invited to any parties. Did you ever try taking a Computer to lunch?

First we'll get the computer to come up with every possible list, using some nested FOR/NEXT loops. We'll use six variables

JOHN KARL RICK SUE MARY ELLA

When a person is on the list, we'll store -1 in the corresponding variable. When a person is not on the list, we'll store 0 in the variable.

We could use other constants instead of -1 and 0; but these happen to match the Computer's internal method of True/False Evaluation.

Type in these lines (Don't forget to use the AUTO command so you won't have to type in all those line numbers):

```
10 YES = -1: NO=0
20 FOR JOHN=YES TO NO
30  FOR KARL=YES TO NO
40   FOR RICK=YES TO NO
50    FOR SUE=YES TO NO
60     FOR MARY=YES TO NO
70      FOR ELLA=YES TO NO
```

Now press (BREAK) so we can skip some line numbers. Type: AUTO 400 and then enter these lines:

We're leaving lots of room in the middle of the program to hold the list-examination logic.

```
400  REM . . . DISPLAY THE CURRENT LIST
410  PRINT "HERE'S THE TRIAL LIST:": PRINT
420  IF JOHN=YES THEN PRINT "JOHN",
430  IF KARL=YES THEN PRINT "KARL",
440  IF RICK=YES THEN PRINT "RICK",
450  IF SUE=YES THEN PRINT "SUE",
460  IF MARY=YES THEN PRINT "MARY",
470  IF ELLA=YES THEN PRINT "ELLA",
480  PRINT
(BREAK)
```

And now one line to close all the loops:

```
500 NEXT ELLA, MARY, SUE, RICK, KARL, JOHN
```

RUN the program—just to make sure it's working so far. It should print out 64 lists (the last one is an empty list—a party to which no one is invited).

Now let's find out which lists will satisfy all the rules.

We'll start by converting Rule 1 into a Computer BASIC statement:

1. JOHN won't come unless SUE's there.

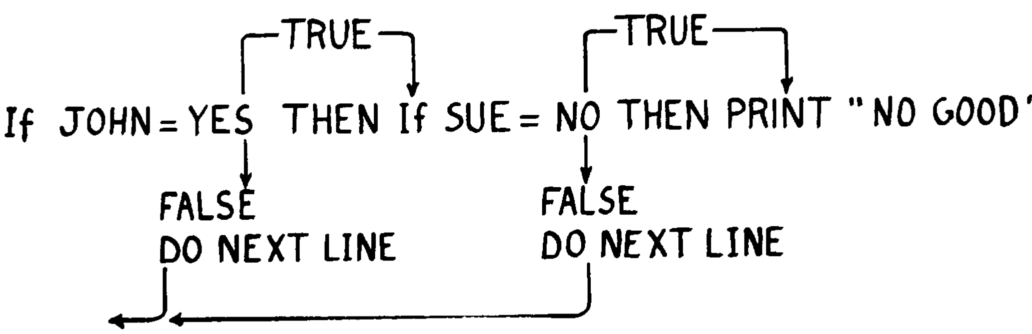
Now what condition will violate this rule? If SUE is invited and JOHN is not. . . that's okay (JOHN will never know the difference). But if JOHN is invited and SUE is not. . . *that's* a violation.

Here's a BASIC statement that checks for this condition (Don't type this in, just look at it):

```
IF JOHN = YES THEN IF SUE = NO THEN PRINT "NO GOOD"
```

We only check SUE's status if JOHN is invited. Otherwise, we don't care.

The action-part of this IF *test* THEN *action* is yet another IF *action* THEN *test* statement. That can be a little tricky to keep in mind. Here's a diagram:



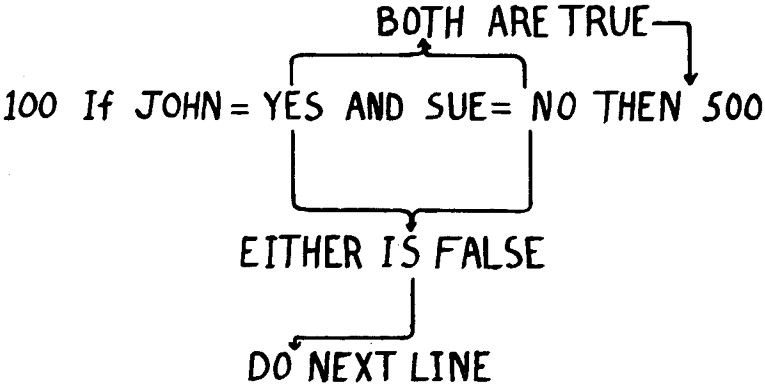
There's another way to accomplish this "double" test. It involves another one of those good old plain-English features that make BASIC easy to learn... it's called AND.

Remember our restatement of Rule 1: If John is invited AND Sue is not... well, let's translate that into BASIC (go ahead and type in this line):

```
100 IF JOHN=YES AND SUE=NO THEN 500
```

If both tests "pass" (JOHN = YES AND SUE = NO), then TRS-80 will skip the list-printing lines and get the next list. If either test does not pass (that is, if JOHN is not equal to YES or SUE is not equal to NO), then your Computer will proceed to the next line in the program.

Here's a diagram of how this line works:



So far the program checks for violations of Rule 1. It's easy to add checks for Rules 2 and 3. Here's Rule 2:

2. KARL won't come unless MARY's there.

Following the example of lines 300-310, it's easy to add this logic to the program:

```
110 IF KARL = YES AND MARY = NO THEN 500
```

Program (Do It Yourself #13-1). Add a check for Rule 3 violations:

3. RICK won't come without ELLA.

by adding line 120.

```
120 ________________________________________
```

Here's our working program:

```
10 YES=-1:  NO=0
20 FOR JOHN=YES TO NO
30  FOR KARL=YES TO NO
40   FOR RICK=YES TO NO
50    FOR SUE=YES TO NO
60     FOR MARY=YES TO NO
70      FOR ELLA=YES TO NO
100         IF JOHN=YES AND SUE=NO THEN 500
110         IF KARL=YES AND MARY=NO THEN 500
120         IF RICK=YES AND ELLA=NO THEN 500
400         PRINT: PRINT "HERE'S THE TRIAL LIST . . . "
410         IF JOHN=YES THEN PRINT "JOHN",
420         IF KARL=YES THEN PRINT "KARL",
430         IF RICK=YES THEN PRINT "RICK",
440         IF SUE=YES THEN PRINT "SUE",
450         IF MARY=YES THEN PRINT "MARY",
460         IF ELLA=YES THEN PRINT "ELLA",
470         PRINT
500 NEXT ELLA, MARY, SUE, RICK, KARL, JOHN
```

Now for the really hard to please. . .

Rule 4 goes like this:

4. ELLA won't come if SUE is present.

This rule will be violated if we have ELLA and SUE to the party. Either one alone will be okay.

Program (Do It Yourself #13-2). Add a check for violations of Rule 4, by adding line 130.

130____

Rule 5 goes like this:

5. MARY can't stand either RICK or ELLA.

This one's a little more complicated. There are three people involved. We've got to be sure that MARY doesn't see RICK or ELLA.

We could handle this logic with two separate IF. . . THEN statements:

```
IF MARY = YES AND RICK = YES THEN 500
IF MARY = YES AND ELLA = YES THEN 500
```

But there's an easier way, involving another plain-English word called OR (Notice that Rule 5 contains OR instead of AND).

```
IF MARY = YES THEN IF RICK = YES OR ELLA = YES THEN 500
```

If MARY is present, then BASIC will check to see if either RICK or ELLA is present.

Let's add this logic to the program:

```
140 IF MARY = YES THEN IF RICK = YES OR ELLA = YES THEN 500
```

TRS-80 will go to line 500 only if all three tests pass. If any one of them fails, our Matchmaker will do line 400 instead.

Rule 6 is the toughest yet:

6. SUE won't come unless KARL or RICK is present.

If Sue is present, then either KARL or RICK must also be there. It's okay if both of them come, as well. If Sue is invited and neither KARL nor RICK are, we've got problems.

```
IF SUE = YES THEN IF KARL = NO AND RICK = NO THEN 500
```

This will work, but there's another BASIC word that allows us to make the statement more closely resemble the rule. The word is NOT.

Type in this line:

```
150 IF SUE = YES AND NOT (KARL = YES OR RICK = YES) THEN 500
```

In plain English, line 150 says, "If SUE is invited AND it is NOT true that KARL OR RICK is invited, then get the next list (go to line 500).

UFO #13-1

BASIC has three special words for performing TRUE/FALSE tests. These words are:

AND
OR
NOT

They are usually used to connect tests in IF. . . THEN statements:

IF *test* AND *test* THEN *action*
IF *test* OR *test* THEN *action*
IF NOT *test* THEN *action*

AND takes two tests and checks whether both pass or not. If either test fails, then "*test* AND *test*" fails; if both tests pass, then "*test* AND *test*" passes.

OR takes two tests and checks whether either test passes or not. If at least one passes, "*test* OR *test*" passes; if both tests fail, then "*test* OR *test*" fails.

NOT takes a single test and checks whether it passes or not. If the test fails, "NOT *test*" passes; if test passes, "NOT *test*" fails.

☑ Chapter Checkpoint #13

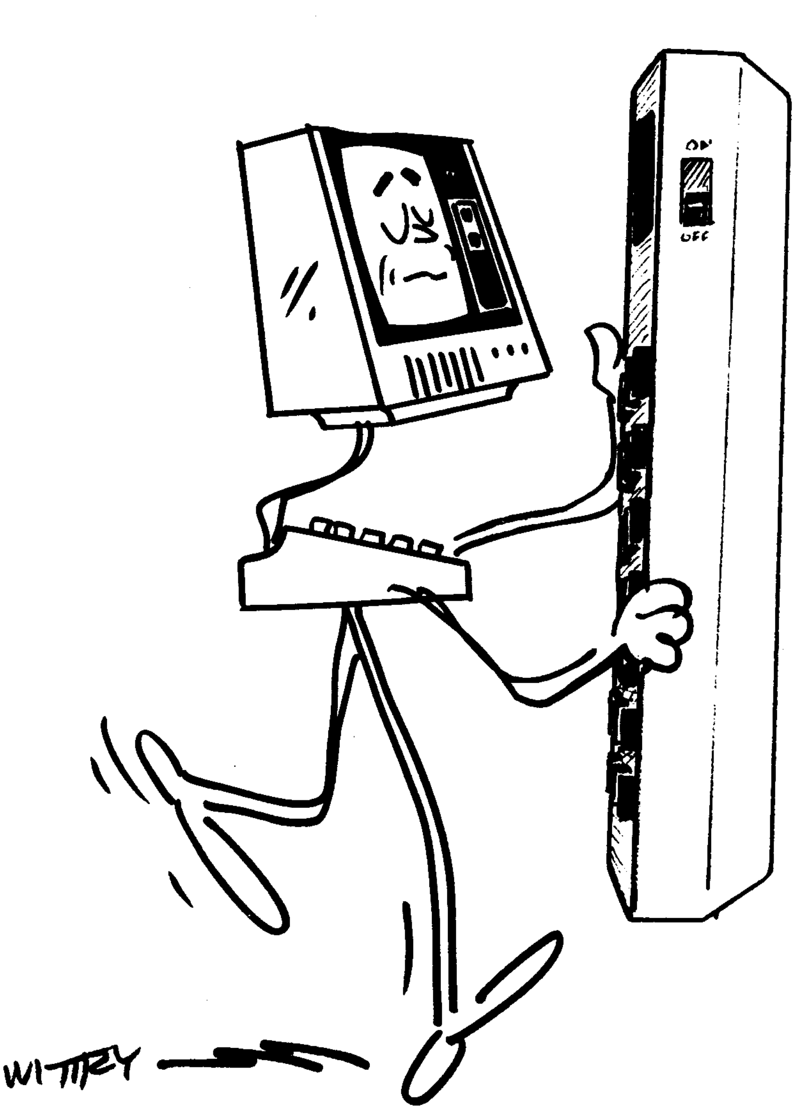
1. Without testing these on your Computer, try to predict which ones will print the message "PASSED!"
   a. `IF (1=1) AND (2=3) THEN PRINT "PASSED!"`
   b. `IF (1=1) OR (2=3) THEN PRINT "PASSED!"`
   c. `IF NOT (1=1 AND 2=3) THEN PRINT "PASSED!"`
   d. `IF NOT (1=1) OR (2=3) THEN PRINT "PASSED!"`
   e. `IF NOT (NOT 1=1 OR 2=3) THEN PRINT "PASSED!"`

2. Each test in column I is equivalent to one test in column II. Match them up.

I.	II.
a. `A<B`	1. `NOT (A>B)`
b. `B =A`	2. `(B<>A) AND NOT (A>B)`
c. `A < = B`	3. `NOT (B<A) AND B<=A`

Compare your answers with ours in the Appendix.

Chapter



Fourteen

A Great Big Calculator

Now we'll put some of our tools to work by designing a calculator program. After we're done, you should have a much better idea of how to create your own programs. If the math gets too heavy, just skim through it—it's not as important as the overall process we're explaining.

Sure, you can get cheaper ones, but this one isn't going to get stolen from your back pocket.

It's important to **define** a program both carefully and precisely before you start so you don't wind up with a disorganized, inefficient and hard-to-understand sequence of BASIC statements.

First the program description.

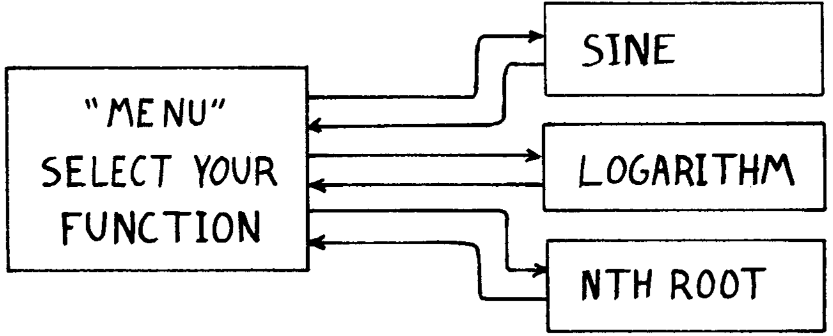
The program should:

1. Let you select any of the following functions:
   Sine
   Logarithm
   Power
2. Ask you to type in the argument(s).
3. Compute and print out the result.
4. Return to Step 1.

The program can be broken into two major sections.

1. The "menu", which tells you what is available and asks you to select a choice.
2. The three calculator routines.
   - each routine will ask you for an input value
   - compute a result
   - print out that result

Here's a diagram:



> This is a little like the Dewey Decimal System you'll find in your local library. It is just a convenient way of categorizing your program by alotting certain number ranges to certain "subjects."

Just to keep things neat, we'll plan our line numbers:

Line range	**Function**
100-199	Menu
200-299	Sine
300-399	Logarithm
400-499	Power

Now that we've laid out the program organization, we can get down to coding.

The following lines display the menu and ask for a selection or choice:

```
110 PRINT "TYPE <1> FOR SINE"
120 PRINT "     <2> FOR LOGARITHM
130 PRINT "     <3> FOR NTH POWER
140 INPUT CHOICE
```

> Why didn't we use GOSUBs and RETURNs to get from the menu to the calculator routines and back?
>
> GOSUB and RETURN are primarily for situations where the routine that's called has no way of knowing "who" called it in the first place—so it doesn't know where to return to.
>
> But looking at the diagram again, you'll see that there's never any question as to who called a particular routine—it's always the menu that does the calling. So each calculation routine must return to the menu when it is done. IF. . .THEN and GOTO are sufficient for this situation.

Once we've got a selection, we make sure it's valid. If it is, we want to transfer control to the appropriate calculation routine. This is a perfect place for ON. . . GOTO. We're only going to provide for a branch to a single line number, since, at the beginning, only sine operations will be available.

```
150 ON CHOICE GOTO 200
190 PRINT: GOTO 110
```

Now for sine calculation routine:

```
200 INPUT "INPUT ANGLE IN DEGREES"; A
210 PRINT SIN(A/57.29578)
220 INPUT "PRESS ENTER TO CONTINUE"; X$
230 GOTO 110
```

> Why do we divide angle A by 57.29578? Because the TRS-80 BASIC SIN function expects the input angle to be in radians, but A is in degrees. To convert degrees, we divide as in line 210.
>
> Degrees and radians are two alternate units of angle measure. For details, look in a trigonometry text book.

These lines check for a 1 (sine) selection, and provide the subroutine to do the sine calculation. Notice that if CHOICE is not 1, the program starts over at the top of the menu. In programmer talk, options 2 and 3 have not yet been "implemented."

What's the need for line 220? Try omitting it and see what happens.

Program (Do It Yourself #14-1). Change line 150 to check for options 2 and 3. Add the appropriate routines at 300 and 400.

When you run the program, type 1 (ENTER) to start the sine routine, 2 (ENTER) for logs, etc.

Hint: For the power function, you'll need to input two values, X and Y. Then the program simply prints X↑Y. Write the changes and new lines here:

____

____

____

____

____

____

____

____

Be sure to complete and RUN the program before turning the page. Then check it against ours on the next page.

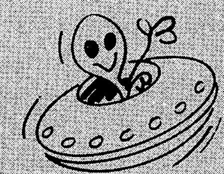
Here are the lines we added and changed:

```
150 ON CHOICE GOTO 200, 300, 400
300 INPUT "ENTER THE NUMBER (POSITIVE ONLY)"; X
310 PRINT LOG(X)
330 GOTO 110
400 INPUT "FOR X TO THE Y POWER, ENTER X"; X
410 INPUT "NOW ENTER Y"; Y
420 PRINT X ↑ Y
430 GOTO 110
```

RUN the program. For certain input values, you'll get an FC (function call) error message. This means that the input value was invalid for the function you selected. Simply RUN the program again with new input values.

Which points out something lacking in the program: it doesn't check for invalid arguments. That leaves the program wide open to input errors which terminate the program with an abrupt error message.

If you're trying to impress someone with a program you've written (or even more importantly, sell it to them), you certainly don't want a mistaken input value to blow you out of the program.



UFO # 14-1.
Always strive to make your programs idiot-proof. This means that each time a user inputs to a program, the program should make sure the input value is in range before trying to use it. If it's out of range, the program should ask for another value.

Here are the limitations of the calculator routines:

Routine	Acceptable Input Range
Sine	Any number
Logarithm	Any number greater than zero

Run the program. Type 1, 2, or 3 and press ENTER.

For the sine calculator, try these input values:
15, 30, 45, 60, 75, 90, 180, 270, 0.

For the logarithm calculator, try these:
1, 2.718281, 5, 10, 150, 0

For the power calculator, try these:

X	Y
2	5
2	.5
5	2
5	−3
−8	5
−8	.5
110	35

Power	If X is negative, Y must be an integer. Furthermore, X ↑ Y can easily produce an overflow error (a value greater than 1.7 E38). For example, X = 100, Y = 35 causes an OV ERROR.

Input checking for sine routine: None needed.

Input Checking for the logarithm routine: See next D.I.Y. Program.

Input checking for the power routine: First we need to determine if Y is a whole number or not. Remember the FIX function? You tried it out in D.I.Y. #10-5. It simply returns the whole-number portion of the argument. If X = FIX(X), we know that X is a whole number.

If we do get a non-integer Y, we then need to find out whether X is negative. If it is, we have invalid input.

Take a look at this fancy statement:

```
IF Y <> FIX(Y) THEN IF X < 0 THEN GOSUB 900
```

Here's how the statement works. When Y is a whole number, the first test fails, therefore program control drops down to the next line.

But if the test passes (Y is not a whole number), control continues on the THEN action part of the statement. In this case, the action is yet another IF... THEN statement.

This second IF... THEN works just like the earlier one: if the test fails (X is not less than zero), control drops down to the next line in the program; if the test passes, control continues over to the action clause, which in this case is a subroutine call.

Add these lines to the program:

```
414 R$="IF X IS NEGATIVE, Y MUST BE INTEGER"
416 IF Y <> FIX(Y) THEN IF X < 0 THEN GOSUB 900: GOTO 400
```

Notice that the program calls a subroutine at line 900 when you input incorrect values for X and Y. The plan is to have one error-handler serve both the logarithm and power routines. Since the routine won't know "who" called it, it is appropriate to use GOSUB instead of GOTO. That way a RETURN will get back to the calling program.

Add this error-handling subroutine:

```
900 PRINT "ARGUMENT(S) OUT OF RANGE:"
910 PRINT R$: PRINT
920 RETURN
```

RUN the program. Try inputting a negative X and non-integer Y. Notice that R$ stores a message which gives the input range, and that the subroutine at 900 uses this value.

Program (Do It Yourself #14-2). Add input checking lines for the logarithm routine. Don't forget to assign a value to the R$ parameter.

Be sure to test your program with invalid inputs for the logarithm routine.

Here is the complete program, with our answer to the last D.I.Y. Program.

```
110 PRINT "TYPE<1> FOR SINE"
120 PRINT "     <2> FOR LOGARITHM
130 PRINT "     <3> FOR NTH POWER
140 INPUT CHOICE
150 ON CHOICE GOTO 200, 300, 400
190 PRINT: GOTO 110
200 INPUT "INPUT ANGLE IN DEGREES"; A
210 PRINT SIN(A/57.29578)
230 GOTO 110
300 INPUT "ENTER THE NUMBER (POSITIVE ONLY)"; X
304 R$="X MUST BE POSITIVE"
306 IF X <=0 THEN GOSUB 900: GOTO 300
310 PRINT LOG(X)
330 GOTO 110
400 INPUT "FOR X TO THE Y POWER, ENTER X"; X
410 INPUT "NOW ENTER Y"; Y
414 R$="IF X IS NEGATIVE, Y MUST BE INTEGER"
416 IF Y <> FIX(Y) THEN IF X < 0 THEN GOSUB 900: GOTO 400
417 IF X=0 THEN 420
418 IF ABS(Y * LOG (ABS(X))) < LOG (1E38) THEN 420
419 R$="RESULT TOO LARGE": GOSUB 900: GOTO 400
420 PRINT X ^ Y
```

```
430 GOTO 110
900 PRINT "ARGUMENT(S) OUT OF RANGE:"
910 PRINT R$: PRINT
920 RETURN
```

Notice we added three extra lines to the power subroutine, lines 417-419. These lines prevent an overflow error from occurring should you type in X and Y such that X ↑ Y is equal to or greater than 1E38.

Line 417 is required to prevent an FC ERROR in line 418 when X equals zero. Line 418 uses this reasoning:

X ↑ Y<=1E38 only if LOG (X ↑ Y)<=LOG(1E38)

This last inequality is equivalent to:

Y * LOG (X) <= LOG (1E38)

We use the ABS function to prevent a function call error when X < 0, and ensure that a negative number (X ↑ Y) is greater than −1E38.

☑ Chapter Checkpoint #14

1. Before you start writing a program, you should (choose one or more):

   a. Do 20 push-ups and 15 deep kneebends.
   b. See if someone else has already written the program you need.
   c. Describe in very specific terms what you want the program to do.

2. List the three functions available on your TRS-80 calculator (the program developed in this chapter).

   ____

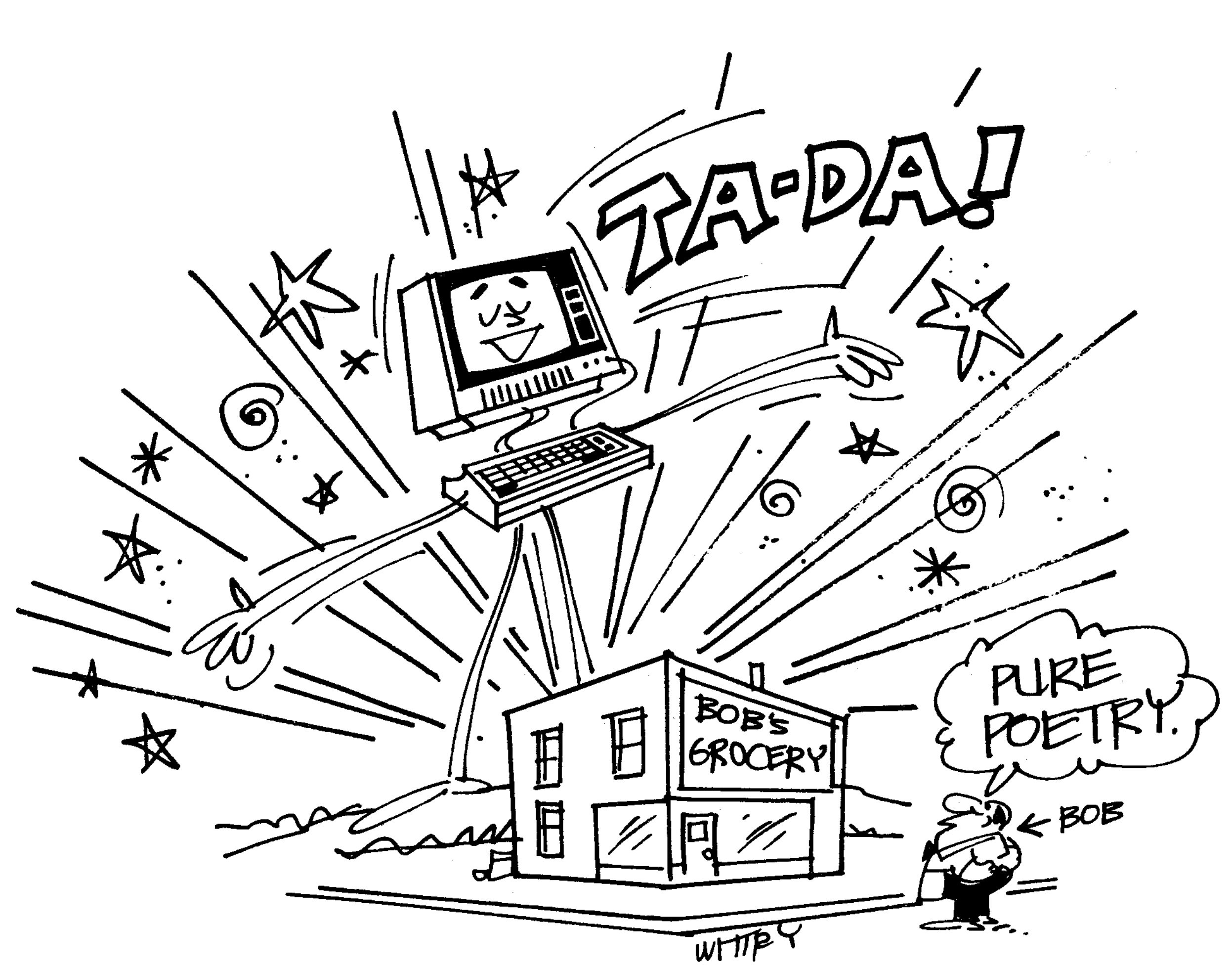
   ____

   ____

   List four other functions that you (or someone with the time, patience and taste for math. . .) could easily add to the calculator program. **Hint:** Look at the list in DIY # 10-5.

3. Describe at least one of the calculator program's limitations (Talk about loaded questions. . .)

Chapter



Fifteen

Bob's Corner Grocery Enters the Computer Age

This is the story of how progress changed the life of Bob Babson, a humble merchant. By typing in the programs in this chapter, you can relive Bob's experiences.

You are about to experience a totally new dimension in literary appreciation. . . computer-assisted fiction!

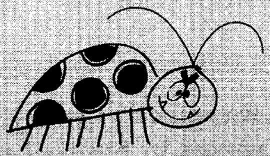
For 25 years, Bob ran a peculiar little grocery store. Peculiar because it carried only three items:

Item #1	MILK
Item #2	EGGS
Item #3	FLOOR WAX

His customers liked it, too, because it was always easy to find what they wanted, be it milk, eggs, or floor wax.

Bob liked it that way, because it simplified store operation and left him with more time for his favorite pastime, poetry.

But as the years passed, Bob became so engrossed in his poetic craft that the weekly process of taking inventory and re-ordering was an unbearable interruption.



Arrays seem to attract two kinds of "bugs".

The first error you might encounter is the ?OM Error, short for "Out of Memory".

The second is the ?BS Error which indicates a "bad" or out-of-range subscript. (More on subscripts later.)

The First Step. . .

One day Bob read about a new invention called the TRS-80 Microcomputer. "*Use it for inventory control—save hours of time!*", the ad started out. He threw down the magazine and ran right out to buy a TRS-80. Only when he returned with the Computer and plugged it in did he begin to understand those last words in the Radio Shack ad, "*Must be programmed first. . .*".

After reading the first 14 chapters of the nifty programming book that came with his Computer, Bob took out his pocket notebook and wrote down a definition of his problem:

1. *Too much time devoted to inventory!*
2. *Not enough time left for poetry!*
3. *Need a way to predict how much milk, eggs and floor wax to order once a month. No more weekly orders. (Will require VERY fresh milk!)*

Then he sat down at his Computer and typed in this program:

```
10 INPUT "HOW MUCH MILK WAS SOLD LAST MONTH"; MILK
20 INPUT "HOW MANY DOZEN EGGS"; EGGS
30 INPUT "HOW MUCH FLOOR WAX"; WAX
40 CLS
50 INPUT "TYPE 1 FOR MILK, 2 FOR EGGS, 3 FOR WAX"; A
60 PRINT "ORDER THIS MUCH:"
70 IF A=1 THEN PRINT MILK
80 IF A=2 THEN PRINT EGGS
90 IF A=3 THEN PRINT WAX
100 PRINT: GOTO 50
```

Type in Bob's inventory control program and RUN it.

The program was pure. . . poetry?. . . well, no, not quite. But it *did* give Bob more time to pen verses in his pocket-sized notebook. He felt that his TRS-80 was earning its keep.

The Dawning of a New Era

One day an unemployed BASIC programmer named Ray came in looking for some floor wax. He noticed the TRS-80, and got to talking to Bob about the inventory control program. When Bob listed the program for him, Ray chuckled and asked if he could dress it up a bit. Here's what he typed:

```
NEW
10 DIM G(3)
20 PRINT "ITEM 1=MILK, ITEM 2=EGGS, ITEM 3=WAX"
30 FOR I=1 TO 3
40   PRINT "ENTER QUANTITY SOLD OF ITEM"; I
50   INPUT G(I)
55 NEXT I: CLS
60 PRINT "ITEM 1=MILK, ITEM 2=EGGS, ITEM 3=WAX"
70 INPUT "WHICH ITEM DO YOU WANT TO RE-ORDER"; I
80 PRINT "ORDER THIS MUCH"; G(I)
90 PRINT: GOTO 60
```

When you see G(1), G(2), etc., read "G-sub-one, G-sub-two . . ." The "sub" stands for "subscript."

As Ray explained it, he had replaced the three "simple variables" EGGS, MILK and WAX with a single "array variable" G( ) which was capable of holding all three values at once.

We'll be building on Ray's (not Bob's) programs, so it would be a good idea to save this one (and each following one) on tape for convenience.

Ray went on, "The ( ) holds an index that tells which value you want: G(1) holds your re-order quantity for milk; G(2), eggs; and G(3), floor wax."

Then he drew a picture of G( ):

G(1)	MILK SALES
G(2)	EGG SALES
G(3)	WAX SALES

"But what was wrong with my simple variables EGGS, MILK and WAX?" Bob wanted to know. "They worked just fine."

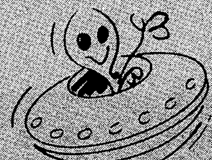
"Right now they *seem* okay, but what about when you add items to your store? If you had 50 items, you'd have to type in 50 separate variables.

"See that little DIM statement in line 10? That tells TRS-80 to create an array G( ) with room for three items: G(1), G(2), and G(3). But the beauty of G( ) is, it's expandable! Simply by changing the DIM statement you could make the array as large as you like. For example, changing line 10 to:

```
10 DIM G(20)
```

would give G( ) the capability to store 20 items!" It was Ray's first chance to talk programming in several months. He beamed with satisfaction.

Being a sensitive man, Bob couldn't bear to tell Ray that he never was going to add a single item to his line. So he nodded approvingly and offered Ray a job as a grocery sacker. And just for Ray's sake, he started using the array G( ).



UFO #15-1

In addition to **simple variables,** BASIC offers **array variables**, or arrays, for short. Arrays let you store related information in an organized manner so that it can be easily accessed.

The DIM ("dimension") statement creates an array. The simplest kind of array has one dimension; to create such an array, use a DIM statement like this:

DIM *variable (size)*

where *variable* is the name of the array (just like any other variable name), and *size* tells what the largest index or "subscript" will be. *Size* can be a numeric variable, constant, or expression.

Once an array has been created with DIM, you can access any item in it by putting the appropriate subscript inside the ( ).

A one-dimensional array is like a numbered **list**; the subscript specifies or points to a particular entry in the list.

To refer to an entire one-dimensional array, not to one particular element, we use this notation:

array-name ( )

This indicates that one subscript is used. You can't use this notation in a program — it's strictly for the book.

Program (Do It Yourself #15-1). Suppose Bob *did* want to carry 20 different items. How would you change his original simple-variable program to handle it? (Don't do it — just *think* about it.)

Now modify Ray's array program to handle 20 items. (Don't make up names for each item category — just call them "Item 1," "Item 2," etc.)

When you're done, compare your program with ours in Appendix A.

Note: The D.I.Y.'s in this chapter are *very* important, so be sure to try them. And be sure to check our answers in Appendix A. Once you get a program running, save it on tape so you won't have to re-type the whole thing when we want to add to it later.

The End of the Honeymoon

There was only one problem with Bob's computerized stock re-ordering system: the store was about to go bankrupt due to items being out of stock all the time. Seems that the demand for eggs, milk and floor wax changed from month to month.

Bob mentioned the problem to Ray at 5:30 one day, just as the weary sacker was preparing to go home.

Ray stared out the storefront window for a moment, remembering his glory days as a programmer. In those days, he used to solve problems like this in his sleep. . .

"Well, Bob, suppose you keep track of how much was sold in each month in the preceding year, and make your monthly orders based upon the sales for the same month in the preceding year?"

"Hmmm. . .", said Bob, "I'll have to think on that one for a while."

Ray wished him luck and wandered home.

Bob began working with Ray's idea right away. "Let's see," he said, "a single month uses three spaces in the array, so 12 months will require 36 spaces. I'm going to have to dimension G( ) for 36 entries.

"Hey! That would make old Ray happy—he always did want to expand that array. Let's see, now. It's going to look like this. . ."

And Bob drew a picture that looked like this:

JANUARY

G(1)	MILK SALES
G(2)	EGG SALES
G(3)	WAX SALES

FEBRUARY

G(4)	MILK SALES
G(5)	EGG SALES
G(6)	WAX SALES

MARCH

G(7)	MILK SALES

Bob struggled with the problem of specifying a particular item for a given month. "Hmmm. What do all the milk quantities have in common? If I want milk for a given month, what subscript do I use?" After several hours, he had a breakthrough.

Breakthrough (Do It Yourself #15-2). What subscript points to milk for a given month *m*? (Try various values for *m*, from 1 to 12. Then make a general answer.)

Milk quantity for month *m* = G (   )

Here's Bob's breakthrough:

"Okay, to specify the milk quantity for any given month, you multiply the month number times 3, then subtract 2. For example, December's milk quantity is stored in G(12 * 3 − 2) = G(34).

"The eggs quantity for any given month is stored in. . ."

When Ray came in the next morning, he found Bob asleep, his head pillowed by the TRS-80 keyboard. The area around his desk was littered with wads of crumpled paper.

Program (Do It Yourself #15-3). Follow through with Bob's idea, storing an entire year's sales record in G( ).

Hint: The eggs quantity for a given month M is stored in:

```
G(M*3 - 1)
```

Where is the wax quantity for a given month M stored?

Check your answer against ours in Appendix A.

The First Quantum Leap

After he listed Bob's still-incomplete program, Ray muttered something about there being no need to store two-dimensional data in a one-dimensional data structure, and went off to modify his first array program. Here's what he came up with:

```
10 DIM G(12,3)
15 FOR M=1 TO 12
18   PRINT "DURING MONTH #"; M
20   FOR I=1 TO 3
30     PRINT "HOW MUCH OF ITEM"; I; "WAS SOLD";
40     G(M,I)=RND(100): PRINT G(M,I); "(TEST VALUE)"
43     FOR T=1 TO 800: NEXT T
45 NEXT I,M
50 CLS
55 INPUT "WHAT MONTH IS IT"; M
60 INPUT "PRESS <1> FOR MILK, <2> FOR EGGS, <3> FOR WAX"; A
70 PRINT "AMOUNT TO ORDER IS"; G(M,A)
80 GOTO 55
```

Special Note about line 40: Notice the new "word", RND. This stands for "random" (unpredictable). RND(100) returns a whole number from 1 to 100. RND(50) returns a whole number from 1 to 50. Etc. RND(0) returns a fraction between 0 and 1.

We're using RND to generate sample data, so you don't have to type in 36 quantities.

When Bob woke up, Ray showed him the program. Bob ran it, and was quite pleased with the results. He was especially happy that the program produced its own sample data, so he wouldn't have to type in numbers (Let some one else do that!).

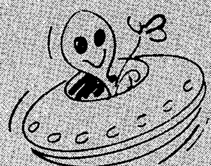
Now he wanted to understand the program. "I still don't see how you got it to store all those numbers without increasing G( ) with DIM G(36). Nor do I see how you got it to locate each item in the array without multiplying the month times the item number and subtracting and all that nonsense."

Ray hadn't had so much attention since his last day at the computer center, the day he accidentally erased the master file index . . .

"Well, Bob, I did expand G( ) — but in a different direction, or, to be precise, in a different *dimension*. You see, you tried to expand G( ) from a short list into a longer one, using the statement DIM G(36). What I did was change the array into a table, containing 12 columns and three rows, using the statement DIM G(12,3). If you consider how many entries there are in a 12 x 3 table, you'll see that I have indeed expanded the capacity of the original array.

By the way, how many entries are there in a 12 x 3 table?

"In programmer-talk, I've gone from a one-dimensional array G( ) to a two-dimensional array G(,)."



UFO #15-2.
An array can have more than one dimension. For each added dimension of an array, you use an additional subscript inside the ( ), with a comma after every subscript but the last.

To create a two-dimensional array, use a statement like:

DIM *variable (size1, size2)*

where *variable* is any variable name; *size1* tells what the largest subscript will be for the first dimension; and *size2* tells what the largest subscript will be for the second dimension.

A two-dimension array is like a *table*; one subscript represents the column; the other, the row. Together, the two subscripts pin-point a particular entry in the table.

To refer to an entire two-dimensional array, and not to any particular element, we use this notation:
array-name (,)
This indicates that two subscripts are used: one before and one after the comma. You can't use this notation in a program, however — it's strictly for the book.

Ray drew a picture of the array G(,):

MONTH

ITEM	1	2	3	4	5	6	7	8	9	10	11	12
1												
2												
3												

"In your single-column array, it took only one subscript to specify an item. In my 12-column array, it takes two subscripts to specify an item — a column number and a row number."

"Okay, I get the idea of what you did — but why? How did you decide to use two dimensions instead of one?"

The Flight of the Blazing Programmer (after the quantum leap)

By this time Ray was beside himself with pride and confidence. He began to have visions of himself as a systems analyst with people waiting in line to have their systems analyzed. "I just looked at the data, realized there were now two organizing principles, and concluded that two dimensions were in order.

"Originally, our sales information was organized by item number alone. There was only one way of looking at the data. But now, we've got two perspectives on the data. . . item number *and* month. You might say that we've gone from a one-dimensional data base to a two-dimensional one. For this reason, it's appropriate to switch from a one-dimensional array to a two-dimensional one.

"You've already mentioned one advantage of this array — that we don't have to multiply the index to get the item for any particular month. For example, to get the sales of milk in November, we just look at G(11,1), which is column 11, row 1 in our table. In your extended list, we'd have to use G(I*3-2).

"But there's another more important advantage of G(,) over G( ). Think about what we're storing — sales totals. Now how do we want these sales figures organized? By month or by item number? Since we've got the data in a two-dimensional array, we can easily look at it from either perspective.

"Here, let me give you a sample of the possibilities." And Ray added these lines to the program on page 167:

```
51 PRINT "PRESS <1> FOR RE-ORDER, <2> FOR SALES ANALYSIS"
53 INPUT A
54 ON A GOTO 55, 100
80 GOTO 51
100 PRINT "SALES ANALYSIS . . . "
110 H=1: L=1: GT=0
115 HT=G(1,1)+G(1,2)+G(1,3): LT=HT
120 FOR M=2 TO 12
130   MT=0
140   FOR I=1 TO 3: MT=MT+G(M,I): NEXT I
150   IF MT>HT THEN H=M: HT=MT
160   IF MT<LT THEN L=M: LT=MT
170   GT=GT+MT
180 NEXT M
190 PRINT "THE GRAND TOTAL OF ITEMS SOLD IS"; GT
200 PRINT "MONTH"; H; "WAS BEST, WITH TOTAL SALES OF"; HT
210 PRINT "MONTH"; L; "WAS WORST, WITH TOTAL SALES OF"; LT
220 GOTO 51
```

Program Analysis (Do It Yourself #15-4). RUN the program and try to figure out how the sales analysis works.

Hints: H stores the subscript of the high-sales month. The program starts out by assuming that month 1 was best (line 110: H=1).

HT stores the total sales for high-sales month H. The program starts out with the total equal to the total sales for month 1 (see line 115). HT is compared with each monthly total MT for months 2 through 12 (lines 120, 130, 140, and 150). When MT > HT, M becomes the new high-sales month, and MT becomes the new high total.

What purpose do L, LT and GT serve?

After running the program, use *immediate* commands to find out:

1. What item sold best in the best month?
2. What item sold worst in the worst month?

Program (Do It Yourself #15-5). Add to the sales analysis portion of the program so you can compute and print total annual sales for each item.

Hint: In the last program, we computed monthly totals. In your program, you'll need item totals. First total up

G(1,1) + G(2,1) + . . . + G(12,1)

That sum will be total annual sales for item 1, milk. Also do the sums for I=2 and I=3 (I stands for the second subscript, the item).

```
220 T1=0: T2=0: T3=0: REM SET TOTALS=0
______________________________________
______________________________________
______________________________________
______________________________________
______________________________________
______________________________________
______________________________________
______________________________________
______________________________________
310 GOTO 51
```

☑ Chapter Checkpoint #15

1. **Arrays:**
   a. are named after Bob's sacker/programmer
   b. let you store related information
   c. are created by the DIM statement
   d. all of the above
2. See if you can draw a rough sketch of a simple two-dimensional array.

Chapter

Sixteen

The Founding of an Empire

You'll need some of the programs developed in the previous chapter. Be sure to do all the D.I.Y.'s in this chapter, and to check your work against the programs in Appendix A.

After running Ray's sales analysis program, Bob Babson recognized that he was wasting Ray's talents by using him as a grocery sacker. So he promoted him to cashier, and gave him permission to fool around with the Computer during his lunch break.

With his powerful new sales analysis program, Bob's business flourished. His inventory of milk, eggs, and floor wax was always right for the demand. Profits were higher than ever.

In fact, Bob decided to open a chain of stores under the name, Bob's "Simply Super" Stores #1, #2, . . . #*n*. He liked the sound of the name. "Alliteration," he explained to Ray.

The only problem Bob now faced was how to raise enough capital to purchase one TRS-80 for each outlet. He considered instituting a temporary employee wage cut to free up the cash. . .

But once again, Ray had a better idea.

"Bob, you still haven't grasped the full potential of arrays. Now think about it for a minute. Up to now, you have been storing data in terms of two organizing principles: item number and month number. That's why we have a two-dimensional array.

"Now you're adding more sales outlets. Instead of putting a Computer in each store, why not simply add an additional organizing principle to the data — namely, the store number. Then every piece of data will be pin-pointed by *three* indices: item number, month number *and store number*. Since we're now talking about a three-dimensional data base, we'll need a *three-dimensional array*.

"How many stores will you start with?"

Well, I was thinking of four, counting this one. That would be one for each side of town." said Bob.

"Here, watch this. . ." And Ray typed in a new program, which was actually quite similar to his first array program.

```
10 DIM G(4,12,3)
12 FOR S=1 TO 4
13 PRINT "SIMPLY SUPER STORE NUMBER"; S
15 FOR M=1 TO 12
18 PRINT "DURING MONTH #"; M
20 FOR I=1 TO 3
30 PRINT "HOW MUCH OF ITEM"; I; "WAS SOLD";
40 G(S,M,I)=RND(100): PRINT G(S,M,I); "(SIMULATED)"
42 FOR T=1 TO 800: NEXT T
45 NEXT I, M, S
50 CLS
54 INPUT "WHAT STORE (1,2,3, OR 4)"; S
55 INPUT "WHAT MONTH IS IT"; M
60 INPUT "TYPE 1 FOR MILK, 2 FOR EGGS, 3 FOR WAX"; A
70 PRINT "AMOUNT TO ORDER IS"; G(S,M,A)
80 GOTO 53
```

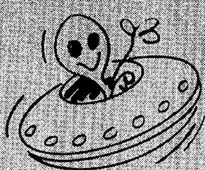
Line 40: Once again, we're using RND to take the place of keyboard entry. If we didn't, you'd have to type in 3 * 12 * 4 = 144 numbers!

Program Analysis (Do It Yourself #16-1). By now you can figure out what's happening in this program.

1. After running the program, type in the following immediate line:
   PRINT G(4, 12, 3)
   What does the number printed represent (what store, month, and item)?
2. How many items can G(, ,) store?
3. The old array G( ) was like a list; G(,) was like a table with rows and columns. How would you visualize the three-dimensional array G(, ,)?

Program (Do It Yourself #16-2). Start with the sales analysis program from the last chapter and make the necessary changes so it handles all four stores, one at a time. You have to change every reference to the array, and add a FOR/NEXT loop.

Compare your answer with ours in Appendix A.



UFO #16-1.
An array can have any number of dimensions, as long as sufficient memory is available. A generalized DIM statement looks like this:

DIM variable (*size1, size2,. . .*)

When referring to an array outside of a program, use this notation:

variable ( )	One dimension
variable (,)	Two dimensions
variable (, ,)	Three dimensions

and so forth, showing how many commas would be needed, but leaving out the actual subscripts.

A three-dimensional array is like a stack of pages, each page containing a table. The first subscript specifies the page number; the second, the row; the third, the column. Together, the three subscripts pin-point a particular entry.

This notation is for use in writing *about* programs.

In actual programs, you must include all the subscripts.

The Simply Super Mart chain was truly an idea whose time had come. All four stores prospered, under the guidance of the Simply Super Sales Analysis Program, of course.

As a result, Bob Babson was able to retire from day-to-day responsibilities in the store. For the first time in his life, he could devote full time to his poetry.

And Ray? Well, Bob made him manager of one of the stores, and put him in charge of computer applications.

This was the opportunity Ray had been waiting for. Little did Bob Babson realize it, but the Simply Super Mart Chain was about to venture into the uncharted land of. . . computerized advertising!

Move Over, Madison Avenue!

Refer to UFO #15-1. You'll notice it says *any* variable name can be used for the array. Does that include string variables?

Exploration (Do It Yourself #16-3). Create an array that will store string values instead of numbers. Store a list of 10 names in the array, then print them out in the same order as they were input.

Now you're ready for Ray's program. . . almost. First we've got to formally introduce a statement we've used once or twice: CLEAR.

We've told you that CLEAR erases the contents of variables. Well, it can also be used to set aside more (or less) space in the Computer for storing your string variables. Read the note in the margin for details.

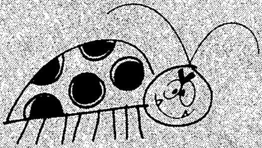
Finished with that? Whew! Now for the program. . .

```
10 CLEAR 100
15 DATA 5,5,4
20 DATA MEN,WOMEN,KIDS,PETS,SHOPPERS
30 DATA LOVE,NEED,ADORE,DEMAND,APPRECIATE
40 DATA CONVENIENCE,LOW PRICES,FRIENDLY ATMOSPHERE,QUALITY
45 READ N,V,F
50 DIM NN$(N),VB$(V),FT$(F)
60 FOR I=1 TO N: READ NN$(I): NEXT I
70 FOR I=1 TO V: READ VB$(I): NEXT I
80 FOR I=1 TO F: READ FT$(I): NEXT I
90 CLS
100 C=RND(5): PRINT NN$(C);"--EVEN";
110 C=RND(5): PRINT NN$(C);"--";
120 C=RND(5): PRINT VB$(C)
130 C=RND(4): PRINT "BOB's SIMPLY SUPER";FT$(C);"!"
140 FOR I=1 TO 1000: NEXT I
150 PRINT
160 GOTO 100
```

RUN the program. But don't let it run too long, or you might start planning a trip to Bob's Simply Super Mart!

Program Analysis (Do It Yourself #16-4). In the *immediate* mode, type in a FOR/NEXT loop that will print out

- All the nouns, NN$( )
- All the verbs, VB$( )
- All the "features", FT$( )



You see, the Computer assumes you will need only enough string space to store 50 characters. Unless you change this with the CLEAR statement, you'll get an Out of String space error (?OS ERROR) when you try to store more than 50 characters. This error may even occur because your program asked the Computer to create temporary storage that was not available in the string space.

To change the string space allowance, use CLEAR with a numeric expression for the maximum number of characters to be stored in string variables and temporary storage.

For example,

```
CLEAR 100
```

allows for 100 characters of string storage.

The string storage is set to 50 each time you start TRS-80 (power-on). If you change it with CLEAR *n*, the new value of *n* will be used until you change it or re-start (power-off/on). CLEAR without a numeric expression leaves the string space unchanged.

You may have noticed cases where the ad slogan repeated a noun, as in:

```
MEN - EVEN MEN - DEMAND
BOB'S SIMPLY SUPER QUALITY!
```

It wouldn't be hard to prevent this from happening. The program just needs to remember which word in NN$( ) was chosen in line 100, and then make sure this one isn't chosen again in line 120. If it is, then repeat line 120 until another word is chosen. For example, type in these lines:

```
100 D = RND (5): PRINT NN$ (D); "--EVEN";
110 C = RND (5): IF C=D THEN 110 ELSE PRINT NN$ (C); "--";
```

Poetic Justice?

The Computerized advertising campaign proved to be a mixed blessing.

First of all, Ray added a printer to his Computer system, and used it to print out reams of advertising flyers. Customer demand increased dramatically; the inventory program had to be modified so that the reorder quantity was always three times last year's monthly order. Bob's suppliers couldn't keep up with his demand, so he had to look for other suppliers. All four stores had to be expanded and remodeled to handle the increased customer load — more counters, wider aisles, piped-in music, etc.

All of which left Bob with less and less time for his poetry. . .

But the ad-slogan program had given Bob an idea. In the past few years, he had come to recognize his own poetry for what it was — doggerel. Well, if Bob was too busy to write doggerel, then maybe he could program TRS-80 to do it for him!

Program (Do It Yourself #16-5). Write a poetry (or doggerel) program.

Decide on your own format, or use this suggested approach:

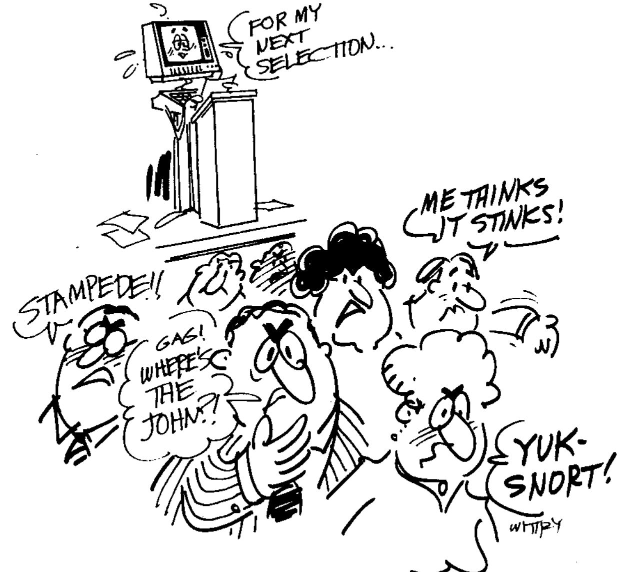
1. Read 10 nouns from DATA statements
2. Read 5 adjectives from DATA statements
3. Read 5 adverbs from DATA statements
4. Read 5 intransitive verbs ("be-verbs") from DATA statements
5. Print out randomly selected words in the following pattern:
   (noun) in the (adjective) (noun)
   (noun) (adverb) (adjective)
   when will the (noun) (verb)?

Here's what a sample poem might look like:

```
BULLS IN THE RED BIRDNEST
WINTER SLOWLY FALLING
WHEN WILL THE TREE AWAKEN?
```

The nature of the poems will be determined by the words you store in the DATA statements.

For our answer, see Appendix A.



Exploration (Do It Yourself #16-6). All right, you've seen string arrays. Now what other kinds of arrays might there be? Hmmm. What other kinds of *numbers* are there? Remember #, %, and !? (You may want to look back at the programs in Chapter 9 for a refresher on numbers.)

1. Create an integer array, and demonstrate that it stores integer values only.
2. Create a double-precision array, and demonstrate that it stores double-precision values.
3. Create a single-precision array, and demonstrate that it stores single-precision values. (Yes, we know, G( ) and all the other grocery arrays were single-precision. We just want you to put them in perspective.)

Two Short Subjects

In all our array programs, we've used 1 as the lowest subscript for the array. This is a natural approach. That way, 1 refers to the first element in the array. Furthermore, we said that the statement:

```
DIM G(20)
```

creates an array that can store 20 items, G(1), G(2),. . ., G(20)

Well, your TRS-80 BASIC arrays are special; each dimension has one extra subscript available, called the zero subscript. It works just like the other subscripts. For example, type in these *immediate* lines:

```
CLEAR
DIM G(1)
G(0) = 1
G(1) = 10
PRINT G(0), G(1)
```

Not one, but *two* elements in the array G( )!

For two-dimensional arrays, both dimensions have zero-subscripts. Type in these **immediate** lines:

```
CLEAR
DIM G(1,1)
G(Ø,Ø)=1
G(Ø,1)=1Ø
G(1,Ø)=1ØØ
G(1,1)=1ØØØ
PRINT G(Ø,Ø), G(Ø,1), G(1,Ø), G(1,1)
```

UFO #16-2.
For every dimension of an array, the lowest subscript allowable is 0. Therefore, after a statement like:

```
DIM G(4, 12, 3)
```

the array G(, ,) actually has 5 * 13 * 4 elements.

Why the zero subscripts? Well, in some applications, it's convenient to be able to use them. At the end of the next chapter, we'll give you an example. For now, you can use them. . . or ignore them.

TRS-80 has another special array feature. Used properly, it's a convenience. But it *can* get you into trouble. Try this (*immediate* mode):

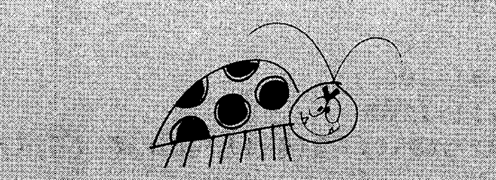
```
CLEAR
A(5) = 1ØØ
PRINT A(5)
```

Now how did that work? You never DIMensioned the array A( )! Here's what happened. CLEAR erases all arrays and other variables. When you used A(5), BASIC automatically created an array A( ). How big is A( )? Try these *immediate* lines:

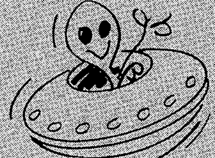
```
A(6)=90
A(7)=100
A(8)=110
A(9)=120
A(10)=130
A(11)=140
```

The last line will cause this to happen:

```
?BS ERROR
```



Remember, that's what happens when you use a subscript out of range. When TRS-80 created the array G( ), it allowed for only 11 subscripts, 0, 1, 2, . . ., 10.



UFO #16-3.
If you use an array reference without first DIMensioning the array, your Computer will automatically create the array. Such "automatic arrays" always have 11 subscripts in each dimension, 0, 1, 2, . . ., 10.

For example, if you use a statement like this:

```
A(5,4,8) = 100
```

without a prior DIM statement, TRS-80 will create the array A(, ,); in effect the Computer automatically performs this statement:

```
DIM A(10,10,10)
```

before giving A(5,4,8) the value 100. Note that you cannot use a subscript greater than 10 in an automatically created array (since each dimension in such an array has a maximum subscript of 10).

Why use automatic array creation? Well, it's simply a convenience.

If your array will need no more than 11 subscripts per dimension, you may choose to skip the DIM statement and let the Computer create the array. But unless you need exactly 11 subscripts per dimension, you'll be wasting memory; the array will be larger than you need. We suggest you always dimension your arrays to the size you need, using the DIM statement.

Arrays and Memory Use (Optional Information).

Multi-dimension arrays can use up your Computer's memory quickly. Of course, the type of array is a factor here, as well.

Integer Arrays

Each array element takes two memory locations, or "bytes." For example, after:

We'll have more to say about "bytes" in Chapter 27. For now, just think "memory storage location" when you read "byte."

```
100 DIM NN%(2,5)
```

array NN%(,) contains 3 * 6 = 18 integer elements; 18 integer elements take up 18 * 2 = 36 bytes of memory.

Single-Precision Arrays

Each array element takes four bytes of memory. For example, after:

```
100 DIM SS!(2,5)
```

array SS!(,) contains 3 * 16 = 18 single-precision elements; 18 single-precision elements take up 18 * 4 = 72 bytes of memory.

Double-Precision Arrays

Each array element takes eight bytes of memory. For example, after:

```
100 DIM SS#(2,5)
```

array SS#(,) contains 3 * 16 = 18 double-precision elements; 18 double-precision elements take up 18 * 8 = 144 bytes of memory.

String Arrays

The string space (what is CLEARed) is used to store the string text itself. However, each string array element requires three bytes of memory "overhead." For example, after:

```
100 DIM TX$(2,5)
```

array TX$(,) contains 3 * 16 = 18 string elements; 18 string elements take up 18 * 3 = 54 bytes of memory. The actual length of each string must be added to this value to get the total memory usage of TX$(,).

General Formula for Computing Memory Requirements of Arrays

The array $G(N_1, N_2, \ldots, N_k)$ requires the following amount of memory:

14 + (k*2) + T * {(N1+1) * (N2+1) * . . . * (Nk+1)}

where k is the number of dimensions in the array, and the value of T depends on the array type:

Type	T=
Integer	2
Single-Precision	4
Double-Precision	8
String	3

In computing the actual memory requirements of **string** arrays, you must add the text length of each element in the array. When the array is first dimensioned, all elements have length 0. The string text will be stored in the string space (reserved by the CLEAR n statement).

☑ Chapter Checkpoint #16

1. See if you can correct this statement to make it meaningful to TRS-80:
```
DIM 1(G,2,6)
```
2. Can you remember what limits the dimensions of an array? (**Hint:** If you forgot maybe TRS-80 can jog your memory.)

Chapter

Seventeen

The Menace from Meklovakia

At great risk, we were able to place our top covert operative, Agent 7E-3, into the totalitarian state of Meklovakia; his mission was to foment insurrection by whatever means possible.

E3, as our man is called for short, was a former programmer whose sneakers and unobtrusive manner made him perfect for the mission. Being logical — a necessity for any programmer — our intrepid operative decided to subvert the computerized Meklovakian Censor.

All writings that are to be published or broadcast in Meklovakia, you see, must first be fed into the Automatic Censor. E3 provided us with a copy of the Censor program. We're half-ashamed to admit it, but it *will* run on your TRS-80!

"Text" — in Computerese — refers to string rather than numeric information. (Remember strings from Chapter 4? They are sequences of characters, including letters, numbers, punctuation, graphics, and even special control codes.)

We'll need to lay some groundwork before looking at the program. In particular, we're going to look at several new BASIC statements and functions. Since we're talking about text rather than numbers, all of them will involve strings. Along the way, we'll be giving a brief refresher course in Meklovakian history.

First you take a string. . .

Let's start by storing a string in a variable. Type in this program:

```
90 CLEAR 50
100 PRINT "TYPE IN A LINE AND PRESS <ENTER>"
120 INPUT P$
130 PRINT P$; " IS STORED IN THE COMPUTER"
140 PRINT
150 GOTO 90
```

What's line 90 for?

Remember the string space? That's where the Computer stores all the string variables and temporary string quantities used by your program. In the last chapter, we used CLEAR 100 to reserve 100 characters for string variables.

We want to set the string space to its initial size of 50 characters. You can do this by turning the Computer off and on again, but it's much easier to do it in the program with line 90.

RUN the program, and try entering these lines:

```
THIS MESSAGE CONTAINS 35 CHARACTERS (ENTER)
>>>>>>>>>>>>-----------<<<<<<<<<<<< (ENTER)
GIVE ME LIBERTY, OR GIVE ME DEATH! (ENTER)
```

Hey — what's this EXTRA IGNORED business? Is the Automatic Censor already in place in the Computer? Must be stored in ROM. . .

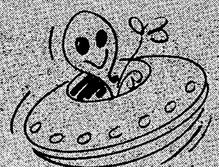
Well, actually, there *is* a kind of censor built-in to the Computer. But it's harmless and quite purposeful. You see, when you type in values to an INPUT statement, TRS-80 looks for certain punctuation symbols to tell it where the text begins and ends.

In particular, **commas** and **colons** serve as markers. So when you typed in GIVE ME LIBERTY, OR GIVE ME DEATH, the Computer assumed that the intended text ended at the first comma. Since it found more text after the comma, it warned you that some information had to be ignored.

There's another very useful marker, the double-quote. If your INPUT text starts with a double-quote, your Computer will accept everything you type into one string — until it comes to another double-quote. This lets you get commas and colons into the string. The quotes do not become part of the string; they are simply markers.

Now try typing in the last string again, but this time start and end with a double-quote.

The ending quote is really not required, unless you are going to type in more than one string value to a single INPUT statement.



UFO #17-1.
During string INPUT TRS-80 stops when it reaches a comma or colon. To make your computer accept these symbols as part of the string, enclose the string inside double-quotes.

When strings are used in name-it statements like

```
A$ = "MY FAVORITE AUTHORS"
```

they must always be placed inside double-quotes.

Provocateurs, Infiltrators, and Other Long Words

With surprising ease, our fearless infiltrator obtained a position in the Censorship bureau and, after a few nervous weeks, decided to test the Automatic Censor for the first time (the Man vs. Machine theme again).

After entering the string input program into the computer, he quietly typed in the following (and you can do likewise):

```
RUN (ENTER)
"THE MEMORY OF MEKLOVAKIA'S GREATEST HERO, SCRUBDEK, LIVES ON
(ENTER)
```

No sirens went off, no bells clanged. The only noise heard, in fact, was a sigh of relief from Numero 7 as TRS-80 answered with:

```
?OS ERROR
```

No sweat, he thought, and the diminutive programmer simply made the following changes in line 90:

```
90 CLEAR 300
```

Now *you* can safely RUN the program, since it was our man in the field who took all the risks. Now for the long-awaited UFO on CLEAR. . .

UFO #17-2.

TRS-80 stores string information in a special area of memory called the string space. The size of this space is initially set to 50 characters. That is, the Computer can store up to 50 characters of string information. You can change this size — increase it or decrease it — with the CLEAR statement:

```
CLEAR n
```

where *n* is a number or a numeric expression telling how much string space you want reserved.

CLEAR also clears out the current values of all variables. For this reason, it is almost always used early in a program. To clear out all variables without changing the current size of the string space, simply use:

```
CLEAR
```

by itself, without a number or numeric expression.

Stored where? In ROM? That must be somewhere just southeast of Sedalia. No. . . Actually, ROM is short for "Read-Only Memory" which is, in effect, the "pre-programmed" stuff your TRS-80 has that you can't change (such as the BASIC language).

How much string space should you clear? First figure out how many string variables you are going to need, and what their maximum lengths will be. Add these up, and you've got a good first estimate. But this still may not be enough if you're doing a lot of string processing (described later in this chapter), because TRS-80 will also use the string space for its internal string calculations.

If you clear too much space, BASIC won't have enough memory left to do anything, and you'll have to reset the Computer. For now, let's leave string space at 300.

So now we can store a long string — even an inflammatory one. What's next?

Before doing anything to the string, we need a way to "look at it" while a program is RUNning. For example, how long is the string? Well, we can always print it, and count the length, but this is rather slow and painful. As you've heard many times already, TRS-80 has a better way. . .

New! Improved! String Measuring Function!

- ***No more tiresome, error-prone character-counting!***
- ***Returns the exact length of any string — instantly!***
- ***Easy to use — just type LEN( ) with your string inside the ( )!***
- ***Hundreds of uses — around the house or office — gets you out of just about any stringy situation!***

Make sure you still have the string input program in memory (if you don't, then type it in again):

```
90 CLEAR 300
100 PRINT "TYPE IN A LINE AND PRESS <ENTER>"
120 INPUT P$
130 PRINT P$; " IS STORED IN THE COMPUTER"
140 PRINT
150 GOTO 90
```

Now add one line to the program then RUN it:

```
125 PRINT "THE LENGTH OF THE STRING IS";LEN(P$)
```

UFO #17-3.

The function LEN returns the length of any string. Its general form is:

LEN(*string*)

where *string* is a string variable or a string constant enclosed in quotes. *string* can even be a string expression (described later).

Since LEN is a function, it cannot be used alone. It must be used in a complete BASIC statement (for example, a PRINT or name-it statement).

Program (Do It Yourself #17-1). Anticipating the horrors of the Automatic Censor, alter the string input program so that it refuses to accept *any* four-letter word you type in, and stores some other word instead.

(Oh — you're partial to words with four letters? Then make your program reject all 10-letter words.)

After you've tried — and succeeded, we hope — be sure to look at our answer in Appendix A.

This program isn't much of a threat to free expression — yet. You can fool it by typing in a couple of words — even if both are, tish, tish, four letters long. For example,

```
DROP DEAD
```

looks perfectly acceptable to the program, even though either word by itself would be rejected. As far as the program is concerned, DROP DEAD is a string nine characters long, so it's okay.

We have ways to deal with those would deceive the censor. . .

What's needed is a way to examine the parts of a string. Well, you can always look at it on the display — but that invites human error and judgment. You might be tempted to accept certain four-letter words simply because they are neither obscene, uncouth, inflammatory, nor abusive. That won't do at all since, by design, the Automatic Censor must be autocratic and arbitrary to a fault.

The following fundamental axiom suggests an approach to the problem:

Fundamental Axiom #17-1

All strings have three parts:

1) A left portion
2) A mid portion
3) A right portion

Aristotle would have liked this Computer!

TRS-80 has a set of functions designed in accordance with this axiom. In the *immediate* mode, type in these lines:

```
P$="LONG LIVE SCRUBDEK"
PRINT LEFT$(P$, 4)
PRINT LEFT$(P$, 9)
PRINT RIGHT$(P$, 4)
PRINT RIGHT$(P$, 8)
```

Are you with us so far? Good, now type this and watch closely:

```
PRINT MID$(P$, 5, 4)
PRINT MID$(P$, 11, 5)
```

UFO #17-4.
TRS-80 has three built-in functions that examine the parts of a string: LEFT$, MID$, and RIGHT$ (pronounced "left-string", "mid-string", and "right-string"). LEFT$ gets the left portion of the string; MID$, the mid portion; RIGHT$, the right portion.

LEFT$ and RIGHT$ have the following general form:
  LEFT$(*string*, *length*)
  RIGHT$(*string*, *length*)

where *string* is a string variable, constant or expression (defined later), and *length* tells how many characters you want to look at.

MID$ has the following general form:
  MID$(*string*, *position*, *length*)
where *string* and *length* are defined above, and *position* tells where the mid-portion starts. It can be any number from 1 (the first character in the string) up to the length of the string.

The three functions cannot stand alone; they must be used in complete BASIC statements (like PRINT or name-it statements).

Now this all may be new to you but not to 7E (actually 7E-3 but remember, he's working undercover now). He finally began to realize that the Meklovakian code wasn't that difficult, and openly began to challenge the system in this manner (after all, that's what he's getting paid to do).

Type along with him in the *immediate* mode:

```
P$="SCRUBDEK LIVES"
PRINT MID$(P$, 1, 8)
P$="WHO KILLED SCRUBDEK?"
PRINT MID$(P$, 12, 8)
P$ = "THE CROWDS WERE SHOUTING,  SCRUBDEK,  SCRUBDEK,  HE'S OUR
MAN"
PRINT MID$(P$, 37, 8)
```

7E-3 is exploring the inner workings of the Censor program we're leading up to. . .

By now, you should be ready for. . .

Fundamental Axiom #17-2.

It's easy to find a string if you know exactly where to look.
The trick is to know where. . .

By chance, Agent 7 stumbled across a classified Meklovakian Computer operation manual that gave a detailed account of how the Automatic Censor ferreted out those unauthorized words. This, of course, made E3's job infinitely easier and he passed on the information verbatim:

Suppose you suspected that P$ contained SCRUBDEK, but you didn't know where. You could use MID$ to search through the string, as follows:

1. Get the length of P$ — LEN(P$) — so you'll know how many eight-character sequences you need to check. Suppose you can't display the following string, but your program can "look" at it with LEN and MID$. P$ contains this value:

   ```
   "LONG LIVE SCRUBDEK"
   ```

2. Get the first eight characters of P$ — MID$(P$, 1, 8) — and compare with "SCRUBDEK".

   ```
   [LONG LIV]E SCRUBDEK
   ```

3. Starting with the second character of P$, get another eight-character portion — MID$(P$, 2, 8) — for comparison.

```
L[ONG LIVE]SCRUBDEK
```

4. Continue until you have checked the entire string. The last portion you check should be MID$(P$, LEN(P$) −7, 8)

```
LONG LIVE[SCRUBDEK]
```

Program (Do It Yourself #17-2). As you may have gleaned from the earlier examples, SCRUBDEK was a Meklovakian patriot who had to be eliminated. His name has become the rallying cry of the Meklovakian Underground. Understandably, the current rulers of that nation would like to erase the name from the national consciousness. Your mission is to anticipate what kind of program they might use to accomplish this goal.

No, you're not aiding the enemy — how can you fight 'em if you don't understand their modus operandi (M.O. to you Dragnet fans)?

Write a program that INPUTs a line of text, and then searches through the text for the word SCRUBDEK. If it finds the word, it prints a warning about what happens to trouble-makers in Meklovakia.

Hint: MID$ and LEN are the only string functions you need. Use MID$ like a moving window to check eight-character portions of the text, starting at position one.

Give it a good try. When you get a working program, be sure it can handle all kinds of situations — SCRUBDEK at the very beginning of the text, very end, inside another word, etc.

When you're done, be sure to look at our answer in the Appendix.

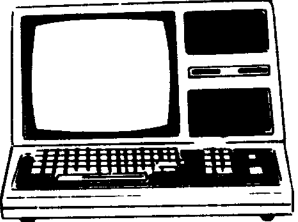
☑ Chapter Checkpoint #17

1. Anytime you use a string within a name-it statement, you must enclose the string inside:

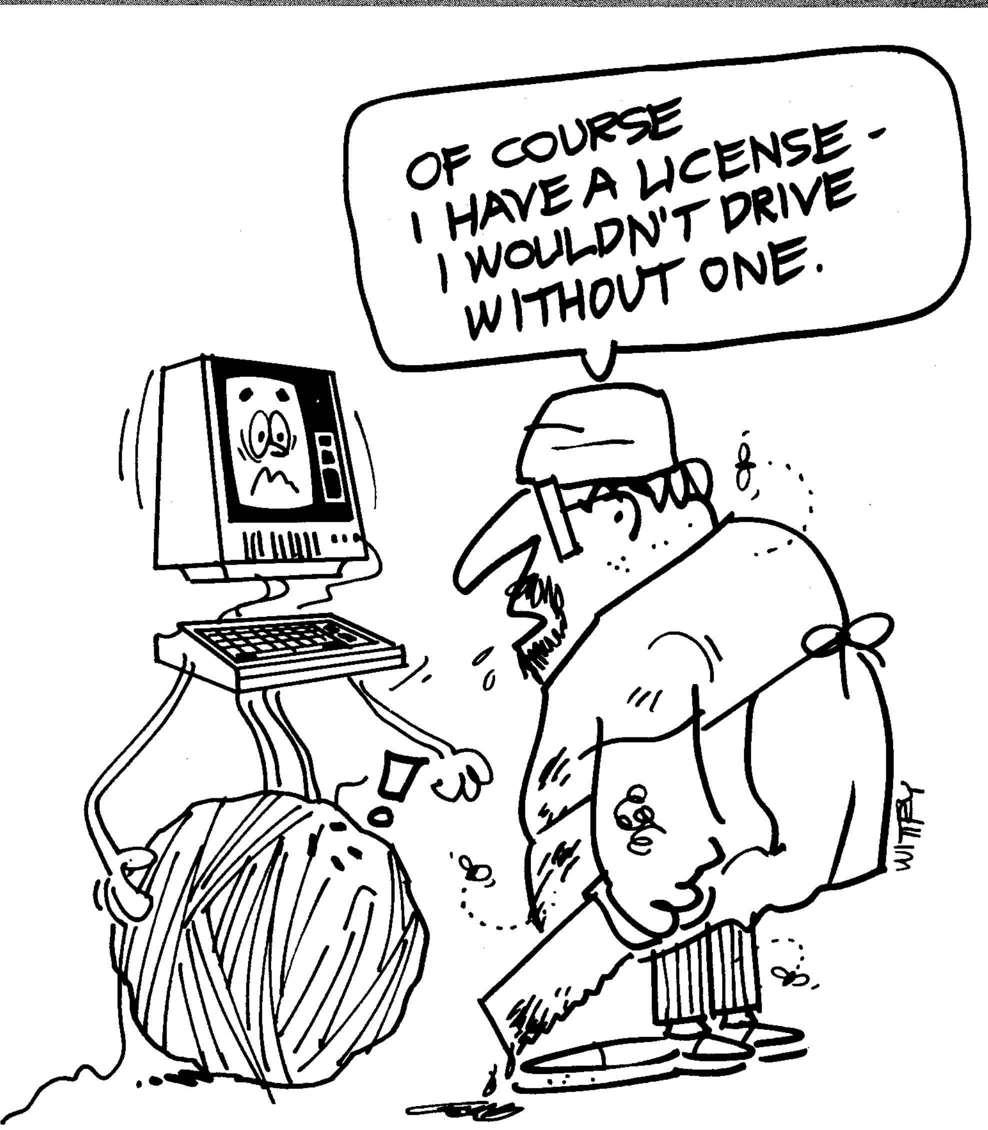
   a. double quotes
   b. daisy clover
   c. numeric expressions

2. What's wrong with this statement?
   "All strings have three parts — a left portion, a right portion, and a child-size portion."

3. In order to expand the string space from its normal ________ characters, you must type ________ and then the number of the characters desired.



Chapter



Eighteen

Our Man in Meklovakia, Continued

Now we've solved the problem of searching through text for a particular string. But two problems remain.

1. How about those four-letter words? After our last accomplishment, this may seem like gilding the lily, but to any right-thinking Meklovakian crat, it's absolutely necessary that we have this capability. A truly effective censor can't be required to know in advance which words are to be censored.

2. Once we've found the offensive words, what do we do with them? We can count them, but little else. Are we in the business of revolution — or simply statistics? Of course, we can be sure the Automatic Censor will do much more than *count* offenses. Its goal is to *eliminate* them.

We've already got the necessary tools to solve problem #1. The trick is to know how to use them (Corollary to Fundamental Axiom #17-2). But problem #2 is really urgent, and requires some new tools, so we'll tackle it first. Meanwhile, you can be thinking about how to find any four-letter word in any string.

What's a crat? That's a generic term for autocrats, bureaucrats, technocrats, hypocrats (that's right, *hypocrats*!).

Here are a few pointers to get you started on problem 1:

First, you've got to decide what constitutes a word, in practical terms. Another way of putting it — what delimits (marks the beginning and ending of) a word?

Once you've pinpointed a word by finding its beginning and end, how do you find out whether it has four letters or not?

String Surgery in Your Spare Time!

- ***No license required!***
- ***Fun — Challenging — Educational!***
- ***Train yourself for a career in Meklovakia!***

First of all, let's get a little perspective on strings. Strings are one of BASIC's four categories of data. The other three are all numeric:

Integers
Single-Precision Numbers
Double-Precision Numbers

You know about the many operations that can be performed on numeric data, but what about strings? At present, all you can do is store them, print them out, and examine them. Given a string, you cannot modify it. Nor can you take two different strings and operate on the two of them together. It's a wonder you've gotten this far with such limitations!

Well, what kinds of operations would you like to perform? On numbers, you've got +, −, *, / and ↑. Which of these might be useful with strings — and how would they work? Hmmm.

Back to the laboratory, Igor, revisited.

Find out which, if any, of the math operators can be used with string values. You might try lines like this:

```
PRINT "LONG LIVE SCRUBDEK"-"SCRUBDEK"
```

If you find an operation that works, make sure you understand what it is doing. Do some experimenting on your own before going on. . .

If you've done some experimenting, you won't be surprised when you type in these immediate lines:

```
A$="HAVE MORE FUN"
B$="MONKEYS"
C$=B$+A$
PRINT C$
```

To get this:

```
MONKEYSHAVE MORE FUN
```

Looks promising, but where'd the space go between MONKEYS and HAVE? Look closely, and you'll see *we didn't put one there*. Try this line:

```
C$=B$+" "+A$
```

There's a single space between the quotes.

UFO #18-1.

The plus symbol + can be used to join two strings. This is called "concatenation," which means "linking" or "stringing together." The general form for concatenation is:

string1 + *string2*

where *string1* and *string2* are string variables, constants, or expressions (described below). The result of such an operation is a string consisting of *string1* followed by *string2*.

string 1 + *string 2* is a **string expression:** it cannot stand alone, but must be used in a statement (like a PRINT or name-it statement). A string expression can also be used inside the parentheses of a string function.

Practical String Surgery (for Graduates of "String Surgery in Your Spare Time")

Now let's get back to the neglected, but not forgotten, hero. He felt that he was just getting close to cracking the Automatic Censor code wide open when his supervisor, an anonymous-looking crat named NOBODY, began watching him more closely than usual.

Such anonymous-sounding names are common among crats.

At first, 7E thought it might just have been the cup of cocoa he spilled on NOBODY's new shirt when he tripped over the extension cord that he had run across the floor to power the Computer. He finally decided, however, that he better do something to get back into NOBODY's good graces again or he might be breaking rocks instead of codes.

He suddenly found his chance one day when he was given some lines to censor: In the *immediate* mode, type

```
P$ = "LONG LIVE SCRUBDEK"
```

According to the supervisor, the obvious offender was SCRUBDEK, which started at position 11. To put it more positively, the first ten letters of P$ were acceptable.

Agent 7E eliminated the problem (and gained himself a promotion) this way: (Type in as well.)

```
P$ = LEFT$(P$, 10) + "NOBODY"
PRINT P$
```

That would turn any crat's head!

Program (Do It Yourself #18-1). With your new skills, add to your DIYP #17-2, so that every time SCRUBDEK is located, it is replaced with the string NOBODY.

After you're done, check our answer in the Appendix.

Meet the automatic censor . . .

Type in the following program and RUN it. It's rather long, so you may need to do some debugging of typing errors before it'll perform.

```
NEW
10 CLEAR 1000
15 DEFINT A-Z
25 DIM K(256)      'K() STORES DELIMITER POSITIONS
30 CLS
40 PRINT "MEKLOVAKIAN OFFICIAL AUTOMATIC CENSOR."
50 PRINT "TYPE IN A QUOTATION MARK, THEN THE TEXT."
60 INPUT P$: S1$=P$
120 S2$=" ": SP=1: KT=0: K(0)=0
125 GOSUB 1000
130 IF SF=0 THEN 200
135 KT=KT+1: K(KT)=SF 'KT=# OF BLANKS, K()=POS'N
```

In line 120, there's a single space between the quotes.

```
140 SP=SF+S4: IF SP>S3 THEN 200 'END OF TEXT
145 GOTO 125
200 K(KT+1)=S3+1      'END OF TEXT MARKER
210 FOR I=1 TO KT+1
220   IF K(I)-K(I-1) <> 5 THEN 290
230   S2$="****": SC=4: SF=K(I-1)+1
240   GOSUB 2000
290 NEXT I
300 IF S3<8 THEN 400
305 SP=1
310 S2$="SCRUBDEK": GOSUB 1000
320 IF SF=0 THEN 400
330 S2$="NOBODY": SC=8: GOSUB 2000
340 IF SR<S4 THEN 400
350 SP=SF+1: GOTO 310
400 CLS: PRINT "AUTHORIZED TEXT FOLLOWS . . .": PRINT S1$
420 END
1000 ' INSTRING SUBROUTINE
1080 SF=0: S3=LEN(S1$): S4=LEN(S2$)
1090 IF S4>S3-SP+1 THEN PRINT "PARAMETER ERROR": RETURN
1100 SL=S3-S4+1
1110 FOR SI=SP TO SL
1120   IF MID$(S1$,SI,S4)=S2$ THEN SF=SI: SI=SL
1130 NEXT SI
1140 RETURN
2000 ' MIDSTRING REPLACEMENT SUBROUTINE
2070 S3=LEN(S1$): SR=S3-SF-SC+1
2080 IF SF<1 OR SR<0 THEN RETURN
2090 S1$=LEFT$(S1$,SF-1)+S2$+RIGHT$(S1$,SR)
2100 RETURN
```

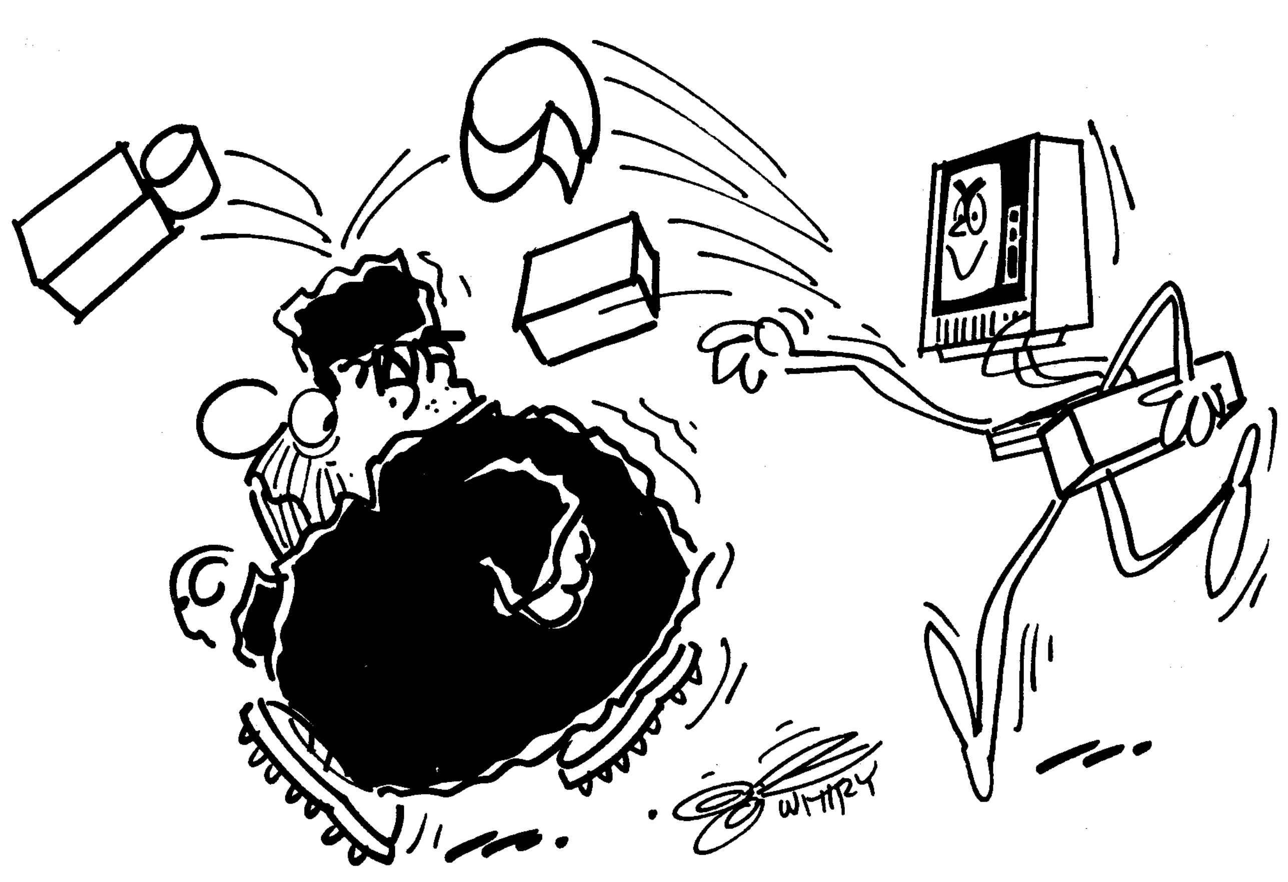
Lines 330, 2070: Note the changes in the midstring replacement subroutine. SC is the number of characters to be *deleted* from S1$. S2$ is *inserted*. So we can change the net length of S1$.

After you get it working be sure to save it on tape. We'll use it in the next chapter.

☑ Chapter Checkpoint #18

1. In order to link two strings, you need to use the ____________ operator.
2. Just for review, can you list the four categories of DATA?
   1. ______________________________
   2. ______________________________
   3. ______________________________
   4. ______________________________

Chapter



Nineteen

Assault on the Automatic Censor

Never underestimate human ingenuity. Agent E3 finally broke through. Much to the consternation of some perplexed Meklovakian crats, the newspapers and radio reports were full of unauthorized words. The population was in an uproar.

What Agent 4 (a.k.a. 7-3) found out was that by placing any kind of punctuation before or after a four-letter word, he could get that word past the censor.

RUN the program again, and try a line like this:

```
"LONG-LIVE-SCRUBDEK--DOWN-WITH-CENSORSHIP!"
```

Use hyphens where a space would normally appear.

Furthermore, our man even found a way to get the taboo name, SCRUBDEK, past the censor!

You see, whoever programmed the Automatic Censor made a small but devastating oversight. He was so used to seeing:

SCRUBDEK

scrawled on walls and buildings throughout the city that he never stopped to think about the following possibilities:

scrubdek
Scrubdek
ScrubdeK
SCRUbDEK
sCrUbDeK

all of which are, to a crat, just as odious as SCRUBDEK.

For Model I TRS-80 owners only: Your TRS-80 may only display capital letters, but that doesn't mean you can't store lowercase letters in a string! You do it by holding down (SHIFT) while you press any alphabet key. The lowercase letter will be input to the string, even though when the string is displayed, it will look like a capital letter!

RUN the Censor program from the last chapter and try typing in this line:

```
"I-LIKE-scrubdeK"
```

Model I users without lowercase: press (SHIFT) while you type "SCRUBDEK."

Escalation in the War against Words

The Meklovakian officials weren't about to give up. They located a promising young student named Hank from the Meklovakian hamlet of Broadaxe, and sent this willing lad to the vaunted halls of the Institute for Forensic String Surgery.

The place was settled by refugees from the American colonies who were anti-freedom of the press.

After a full two years of research and experimentation on live strings, Hank came back with the listing for a new program tucked in his knapsack. Agent 7, who had passed the two years in uncensored bliss, held his breath. The press murmured nervously, and broadcasters mouthed their stories mindlessly. All dreaded the installation of the new Automatic Censor.

Add the following lines to your Automatic Censor (some replace existing lines):

See Appendix B, sample program #5, for the complete program listing.

```
20 DIM K$ (255)      'K$() STORES DELIMITER IMAGE
45 PRINT "BROADAXE VERSION--YOU TYPE IT IN--I CUT IT UP"
70 FOR I=1 TO LEN(S1$)        'CHANGE LOWERCASE TO UPPERCASE
75 S2$ = MID$ (S1$, I, 1)
80 IF S2$ < CHR$(97) OR S2$ > CHR$(122) THEN 95
85 S2$ = CHR$( ASC(S2$) - 32)
90 SF = I: SC = 1: GOSUB 2000
95 NEXT I
100 FOR I=1 TO 255: K$(I) = "": NEXT I      'CLEAR OUT K$()
110 DATA " ", ",", ";", ".", "!", "?", "-", "0"
115 READ S2$: IF S2$ = "0" THEN 190      'CHECK ALL DELIMS.
120 SP = 1
130 IF SF = 0 THEN 115 'GET NEXT DELIMITER
135 K$(SF) = S2$
140 SP = SF + S4: IF SP > S3 THEN 115 'END OF TEXT
145 GOTO 125
190 KT = 0: K(0) = 0    'KT COUNTS BLANKS, K(0) IS ALWAYS ZERO
192 FOR I=1 TO 255
194 IF K$(I) <> "" THEN KT = KT + 1: K(KT) = I
196 NEXT I
```

Line 110: The first DATA item is a single blank space inside quotes.

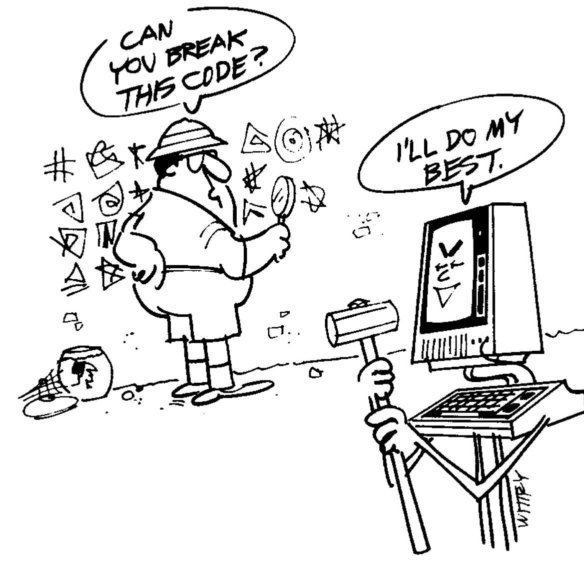
Line 194: K$(I) is compared with the null string. There are **no spaces** between the double quotes: "".

Now RUN it. Try to get contraband text past the censor—use the tricks that worked with the old censor.

When you run it, be prepared for a wait of up to a minute or so — crats *never* move fast, you know!

Program Analysis (Do It Yourself #19-1). Study lines 100 to 196 and figure out what's happening. (Programmer Hank was good enough to leave a few explanatory remarks in the listing.)

1. What is the purpose of all those DATA items? Try taking out a few of them and see if the program suffers for it. (Don't take out the "0"; it's just an end-of-list indicator.)
2. The censor can still be fooled. (Did you try *all* the punctuation and other special symbols as "camouflage"?) Make a few simple additions to the DATA list to make the program even more rigorous.
3. What does the array K$(   ) store? You might want to put a STOP statement right after line 145, and then print out the entire array.



The ASCII Code and How to Break It

What about lines 70-95? The remark in line 70 says "CHANGE LOWERCASE TO UPPERCASE." And line 80 uses the < and > operators with what appear to be string operands. What's it all about?

If you haven't yet CSAVEd the Broadaxe version of the Automatic Censor, do it now. Then erase it from memory, and type in this little one:

```
10 CLS
20 FOR I = 32 TO 127
30 PRINT "CODE"; I, "ASCII CHARACTER="; CHR$(I)
40 IF I-31 <> INT ( (I-31) / 10 ) * 10 THEN 60
50 INPUT "PRESS <ENTER> TO CONTINUE"; X: CLS
60 NEXT I
```

Now RUN it. You'll see a table of codes and ASCII (pronounced ASK-ee") characters. This method of assigning a number to each character allows the Computer to represent text in terms of numbers. In fact that's the *only* way it can store text.

RUN the program again and notice that nothing is displayed for code 32. Actually, a blank space is printed, since code 32 represents a blank in the ASCII code. Prove it? Type in this immediate line:

```
PRINT ">"; CHR$(32); "<"
```

the space in > < was put there by CHR$(32).

By the way, ASCII stands for American Standard Code for Information Interchange. "AAW. . ." (Short for Aren't Acronyms Wonderful.)

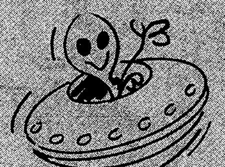
Of course, you can ignore this fact most of the time, and proceed as if "LONG LIVE SCRUBDEK" and other strings were really tucked away in the Computer somewhere . . .

TRS-80 has two functions, ASC and CHR$, for going back and forth between codes and ASCII characters. In the program you just entered, the CHR$ function takes a code and returns the ASCII character associated with it.

To see ASC in action, type in this program:

```
NEW
10 INPUT "TYPE IN A LETTER OR NUMBER"; C$
20 PRINT "THE ASCII CODE FOR THAT CHARACTER IS"; ASC(C$)
30 PRINT
40 GOTO 10
```

RUN the program, and try typing in various characters. Be sure to compare the results of typing unshifted and shifted letters, for example, (A) and (SHIFT) (A).



UFO #19-1.
TRS-80 has two functions for conversions between ASCII codes and their associated characters.

ASC(*string*)
returns the ASCII code for the first character of *string*. *string* can be a string variable, constant, or expression. It must have a length greater than zero (it can't be the null string).

CHR$(*code*)
returns the character associated with *code*. *code* must have a value between 0 and 255 inclusive, and can be a numeric variable, constant or expression.

For Model I TRS-80 Owners Only:
Did you notice that the alphabet characters were displayed twice, once with codes 65 to 90, and again with codes 97 to 122? The second set of letters is the lowercase alphabet, but as we mentioned previously, unmodified Model I TRS-80's show only uppercase (capital) letters. If you had the Radio Shack lowercase modification installed, or connected an upper- and lowercase printer to your TRS-80 and substituted LPRINT for PRINT in line 30, you'd see the lowercase alphabet for codes 97-122.

Now you've probably got some ideas on what's going on in lines 70-95 of the Broadaxe Automatic Censor.

When faced with the problem of catching words like:
scrubdek   Scrubdek   ScrubdeK   SCRUbDEK   sCrUbDeK

Hank decided against checking for every possible combination of upper- and lower case characters that might spell the taboo word. That would have been a brute-force, plodding approach, and at the Institute for Forensic String Surgery, such methods are rarely used.

Instead, he chose a more direct (and more brutal) solution: change every lowercase character to uppercase.

Of course, such a change would have a tremendous impact on the literary world (what would happen to e. e. cummings' poems?). But perhaps that was an appropriate punishment for those who had used lowercase input to circumvent the original Automatic Censor. . .

Changing lowercase to capitals is quite simple. If you look at the ASCII table in Appendix B, you'll see that every lowercase character has a code that's equal to its uppercase counterpart's code plus 32. For example, the code for "A" is 65; for "a", 65 + 32 = 97.

Now all that's necessary is to find all characters from "a" to "z", and change them to the codes for the proper character in the "A" to "Z" set.

Line 75 looks at each character in the input string, one at a time. Line 80 checks to see whether the character under scrutiny (S2$) is between ("a") 97 and ("z") 122. If it is, line 85 sets it equal to its uppercase counterpart. Line 90 then calls the midstring replacement subroutine to put in the new uppercase character.

Line 80 deserves a little more attention:

```
IF S2$ < CHR$(97) OR S2$ > CHR$(122) THEN 95
```

The line reads, "If S2$ is less than ASCII character 97 or S2$ is greater than ASCII character 122 then go to line 95."

How can one string be "less than" or "greater than" another? RUN this program and you'll soon get the idea. . .

```
NEW
10 INPUT "TYPE IN ANY WORD"; A$
20 INPUT "TYPE IN A DIFFERENT WORD"; B$
30 IF A$ = B$ THEN 20
40 IF A$ < B$ THEN F$ = A$: S$ = B$: GOTO 60
50 F$ = B$: S$ = A$
60 PRINT F$; " PRECEDES "; S$; " ALPHABETICALLY"
70 PRINT: GOTO 10
```

You don't have to type in words—any sequence of characters is okay. (Remember to start with a " if you want to include commas or colons in a string.)

Here are some suggestions on strings you might type in:

"AAA WRECKING SERVICE" and "aaa WRECKING SERVICE"
"100" and "2"
"SCRUBDEK" and "SCRUBDEK THE PATRIOT"
"A" AND "a"

UFO #19-2.
Strings can be compared with the following operators:

Operator	Meaning
<	Precedes alphabetically
>	Follows alphabetically
=	Is the same as
<>	Is not the same as
<=	Precedes or is the same as
=>	Follows or is the same as

Comparison is made character-for-character on the basis of each character's ASCII code; a character with a lower code is precedent over a character with a higher code.

During string comparison, when a non-matching character is found, the string containing the lower-coded character is considered to precede the other string.

If no non-matching characters are found, the shorter string is precedent.

Program Analysis (Do It Yourself #19-2.) The following program inputs 15 words, places them in alphabetical order, then prints out the alphabetized list.

```
100 CLEAR 300
110 DIM A$(15)
120 FOR I=1 TO 15
130 PRINT "TYPE IN WORD #"; I;
140 INPUT A$(I)
150 NEXT I
160 T=0     'WHEN T=1, A SWAP HAS OCCURRED
170 FOR I=1 TO 14
180 IF A$(I) < A$(I+1) THEN 200
185 'THE NEXT LINE SWAPS THE TWO STRING ELEMENTS
190 X$=A$(I): A$(I)=A$(I+1): A$(I+1)=X$
195 T=1     'SET T SINCE A SWAP OCCURRED
200 NEXT I
210 IF T=1 THEN 160    'REPEAT UNTIL NO SWAPS ARE MADE
215 CLS
220 PRINT "HERE IS THE ALPHABETIZED LIST",
230 FOR I=1 TO 15
235 PRINT
240 PRINT A$(I),
250 NEXT I
260 GOTO 260
```

Type in the program, run it, figure out what's happening. Notice the use of T as a "flag" to indicate when a "swap" has occurred. For further comments, see Appendix A.

Program (Do It Yourself #19-3.) No more tinkering with censorship! Put some of your new knowledge to good use by writing a "copy editor" program that:

1. Inputs text.
2. Breaks it up into its component sentences and words.
3. Prints out:
   a) The number of sentences.
   b) The length of each sentence.
   c) The average word length (in letters).
   d) The average sentence length (in words).
4. Advises the user to use longer/shorter words/sentences. (You decide what's optimum for each!)

Most of the logic you'll need is contained in the Broadaxe version of the Automatic Censor.

This program will probably be rather slow in giving you your "critique."

Give it a good try. If you can't get it working, don't be too discouraged. There's always a position waiting for you in Meklovakia!

Be sure to look at our answer in Appendix A.

Program (Do It Yourself #19-4). Write a program that:

1. Inputs and stores 10 text strings, like this:

   ```
   TYPE IN YOUR TEXT
   ```

   After you type in the text and press (ENTER), the program prints a message like this:

   ```
   THAT WAS ENTRY #1. YOU HAVE 9 ENTRIES LEFT.
   ```

2. Prints out the strings one at a time, asking the user if any changes are needed. Like this:

   ```
   HERE IS ENTRY #1. DO YOU WANT TO CHANGE IT? (Y/N)
   ```

   If you type Y (ENTER), the program will ask you:

   ```
   TYPE IN THE POSITION WHERE THE CHANGE STARTS
   ```

   Then:

   ```
   TYPE IN THE NUMBER OF CHARACTERS TO CHANGE
   ```

   Then:

   ```
   TYPE IN THE NEW TEXT
   ```

3. Makes the specified changes.
4. Prints out the entire corrected text.

Trying this one will teach you a lot. Don't forget to look at our answer in Appendix A when you're done.

Epilogue: Who Won the War Against Words?

Agent 7E realized he had finally met his match, but, on the other hand, Hank had come to the same frustrating conclusion.

For months, they had stood side-by-side, unknowingly battling each other but becoming great pals in the meantime. Finally E3 decided on one final, bold stroke that would hopefully solve the problem — he approached his friend and offered to take him to the New World. Surprisingly, Hank agreed!

Oh no! He hadn't changed his political beliefs. Instead, he opted for something far more important — a life that offered ball-point pens, Levi's, and Big Mac's.

☑ Chapter Checkpoint #19

1. ASCII (pronounced ASK-ee) is a:
   a. request in Missouri-ese.
   b. code for representing text in terms of numbers
   c. way to delete characters

2. Match each function with its output:

a. CHR$	1. code
b. ASC	2. string

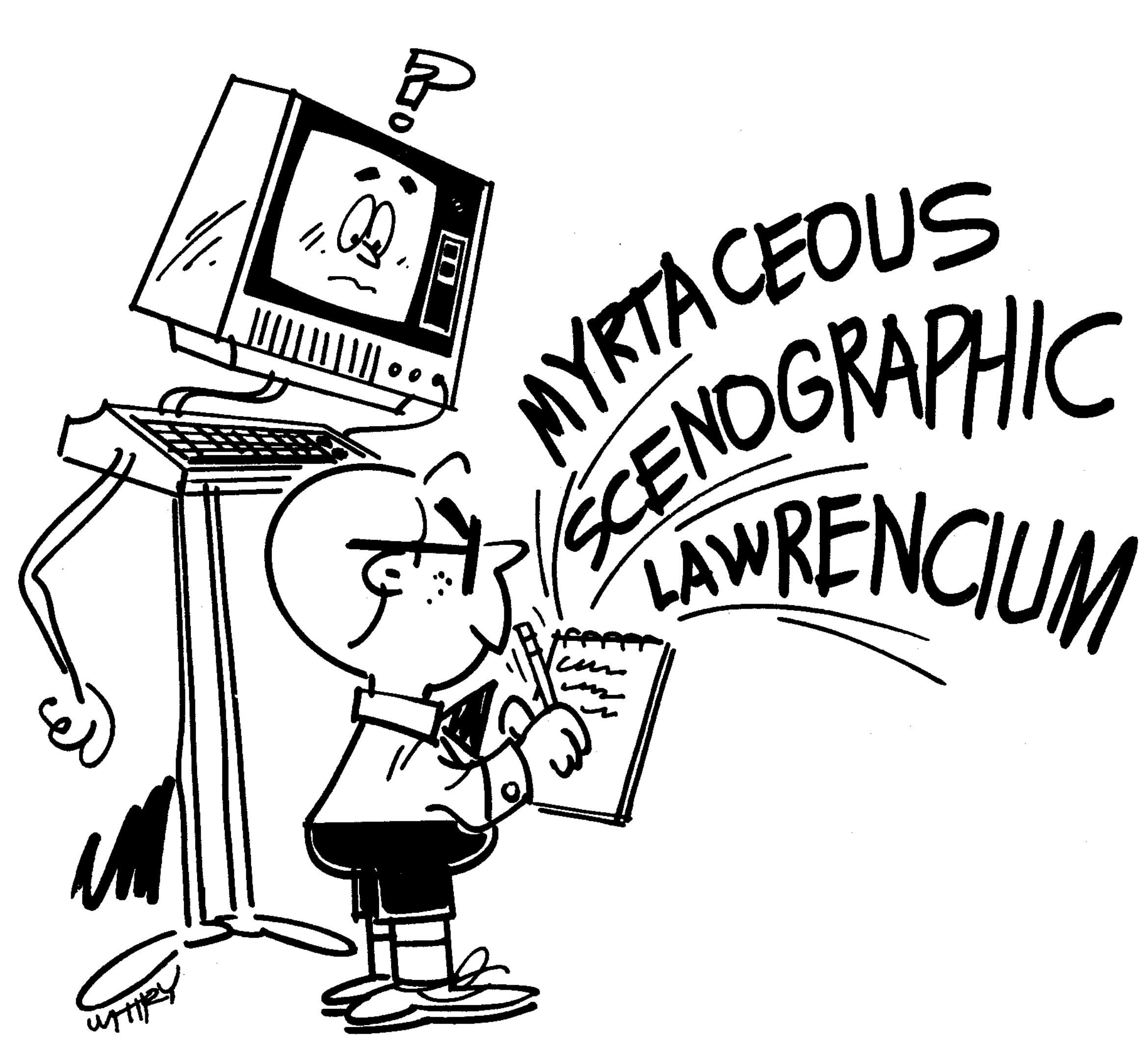
3. If the code for "M" (uppercase) is 77 then the code for "m" (lowercase) is ________.

4. One practical application of ASCII coding system is that it allows TRS-80 to put lists of words in ________ order.

Part III
Exploring the Territory

- Advanced PRINT
- Graphics
- Special Keyboard Techniques
- Cassette and Printer Operations
- Memory Addressing
- Error-Handling

Chapter



Twenty

Advanced Print

By now, you've probably come to take PRINTing for granted: you tell TRS-80 what you want printed, and it takes care of where to print it, when to start a new line, what to do if it reaches the bottom of the display, etc.

In this chapter, we're going to explore ways to take a more active part in how TRS-80 outputs to the display. You might say we're going to turn off the automatic pilot and fly manually. As you'll see, we can do much more this way.

But first, we'll recap what we've learned already.

Recap (Do It Yourself #20-1). Answer these questions. RUNning the programs will help you come up with the answers.

A. How many character positions (columns) are there to a line on the screen? RUN this program to number each column:

```
10 CLS
20 PRINT
   "0         1         2         3         4         5
         6"
30 PRINT "01234567890123456789012345678901234567890
   1234567890123456789 0123"
40 PRINT "IS LINE THREE BLANK? WHY?"
50 GOTO 50
```

B. Add a trailing semi-colon to line 30 and RUN the program again. Now there is no blank line. . . What happens each time TRS-80 prints a character in the rightmost column of the display?

Note: In line 20, the digits in quotes are spaced ten columns apart. (There are nine spaces between "0" and "1," "1" and "2," etc.)

In line 30, there are 64 digits inside the quotes. When lines 10 and 20 are printed, they should give column numbers when read vertically:

```
Col. 0     Col. 10    Col. 20    etc.
0          1          2
012345678901234567890
```

C. How many PRINT lines are there on the display? RUN this program to number them:

```
10 CLS
20 FOR I = 1 TO 15
30 PRINT I 'NUMBER FIRST 15 LINES
40 NEXT I
50 PRINT I; 'BOTTOM LINE IS A SPECIAL CASE (PREVENT SCROLL)
60 GOTO 60 'SAVE DISPLAY
```

Why is the bottom line a special case?

D. How many characters (including blank spaces) does the Computer actually output to the display when it prints a number? RUN this program:

```
10 CLS
30 PRINT "0....5....0....5....0....5....0....5"
35 READ A: IF A=0 THEN END
40 PRINT A;
50 GOTO 35
60 DATA 1, 10, 5.3, -1, -10, -5.3, -5, -5.30000, 000.1, 0
```

In general, what can you say about BASIC's procedure for printing out numbers?

E. How about strings? How many characters (including blank spaces) does BASIC actually output to the display when it prints each of these string values: "HELLO"; "TWO WORDS"; "1"; "-1"? (Hint: use print lists with semi-colons again.)

F. What's the maximum number of digits BASIC will print out for a single-precision number? a double-precision number? What happens to leading zeros (zeros to the left of the first significant digit)? What happens to trailing zeros (zeros to the right of the least significant fractional digit)?

G. How can you predict where printing will start? Try out this program:

```
10 CLS
20 PRINT "AFTER CLS, PRINTING STARTS HERE": PRINT
30 PRINT "THIS PRINT STATEMENT ENDS WITHOUT PUNCTUATION"
40 PRINT "PRINTING RESUMES HERE": PRINT
50 PRINT "THIS PRINT STATEMENT ENDS WITH A ';'";
60 PRINT "PRINTING RESUMES HERE": PRINT
70 PRINT "THIS PRINT STATEMENT ENDS WITH A ','",
80 PRINT "PRINTING RESUMES HERE": PRINT
```

Question E: Modify the preceding program to handle strings instead of numbers.

Question F: Try these immediate lines:

```
PRINT 1.2345678!
PRINT 1.23456789012345678 9#
PRINT 000001, 1.00000
```

H. Ever notice that the cursor disappears when you type RUN? Or that it reappears during an INPUT? How can you predict where the cursor is going to reappear? Try this:

```
10 CLS
20 PRINT "THE CURSOR IS INVISIBLE NOW"
30 FOR T=1 TO 990:NEXT T
40 INPUT "IT RETURNS FOR AN INPUT (PRESS <ENTER>)"; X
50 GOTO 20
```

Be sure to compare your observations and answers with ours in Appendix A.

"Advanced printing" refers to controlling where things are printed and how they are printed. Perhaps the last D.I.Y. gave you some ideas about why you might want to control these things. Let's explore one application that might call for some advanced printing.

Printing Tables of Numbers

Suppose we have a bunch of numbers to display. The numbers might represent anything from dollars to inches of rainfall. Our goal is to display the numbers in a neat table.

For convenience, we'll generate random sample numbers with RND (random), and we'll try several ways of outputting them in columns. The first and easiest way is PRINT with a trailing comma.

RUN this program:

```
10 CLS
20 PRINT "COLUMN A", "COLUMN B", "COLUMN C", "COLUMN D"
30 FOR I = 1 TO 56
40 X = RND(99) + RND(0) 'GENERATE SAMPLE NUMBER
50 PRINT X,
60 NEXT I
70 GOTO 70 'SAVE DISPLAY
```

Line 40 generates random numbers between 1 and 100, as follows: RND(99) returns an integer from 1 to 99, and RND(0) returns a fraction between 0 and 1. The sum of these two random numbers will be between 1 and 100. The trailing comma in line 50 "automatically" creates four columns of figures.

Your display will look something like this (of course, your numbers will most likely be different):

```
COLUMN A        COLUMN B        COLUMN C        COLUMN D
77.0575         13.0247         57.1814         47.1631
4.6652          15.3687         86.5021         93.5092
43.0401         39.2367         95.0283         48.9092
55.4142         94.2629         92.8904         39.8335
85.1913         17.1285         47.1405         94.5491
17.4471         13.0616         40.2529         90.6199
16.4292         14.337          15.6235         99.4069
84.0569         26.7765         22.0702         46.5677
15.4216         75.2488         75.3328         50.5569
31.2865         64.1694         1.5728          81.126
44.5865         18.0006         82.8738         8.82325
1.9288          35.7518         18.7411         46.494
36.092          36.4791         64.602          10.7802
17.5269         18.7856         88.3055         64.621
```

Neat, isn't it? But suppose we want more columns than that? Or suppose we want to position them differently. This is where the comma bows out to make way for a much more versatile print feature. . . .

Program (Do It Yourself #20-2). Try the program with a trailing semi-colon instead of a trailing comma in line 50. How does that look? Not too good? How about printing some blank spaces after each number (like this: "   ").

The TAB Function

To see the TAB function in action, type in these statements (immediate mode):

```
PRINT TAB(10) "COLUMN ONE"; TAB(40); "COLUMN TWO"
PRINT "NAME"; TAB(20); "AGE"; TAB(30); "PHONE NUMBER"
PRINT TAB(5); "5"; TAB(25); "25"; TAB(50); "50"
PRINT TAB(5); 5 ; TAB(25); 25; TAB(50); 50
```

After working through the recap (DIY 20-1), you can probably understand why the last two PRINT statements produce slightly different results.

Remember, when the string "5" is printed, it takes up exactly one space; but when the number 5 is printed it takes up three spaces (a leading blank followed by the digit, followed by a trailing space. And so on with the other numbers.

UFO #20-1

TAB is a PRINT function for positioning the cursor to a column position so that the next item printed will start there. The general format for TAB is:

TAB(*n*)

where *n* is a numeric expression with a value from zero to 255. The columns on the video display are numbered from zero through 63. If *n* exceeds 63, the cursor "wraps around" to the lower lines. For example, TAB(64) positions the cursor at column 0 of the next line; TAB(128+4) positions the cursor to column 4 two lines below the current cursor position.

If the cursor is already past the position specified by *n*, the TAB will be ignored.

Note to Model I Users: TRS-80 Model I will wrap around on the same line.

Now let's use TAB to revise the 0-to-100 number generator. Instead of four columns, we'll use five. The columns will start at positions 0, 10, 20, 30, 40 and we'll place headings over the center of each column:

```
10 CLS
20 PRINT TAB(5);"A";TAB(15);"B";TAB(25);"C";TAB(35);
  "D";TAB(45);"E"
30 FOR I=1 TO 15 'DO 15 LINES
40 FOR J=0 TO 40 STEP 10
50 X=RND(99)+RND(0) 'GENERATE SAMPLE NUMBER
60 PRINT TAB(J);X; 'NOTE TRAILING SEMI-COLON
70 NEXT J
80 IF I=15 THEN 100 'NO NEW LINE IF AT BOTTOM OF SCREEN
90 PRINT 'FORCE A NEW LINE
100 NEXT I
110 GOTO 110
```

Save this program on tape. We'll be referring to it and modifying it in the next chapter. Call it the "average rainfall" program.

Notice the TAB value doesn't have to be a numeric constant — it can be any numeric expression. This means a single TAB(*n*) function in your PRINT statement can generate any number of TAB positions.

This produces a table of 5 * 15 = 75 numbers:

A	B	C	D	E
40.0325	16.0714	39.5522	18.5358	70.7737
83.5779	41.1408	8.27347	40.2796	40.4357
80.2956	71.787	29.821	95.8808	98.8109
48.7733	27.0908	12.4457	22.653	5.9759
86.7038	45.4602	14.4601	80.6854	67.6966
5.55063	23.0423	38.2392	94.0277	94.1404
33.8815	13.5215	1.07752	68.9497	16.6315
63.4636	81.1482	80.4956	77.618	24.5434
42.964	29.5921	11.3386	49.2858	49.1708
15.85	44.9627	70.7627	81.8365	84.177
92.8315	39.0926	77.9508	85.2778	5.95834
82.9412	85.9255	11.1537	18.5282	51.0207
17.0939	54.6032	5.39804	68.5905	92.2114
56.7215	23.5967	79.9568	95.9554	57.4005
66.8248	6.97246	20.4248	46.8294	16.9549

Note: we can control the column position during video output. But examine the table closely. Notice that all of the numbers are "flush left", that is, they are aligned according to their left-most digit. For more numeric tables, we'd rather have the numbers aligned according to decimal position, so that the ones, tens, hundreds, etc., line up.

Notice also that the numbers are printed with varying degrees of apparent precision. They don't all contain the same number of digits to the right of the decimal point. In many applications, this too would be a fault.

For example, suppose the numbers in our 0-to-100 table represent average monthly rainfall figures, in inches or cubic centimeters, taken from around the country. The problem is, the rain gauges measure only down to tenths of a unit. Many of these averages contain far too many digits of precision.

This is a common problem: through computations such as division and multiplication, we end up with more apparent precision than we started with.

In such cases, we want to override TRS-80's standard method of displaying numbers. We can do this with a special form of PRINT and we'll cover that in the next chapter.

Graphic Demo. (Do It Yourself #20-3). This program uses TAB to plot points on the screen. The sine and cosine functions make nice curves, so we'll use them. You don't have to know trigonometry to appreciate the results of this program; just type it in and RUN.

(Be sure you first CSAVE the average rainfall program. We'll be using it again in the next chapter.)

```
NEW
10 CLS
20 DR=3.141593/180 'DEG.*DR=RADIANS
30 FOR ANGLE=-180 TO 179 STEP 6
40 X=ANGLE*DR 'CONVERT DEGREES TO RADIANS
50 SI=SIN(X): CO=COS(X)
60 SC=SI*30+32: CC=CO*30+32 'COLUMN POSITIONS
70 MN=-(SC<CC)*SC-(CC<=SC)*CC 'MN=MIN(SIN,COS)
80 MX=-(SC>CC)*SC-(CC>=SC)*CC 'MX=MAX(SIN,COS)
90 PRINT TAB(MN);"*";TAB(MX);"#"
100 NEXT ANGLE
110 GOTO 30
```

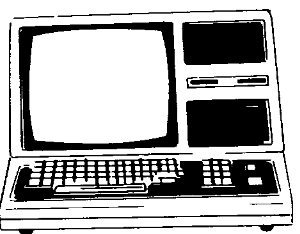
Lines 70 and 80 use "logical expressions" like SC<CC. This advanced subject is covered in your BASIC Reference Manual.

The program plots two points per line, SIN(X) and COS(X), by tabbing over an amount which is calculated from SIN(X) and COS(X). Lines 70-80 determine the minimum and maximum (smallest and largest) of the pair (SIN(X), COS(X)). After looking at line 90, can you see why we need to know the minimum and maximum? (See comments in Appendix A.)

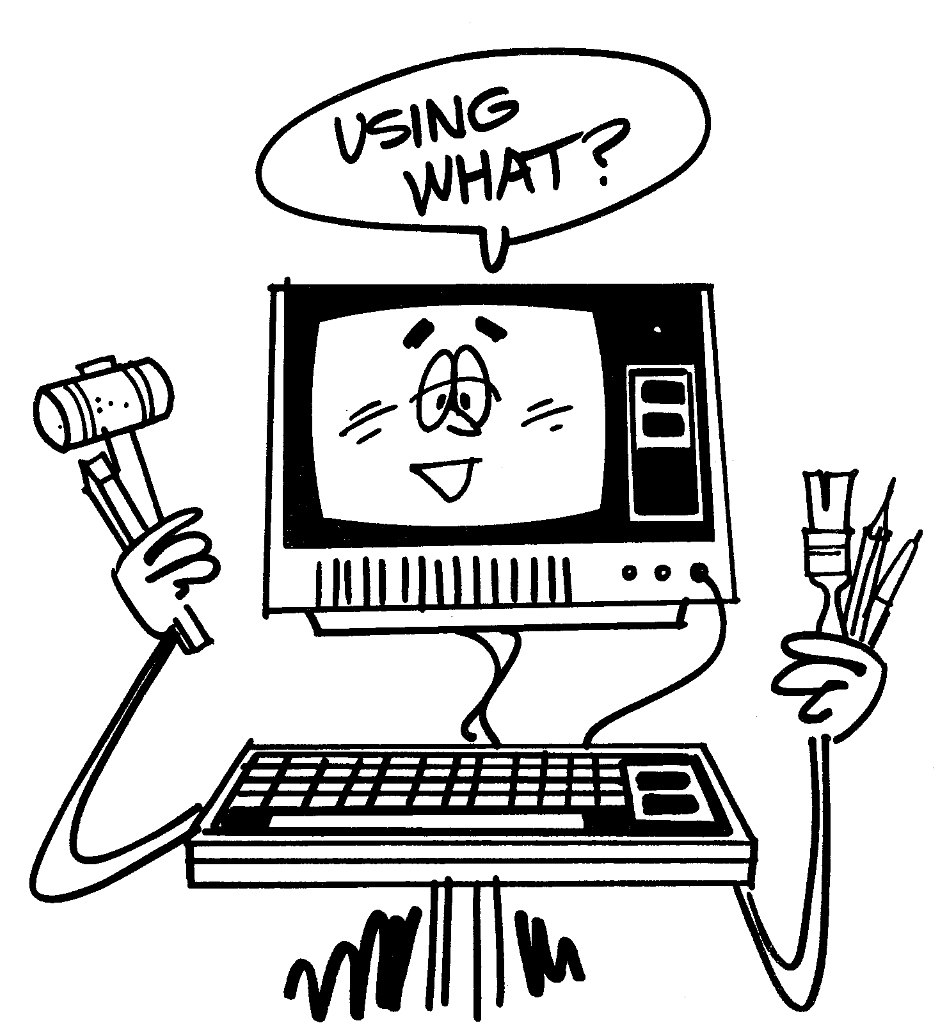
☑ Chapter Checkpoint #20

1. Which statement will print an asterisk in the fourth column of the current line?
   a. `PRINT TAB(3);"*"`
   b. `PRINT TAB(4);"*"`
   c. `PRINT TAB(5);"*"`
   d. `PRINT TAB(4),"*"`

2. Another way to vary column spacing across the Display is to vary the use of the punctuation marks, ____________ and ____________.



Chapter



Twenty One

PRINT USING

This chapter continues our exploration of ways to control the format in which data is printed.

In particular, refer to the table of numbers at the end of Chapter 20. We want to control the number of fractional digits printed . . .

The idea behind PRINT USING is to give TRS-80 a custom format to use for printing out the numbers. In this case, our format would tell the Computer to print all the numbers with the fractional part rounded to one digit. For example, we'd want 23.4888 to be printed as 23.5. Just for consistency, we'd want 12 to be printed as 12.0.

In general, we want our numbers printed in this format:

```
##.#
```

where each # stands for a digit, and the . shows where we want the decimal point.

Type in these statements (use the immediate mode):

```
PRINT USING "##.#"; 23.4888
PRINT USING "##.#"; 12
```

The new addition to PRINT consists of the word USING followed by the ##.# format. Notice that neither of these was actually printed. However, both numbers were printed using that format! 23.4888 was rounded to 23.5, and 12 was displayed with a zero to the right of the decimal point.

Now try these statements:

```
PRINT USING "##"; 12.4888
PRINT USING "##"; 12
```

Getting the idea? The characters inside the quotes tell your Computer exactly how to print the numbers. Used in this way, # and . are special "field specifiers" indicating respectively the number of digits to print and the position of the decimal point. There are many others you can use in a PRINT USING statement; we'll go over each of them in this chapter.

UFO #21-1

PRINT USING lets you control the format of output to the video display. Here's the general format:

PRINT USING *format; item-list*

format tells TRS-80 what format to use in printing each of the items in *item-list*. *format* can be a string constant enclosed in quotes or a string expression. *format* usually consists of a sequence of pre-defined field specifiers. Here are two of them:

- # Digit specifier
- Decimal point position

You've seen what happens when the number to be printed contains more digits to the right of the decimal point than are called for in the format string: the number is rounded to the specified number of digits.

You've also seen what happens when the number contains fewer digits to the right of the decimal point than are called for in the format string. TRS-80 adds zeros to fill the required number of decimal places.

What about the case where the number contains fewer digits to the left of the decimal point than are called for? For example, type in these lines:

```
PRINT USING "###.#"; 123
PRINT USING "###.#"; 12.3
PRINT USING "###.#"; 1.23
PRINT USING "###.#"; -1.23
```

Look closely at how the numbers are aligned on the display:

```
123.Ø
 12.3
  1.2
 -1.2
```

The first number has the specified number of digits to the left of the decimal point. Therefore, it is printed exactly as is. TRS-80 didn't even insert a leading space in front of the number, as it does with an ordinary PRINT statement. In printing the second, third and fourth numbers, TRS-80 inserts enough space in front of the digits to fill the field.

The final case to consider is when the number contains more digits to the left of the decimal point than the format allows for. To see what happens, type in this line:

```
PRINT USING "##.#"; 123.45
%123.5
```

TRS-80 went ahead and printed out the entire number, but it inserted a % sign to warn that the number did not "fit" into the specified format.

Try this one:

```
PRINT USING "##.#"; -12.345
%-12.3
```

The number didn't fit the format because the minus sign takes up a space, too. So TRS-80 put a "%" in front of the number as a warning, but printed the entire whole-number portion anyway. The fractional portion was rounded to suit the format requirements.

UFO #21-2

In PRINT USING statements, the format tells exactly how many characters (including blank spaces, decimal point, minus sign, etc.) are to be taken up by the number when it is displayed:

- To fill the required spaces to the right of the decimal point, TRS-80 adds zeros.
- To fill the required spaces to the left of the decimal point, TRS-80 adds spaces.
- If there are too many characters to the right of the decimal point, TRS-80 rounds the number.
- If there are too many characters to the left of the decimal point, TRS-80 prints all of them but prefixes a warning "%" to the number.

Program (Do It Yourself #21-1). Modify the "average rainfall" program (the 0-to-100 table in Chapter 20) to print out numbers rounded to one-tenth of a unit (one digit to the right of the decimal point). Allow for numbers that contain up to three digits to the left of the decimal point.

Important Note: To use both TAB and USING in a single PRINT statement, you must do the TAB first. For example:

```
PRINT TAB(15); USING "##.##"; 1.2345
```

The output from D.I.Y. 21-1 should have looked something like this:

A	B	C	D	E
89.3	61.8	86.9	87.4	12.6
90.7	11.5	87.8	38.6	11.5
9.8	4.0	42.9	92.5	69.4
42.5	21.4	69.8	84.7	49.5
44.3	83.1	90.3	53.4	7.2
39.9	26.8	21.2	74.9	51.1
75.9	5.2	38.5	97.7	21.2
91.2	63.5	63.7	96.6	55.3
35.5	97.6	32.8	7.3	16.5
59.9	5.6	40.9	89.9	67.1
94.0	87.9	14.3	53.3	25.9
68.0	70.4	9.3	21.3	32.5
13.7	37.5	56.2	21.0	21.5
1.1	99.7	10.9	44.7	74.7
72.9	60.6	58.2	95.0	92.0

So, we've controlled the precision of the output—but notice something else: all the numbers are now aligned according to the decimal point. This makes the output look much neater. For displaying tables of dollars and cents, this would be very important.

In the previous display, the numbers were printed "flush left" (pushed over to the left margin).

PRINT USING with a List of Variables

You can list several variables to be printed according to the format string. You can use either semi-colons or commas to separate them. However, TRS-80 will not tab over when it hits the comma. It will treat it just like a semi-colon. The same goes for trailing commas or semi-colons; either will cause TRS-80 to start the next printing immediately after the last character printed.

Type in this line:

```
PRINT USING "####"; 123, 456, 789
```

This prints out a line like this:

```
 123 456 789
```

There is one space in front of each number, since the format string allows for four digits to the left of the decimal point, and these numbers require only three.

Now this one:

```
PRINT USING "##"; 11,22,33
```

which prints out:

```
112233
```

This should convince you that PRINT USING does not automatically insert leading and trailing spaces around numbers. It prints out exactly as many characters as are called for in the format string.

UFO #21-3
You can't use commas in PRINT USING statements to cause printing in zones. If you use a comma it will have the same effect as a semi-colon. When TRS-80 prints a list of values according to a USING format, it starts at the beginning of the format string each time it prints a new item in the list.

USING Meets the Better Business Bureau

So far, USING has enabled us to control how many digits are displayed when a number is PRINTed, and where the decimal point appears. There are many other things we'd like to control when printing for business applications.

For example, in accounting tables, it's common to display financial figures with the sign (+ or −) to the right of the number instead of the left. Often only negative amounts ("debits") are shown with a sign. In some cases, we want a sign in *front* of every number, positive or negative.

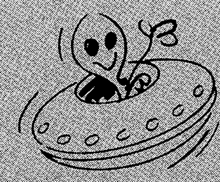
In our previous USING examples, positive numbers were displayed without any sign, and negative numbers had a leading sign. You can select either of these formats by careful placement of + and − signs inside the format-string.

For trailing + and − signs, place a + directly after the last # in the format string. For trailing − signs only, place a − directly after the last # in the format string. Try these examples:

```
PRINT USING "##.#+"; -12.3, 12.3
PRINT USING "##.#-"; -12.3, 12.3
```

For leading +/− signs on every number, place a + ahead of the first # in the format string.

```
PRINT USING "+##.#"; -12.3, 12.3
```



UFO #21-4

To control the position of +/− signs when numbers are printed out, use + and − inside the USING format string. For trailing − signs, put a − after the last #. For trailing + or − signs, put a + after the last #. For leading + or − signs, put a + before the first #. Examples:

```
PRINT USING "##.##-"; n     (Trailing -)
PRINT USING "##.##+"; n     (Trailing +/-)
PRINT USING "+##.##"; n     (Leading +/-)
```

What happens if you place a − ahead of the first # in the format string? Well, you might guess it would cause a − sign to be displayed in front of each negative number. Go ahead and try it:

```
PRINT USING "-##.#"; -12.3, 12.3
```

What? The Computer prints this:

```
-%-12.3-12.3
```

Here's what happened: The minus sign has a special meaning in a format string *only* when it follows the last # of a numeric field. If you use it anywhere else in a format string, TRS-80 interprets it as a literal character. In other words, the Computer assumes you want that exact character inserted at that exact position in every value printed!

Therefore TRS-80 inserted a – before it displayed the number –12.3. But –12.3 requires five spaces, and your numeric field contained only four: ##.#. That's why the Computer prefixed the % to the number before printing it. Then TRS-80 went on to print the second value, 12.3. Following your format string literally, it inserted a – before starting to print the number.

You can try embedding other literal characters inside a PRINT USING *format-string*. For example:

```
PRINT USING "-----> ##.#"; -1.23
```

In this case, "– – – – – > " (even the space after ">"), was treated as a literal string to be inserted before the numeric field "##.#".

Dollars and Cents

Now we're going to introduce a few more format specifiers which are especially useful for printing out financial figures: $ and *. We'll do it by way of a fantasy program. Type in these lines:

```
NEW
10 CLS: PRINT "***TRS-80 READI-LOAN CENTER***"
20 INPUT "TYPE IN YOUR NAME (NO COMMAS, PLEASE)"; N$
30 INPUT "NOW HOW MUCH $ DO YOU NEED? (UP TO $9999.99)"; M
40 IF M<0 OR M>9999.99 THEN 30
50 PRINT "HMMM . . ."
60 FOR I=1 TO 3000: NEXT I
70 PRINT "LOAN APPROVED"
80 PRINT: PRINT "PAY TO THE ORDER OF "; N$; "  ";
90 PRINT USING "####.## DOLLARS"; M
100 PRINT: INPUT "NEXT"; M
110 GOTO 10
```

Line 60: The program pauses for about 10 seconds while it "considers" your request.

Run the program.

Notice we're using literal characters in the format string of line 90 to add the word "DOLLARS" at the end of the numeric field. That's one way to handle financial figures. But it won't do for most applications. Here's a better way.

Change line 90 to:

```
90 PRINT USING "$#####.##"; M
```

Now run it again.

For Your Own Protection (Not That We Don't Trust You)

Run the program again, and ask for a loan of two dollars. The payout line will look like this:

```
$    2.00
```

If this were on paper, think how easy it would be to insert a few extra digits between the $ and the 2, suddenly increasing the loan amount drastically. TRS-80 is not about to allow such a crude ploy. In fact, it offers three devices to prevent tampering with numeric fields:

$$ The run-on $, which never leaves any room between the $ and the first digit.

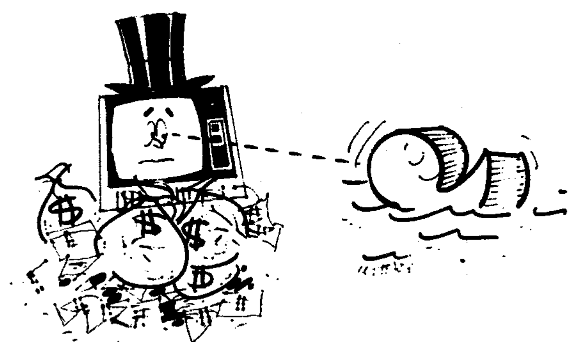
****** The asterisk fill, which fills all leading blanks with asterisks

****$** The fail-safe payout, which puts the $ just ahead of the first digit, and then fills any leading spaces with asterisks.

RUN the program with each of the following versions of line 90. Be sure to ask for small loans like $2, to demonstrate what happens to the spaces:

```
90 PRINT USING "$$####.## "; M
90 PRINT USING "******.## DOLLARS"; M
90 PRINT USING "**$####.##"; M
```

> **Model I Users:** The * will also be printed after the number if there is a specified trailing negative sign. For example:
> `"*****.##-"`



High Finance and the Floating Comma

When a number has more than four or five digits, it becomes difficult to read and comprehend. Exactly how many is 1000000? By counting digits, we discover it is one million.

Such numbers are a lot easier to read when we insert commas to separate hundreds, thousands, millions, billions, etc: 1,000,000.

There's a format specifier to do this: the comma. Try these immediate statements:

```
PRINT USING "#,##########"; 1234567890, 1234567, 123
PRINT USING "#,###"; 123, 1234
```

Only a single comma is required. It doesn't matter where you place it, so long as it's in the numeric field.

Note: Make sure your print field is wide enough to accommodate extra commas.
Example:

```
PRINT USING "#,######"; 1234567#
%1,234,567
```

Each comma inserted uses one print position, so leave room. It's a good idea to insert commas for the *maximum* field you might need so you can plan your space better.

Program (Do It Yourself #21-2). Change line 90 of the "Readi-Loan" program to print a "floating comma" in all amounts exceeding 999.99.

What Can USING Do for the Scientist?

Science has to deal with larger ranges of numbers, from very small ones such as .0000000001 to quite large ones like 100000000. Scientific notation was invented to handle such wide ranges conveniently. "E-notation" is the Computer's version of scientific notation.

What? You thought we put in one too many #? Actually, you might need that extra space for a minus sign sometimes.

The numbers above are represented like this:

Scientific Notation	**E-Notation**
1×10^{9}	1 E+09
1×10^{-9}	1 E-09

You can force TRS-80 to print numbers in E-notation (or D-notation) by adding a special sequence of characters after the numeric field: four up-arrows.

Remember, if it's a double-precision value, BASIC represents it with a D in place of the E.

Try these examples:

```
PRINT USING "####.###^^^^"; 1.234, 12.34, 123.4, 1234
```

Model III Users: Don't forget that your Display will show a [(left-bracket symbol).

Notice that every number is displayed with three digits to the left of the decimal point. Yet there are four # signs before the decimal point. The ↑↑↑↑ specifier has this quirk: you always specify one extra # position to the left of the decimal point. For example, if you want one digit printed to the left of the decimal point, and three to the right, use this format string:

```
"##.###^^^^"
```

So what happens if you use just one # ahead of the decimal point? Try it:

```
PRINT USING "#.###^^^^ "; 1.234, 12.34, 123.4, 1234
```

TRS-80 prints each number with a zero to the left of the decimal point

UFO #21-5
There are nine format specifiers to be used in conjunction with the numeric field specifier #.

.	Positions the decimal point
+	Ahead of first #, causes leading +/− sign to be printed. After last #, causes trailing +/− sign to be printed.
−	After last #, causes trailing − sign to be printed for negative numbers.
$	Use ahead of the first # to cause a $ to be printed ahead of the number.
$$	Use ahead of the first # for a run-on $ (no spaces between $ and first digit).
**	Use ahead of the first # to fill any leading spaces with asterisks.
**$	Use ahead of the first # for a run-on $ and asterisks in all leading spaces.
,	Use once anywhere in the #-field for "floating comma" in output number.
↑↑↑↑	(4-up-arrows) Use after #-field to display number in E-format. Will use D-format for double-precision numbers.

Characters which do not have pre-defined meanings terminate a numeric field and are treated as "literals" to be inserted into the output at the specified position.

PRINT USING for String Values

So far we've used USING to control how the Computer outputs numeric values. There are times when you'd like to do the same with string values. For example, let's go back to the Readi-Loan program.

```
10 CLS: PRINT "***TRS-80 READI-LOAN CENTER***"
20 INPUT "TYPE IN YOUR NAME (NO COMMAS, PLEASE)"; N$
30 INPUT "NOW HOW MUCH $ DO YOU NEED? (UP TO $9999.99)"; M
40 IF M<0 OR M>9999.99 THEN 30
50 PRINT "HMMM . . . "
60 FOR I=1 TO 3000: NEXT I
70 PRINT "LOAN APPROVED"
80 PRINT: PRINT "PAY TO THE ORDER OF "; N$; "   ";
90 PRINT USING " ####.## DOLLARS"; M
100 PRINT: INPUT "NEXT"; M
110 GOTO 10
```

Suppose we want to print a payee line like this:

```
PAY TO THE ORDER OF L. P. LONGNAME
```

Regardless of how the name is entered, we want to print it out as two initials followed by the last name. To do this, we'll need to input first name, middle name, and last name separately. So change line 20 and add lines 23 and 26:

```
20 INPUT "FIRST NAME"; N1$
23 INPUT "MIDDLE NAME"; N2$
26 INPUT "LAST NAME"; N3$
```

Now we want to replace line 80 with a PRINT USING statement that will print out a line like this:

```
PAY TO THE ORDER OF <1ST INITIAL>. <2ND INITIAL>. <LAST NAME>
```

Don't type this line (or the next). They are incomplete.

The first part is easy—we just include it inside the format string as a literal field:

```
PRINT USING "PAY TO THE ORDER OF . . .
```

Next we need to tell TRS-80 we want to display a string value, and how many characters to print in that value.

Experimentation Time

To indicate a single-character string value, use the symbol !. For example, type in these lines in the **immediate** mode:

```
N1$="JOSEPHINE"
N2$="LEE"
N3$="OVERHILL"
PRINT USING "!"; N1$
PRINT USING "!"; N2$
```

The "!" specifies a string value of length one, so TRS-80 displays only the first character of N1$ and N2$.

Now a couple of examples you may have to think about:

```
PRINT USING "!"; N1$, N2$
```

The print list contains two string variables, but the format specifies only a single string character. So TRS-80 uses the print list twice, once for each variable. Since the format doesn't include any blank spaces, both the values are packed together: "JL". To separate them, we can use this format:

```
PRINT USING "!  "; N1$, N2$
```

Do all these examples in the immediate mode, so BASIC will keep the current values of N1$, N2$, and N3$.

Another approach is to specify two distinct string values in the format string:

```
PRINT USING "!!"; N1$, N2$
```

Very Important: the "!!" specifies two string values of length one, not one string value of length two. So TRS-80 displays the first character in each string. To separate these characters, we insert a literal blank space:

```
PRINT USING "!  !"; N1$, N2$
```

Now we're getting close to solving the payee problem. How do you insert a period after each initial? (Hint: Make it a literal character.)

Answer:

```
PRINT USING "1ST INITIAL=!. 2ND INITIAL=!."; N1$, N2$
```

Okay, now we know how to print out the FIRST character of any string value. But how do we get an *entire* value?

This calls for a new format specifier: %% (a pair of percent signs), indicating a string value containing two characters. To increase the length of the string, add spaces inside the %%. Each space you add adds another character to the string.

For example, type:

```
PRINTUSING"%%";N3$(no spaces)
PRINTUSING"% %";N3$(one space)
PRINTUSING"%  %";N3$(two spaces)
PRINTUSING"%      %";N3$(six spaces)
```

Now we're getting quite close to the answer. Try this line:

```
PRINT USING "!.!.%      %"; N1$, N2$, N3$
```

Use six spaces between the %-signs.

Note that %   % asks for exactly eight characters—the length of OVERHILL. If N3$ contained a longer name, only the first eight characters would be displayed. So just to be safe, we might put 23 spaces between the %% to allow for N3$ to contain up to 25 characters. Here's the final version of line 80:

```
80 PRINT USING "PAY TO THE ORDER OF !. !. %%";N1$, N2$, N3$
                                          ↑
                                      23 spaces
```

We omitted them, but be sure *you* put 23 blank spaces between the %-signs.

Line 80 can be made much shorter by storing the format string in a string variable. Add line 75 and change line 80 as follows:

```
75 A$="PAY TO THE ORDER OF !. !. %          (23 spaces)          %"
80 PRINT USING A$; N1$, N2$, N3$
```

UFO #21-6

There are two string field specifiers for use with PRINT USING:

!	Gets the first character in the string
%%	Gets the first two or more characters in the string: %%gets two characters; % % gets three; each additional space gets an additional character. If the data doesn't contain enough characters to satisfy the %% field, it is printed with enough trailing spaces to fill the field.

Program (Do It Yourself #21-3). Change the resident program to include a rejection of, say, people born on odd-number days of the month. (You'll have to include a question of the date of the applicants birth, an IF. . .THEN statement, and a rejection notice.)

☑ Chapter Checkpoint #21

1. The primary purpose of PRINT USING is:
   a. To confuse the programmer.
   b. To improve the Computer's precision.
   c. To control the format in which data is printed.
   d. To speed up output to the display.

2. Which of the following is *not* a PRINT USING format specifier?
   a. "#"
   b. "!"
   c. "$"
   d. "@"

3. If PRINT USING finds a non-special character in its format-string, it will:
   a. Print it as a literal character.
   b. Give you a syntax error message.
   c. Ignore it.
   d. Print out its ASCII code.

Chapter

Twenty Two

Control That Cursor!

Your domination of the Computer's PRINT mechanism is nearly complete. You can control tab positions, format numbers down to the last decimal point, comma, and +/– sign; you can even control how many characters of a string are printed. Now we're going to tame the cursor.

This will be a very important accomplishment, because the cursor determines where printing starts. If you can position the cursor anywhere on the display, then you can print anywhere on the display.

The On-Again, Off-Again Cursor

We talk about this cursor as if it were a reliable, identifiable element on the display at all times. But the cursor disappears while a program is running, coming back only for an input statement. What good does it do to talk about the cursor position, when we can't *see* the cursor?

Our first exercise of authority over the cursor will be to turn it on—make it visible during program execution—even when it's not handling an INPUT.

Instead of printing letters and numbers, you can print a **control character** that tells TRS-80 to turn on the cursor. Your Computer won't display anything, but the cursor will come on at the current print position.

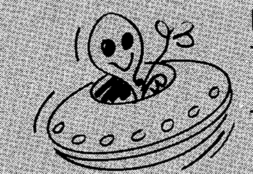
So how do you print a control character? Remember the CHR$(*n*) function from Chapter 19? We used it to print the character associated with one of the ASCII codes. For example, since 65 is the ASCII code for the letter "A",

```
PRINT CHR$(65)
```

prints that letter.

The control code meaning "cursor on" is 14. The control code meaning "cursor off" is 15. To prove it, try this short program:

```
NEW
10 CLS
20 PRINT "NOW YOU SEE IT . . . "
30 PRINT CHR$(14)     'TURN CURSOR ON
40 FOR I=1 TO 1000
50 NEXT I
70 PRINT "NOW YOU DON'T . . . "
80 PRINT CHR$(15)     'TURN CURSOR OFF
90 GOTO 90
```



UFO #22-1
To turn the cursor on during program execution, use
PRINT CHR$(14)
To turn it back off, use
PRINT CHR$(15)

Whenever you want to keep track of the current print position, place a PRINT CHR$(14) toward the beginning of your program. In this chapter, we'll use this trick often.

Before we can start telling the cursor where to go, we've got to have a way of specifying the destination. For example, suppose we want to position the cursor to the "middle" of the display:

Using Old Tools

Starting from the top left corner of the display, the "middle" is eight lines down and 32 columns over. So how do we position the cursor to the top left corner? The two ways we know about are CLS and (CLEAR).

Since the cursor starts there after a CLS, we call it the "home" position.

RUN this:

```
NEW
10 CLS                'CLEAR SCREEN AND HOME CURSOR
20 PRINT CHR$(14);    'CURSOR ON
30 FOR I=1 TO 7       'DROP DOWN 7 LINES
40 PRINT
50 NEXT I
60 PRINT TAB(31);"*" 'TAB(31)=32ND COLUMN
70 GOTO 70            'HOLD DISPLAY
```

Another way to position the cursor is to advance it one space at a time up to the desired position. This involves calculating the total number of print positions from "home" to the destination.

For example, to print an asterisk in column 32 of the second line, you must print a full line of spaces plus 31 additional spaces, for a total of 95 spaces:

It's a rather inefficient technique, but it will prepare you for another advancement in PRINT technology, to be introduced later in this chapter.

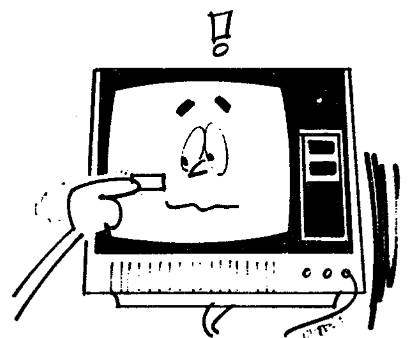
```
NEW
10 CLS
20 PRINT CHR$(14);
30 FOR I=1 TO 95
40 PRINT " ";     'ONE SPACE BETWEEN QUOTES
50 NEXT I
60 PRINT "*";
```

Program (Do It Yourself #22-1 to 22-5). Use the last technique to print an asterisk:
22-1. In the middle of the display.
22-2. In the middle of the last line.
22-3. At the second-to-last position on the last line.
22-4. At the last position on the last line (The display will automatically scroll up—there's no way to stop it when you print on the last position of the last line.
22-5. Final Question: How many print positions are there on the display?

Using old tools, we've been able to position the cursor by advancing it from a known position to a specific destination. But the techniques we used have obvious drawbacks:

1. You can't move the cursor back to a higher line unless you first erase the display with CLS.
2. You have to know where the cursor is before you move it.

There's a very intriguing way to get around the first drawback. It involves more control codes (like CHR$(14)).



Cursor Motion Controls

There are six control codes for moving the cursor without changing what's on the display:

Code	Cursor motion
24	Back (left)
25	Forward (right)
26	Down
27	Up
28	Home position
29	Beginning of line

Here's a demonstration program:

```
NEW
10 CLS
20 DEFINT A-Z
25 REM . . . SAVE CURSOR CONTROL CODES FOR CONVENIENCE
30 CN=14: CF=15: BK=24: FD=25: DN=26: UP=27: HM=28: BG=29
40 PRINT CHR$(CN);    'CURSOR ON
50 PRINT "DEMONSTRATION OF CURSOR MOTION"
60 PRINT "FORWARD MOTION": C$=CHR$(FD): GOSUB 500
70 PRINT "BACKWARD MOTION": C$=CHR$(BK): GOSUB 500
80 PRINT "UPWARDS MOTION": C$=CHR$(UP): GOSUB 500
90 PRINT "DOWNWARDS MOTION": C$=CHR$(DN): GOSUB 500
100 END
500 REM . . . SHOW OFF CURSOR CONTROLS
510 GOSUB 1000    ' DELAY
520 FOR I=1 TO 1024
530 PRINT C$;
540 NEXT I
550 RETURN
1000 FOR I=1 TO 300: NEXT I    'DELAY SUBROUTINE
1010 RETURN
```

There are many other control codes related to the video display. We'll cover some of them later. They are all listed in the Appendix, so you can experiment with them on your own.

So now you can move the cursor all over the display—but you've still got to know where the cursor is before you can move it to a specific place on the display.

TRS-80 has a special form of PRINT that solves both problems instantly. . .

PRINT @

The display contains 16 * 64 = 1024 print positions. Suppose we number each of them like this:

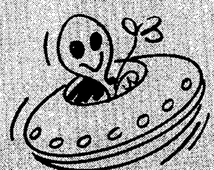
```
0,1,2,...



     ...,1022,1023
```

For a completely numbered screen, see Appendix E.

Having done this, we are ready for PRINT @.

Try these examples in the **immediate** mode:

```
CLS
PRINT @ 736, "*"
PRINT @ Ø, "*"
CLS
PRINT @ 736, "*";: PRINT @ Ø, "*"
```



UFO #22-2

PRINT @ lets you start printing anywhere on the display. It positions the cursor to the specified print location. The locations are numbered from 0 ("home") to 1023 ("end").

The general format for PRINT @ is:

PRINT @ *n*, *print-list*

n is a numeric expression between 0 and 1023. The comma after *n* is required and does not produce any tabbing. *print-list* must contain at least one value to be printed.

To position the cursor without printing anything, use this form:

PRINT @ *n*, "";

"" is a null string. There are *no* spaces between the quotes.

To combine PRINT@ and PRINT USING, you must place @ *n*, first, as in:

PRINT@ *n*, USING *format*; *print-list*

Having PRINT @ changes the whole nature of output to the video display. Without it, the display is like a constantly scrolling piece of paper; with it, the display is more like a blackboard on which you write over and over, erasing one section and rewriting that section. For example, let's bring back the clock program, but this time take full advantage of PRINT @ and the USING function as well. Type is this revised version:

```
NEW
10 INPUT "ENTER THE CURRENT HOUR"; HR
20 INPUT "ENTER THE CURRENT MINUTE"; MN
30 INPUT "ENTER THE CURRENT SECOND"; SC
40 CLS
50 CTR=8*64+26
60 FOR HR=HR TO 23
70 FOR MN=MN TO 59
80 FOR SC=SC TO 59
90 PRINT @ CTR, USING "##:##:##"; HR, MN, SC
100 FOR T=0 TO 280
110 NEXT T
115 NEXT SC
120 SC=0
125 NEXT MN
130 MN=0
135 NEXT HR
140 HR=0: GOTO 60
```

In the next few chapters, we'll be using PRINT @ quite often. Just to get ready, run this version of the sine-wave graphing program. This one uses a horizontal X-axis. Compare it with D.I.Y. #20-3.

```
NEW
100 'DEMONSTRATION OF PRINT @ CAPABILITIES (SINE GRAPH)
110 '
120 ' INITIALIZATION
130 CLS: CLEAR 100
140 DEFINT I-K
150 '
160 ' SET UP CONSTANT VALUES
170 IL=8*64+1    ' IL=START POS OF CENTERLINE
180 IC=31    ' COLUMN POSITION OF Y-AXIS
190 NL$=CHR$(26)+CHR$(24)    ' CURSOR MOTION (DOWN,BACK)
```

Although the program uses math functions, its real purpose in this book is to illustrate what's required when you want to graph *any* information. You may skip it and come back later. . .

```
200 X1$="Ø"     ' LEFT MARKING FOR X-AXIS
210 X2$="+"+NL$+"3"+NL$+"6"+NL$+"Ø"     ' RT. MARKING
220 Y1$="+1": Y2$="-1"     ' Y-AXIS MARKINGS
230 DR=1.745329E-2     ' DEGREES TO RADIANS CONVERSION FACTOR
240 '
250 ' DRAW X- AND Y- AXES
260 PRINT @ IL, STRING$(60,"-")   ' X-AXIS
270 FOR I=Ø TO 6Ø STEP 1Ø
280 PRINT @ IL+I, "+";     ' SCALE MARKINGS
290 NEXT I
300 FOR I=2 TO 14     ' Y-AXIS
310 IR=I*64+IC
320 PRINT @ IR, "!";
330 NEXT I
340 '
350 ' LABEL X- AND Y-AXES
360 PRINT @ 8*64, X1$;
370 PRINT @ 6*64+62, X2$;
380 PRINT @ 64+30, Y1$;
390 PRINT @ 15*64+30, Y2$;
400 '
410 ' GRAPH HEADING
420 PRINT @ 12, "SINE FUNCTION ON THE INTERVAL <Ø,36Ø>";
430 '
440 ' COMPUTE AND GRAPH SINE CURVE ON INTERVAL <Ø,36Ø>
450 FOR I=Ø TO 36Ø STEP 6
460 V=SIN(I*DR)
470 PRINT @ 1, "X=";: PRINT @ 54, "SIN(X)=";
480 PRINT @ 64, USING "###"; I;
490 PRINT @ 116, USING "##.######"; V;
500 IX=I/6+1   ' ADJUST RANGE OF X FOR DISPLAY
510 IY = -FIX(V * 6) + 8 'ADJUST RANGE OF Y FOR DISPLAY
520 IP=IY*64+IX   ' CONVERT ROW-COLUMN TO @-POSITION
530 PRINT @ IP, ".";   ' DISPLAY POINT
540 NEXT I
550 FOR I=1 TO 5ØØ: NEXT I: CLS: GOTO 25Ø
```

Note: Displaying graphs requires careful planning. First you decide how much of the display will be used. In this program, we decided to center the graph in the area bounded vertically by columns 1 and 61, and horizontally by rows 2 and 14. The center point of this area is at column 31, row 7 (represented in the program by IC and IL-1, respectively).

Line 260 prints a line of 60 hyphens; lines 270-290 goes back over this line, replacing every 10th hyphen with a + to indicate a scale.

Lines 300-330 print a vertical "line" composed of exclamation points. Notice how we move down the rows: to print at a particular row-column address (R,C), we use the formula:

*@-position = R * 64 + C*

To put an ! at column 31 of every line from 2 through 14, we use the PRINT statement:

```
PRINT @ I * 64 + 31, "!";
```

letting I count from 2 through 14.

Lines 360-390 add the numeric labels to the axes. Variable X2$ is a little special. Run the program, press (BREAK), then in the **immediate** mode type:

```
CLS
PRINT X2$
```

What's this? A string that prints vertically? It appears to contain four characters:

```
+
3
6
Ø
```

But that's just an appearance. X2$ also contains several control characters — the ones that move the cursor around. Look at line 190:

```
190 CM$=CHR$(26)+CHR$(24)
```

Each time we print CM$, the cursor moves down to the next line and backs up a space. Line 210 builds up a string by alternating quoted strings and CM$:

```
210 X2$="+"+CM$+"3"+CM$+"6"+CM$+"0"
```

Now X2$ contains both display characters and cursor motion controls. The statement,

```
PRINT X2$
```

causes this result: TRS-80 prints the a "+", then drops down a line, then backs up a space, then prints a "1", then drops down and back, then prints an "8", etc.

Lines 450-540 handle the sine-computations and graph the results.

Line 450: I counts from 0 to 360 degrees. This represents the complete sine-cycle. Since there are only 60 positions on the X-axis, we'll count by sixes. That way we'll plot one point for each hyphen on the axis.

Line 460: TRS-80's trigonometric functions require angles measured in radians, not degrees. This line performs the conversion.

Lines 470-490: In addition to graphing the curve, we want to show the actual values of X and SIN(X). Run the program again, and use SHIFT @ to pause so you can check these values. Isn't PRINT USING great? It ensures that all the SIN(X) values have a "uniform" look. Much more impressive than simply PRINTing unformatted values!

Lines 500-510 adjust the X and Y values to suit the requirements of our graph. I ranges over values from 0 to 360; therefore I/6 + 1 ranges from 1 to 61 — which matches the column range on our graph. Y can range from +1 to −1; therefore −Y * 6 + 8 ranges from 2 to 14, which matches the line range of the graph.

Line 520: Given a row-column address, this line computes the equivalent @ address.

Before you go on to the next chapter CSAVE the sine-graph program because you'll need it later.

☑ Chapter Checkpoint #22

1. You can turn the cursor on during program execution by typing PRINT CHR$ (________) and turn it "off" by typing PRINT CHR$ (________).

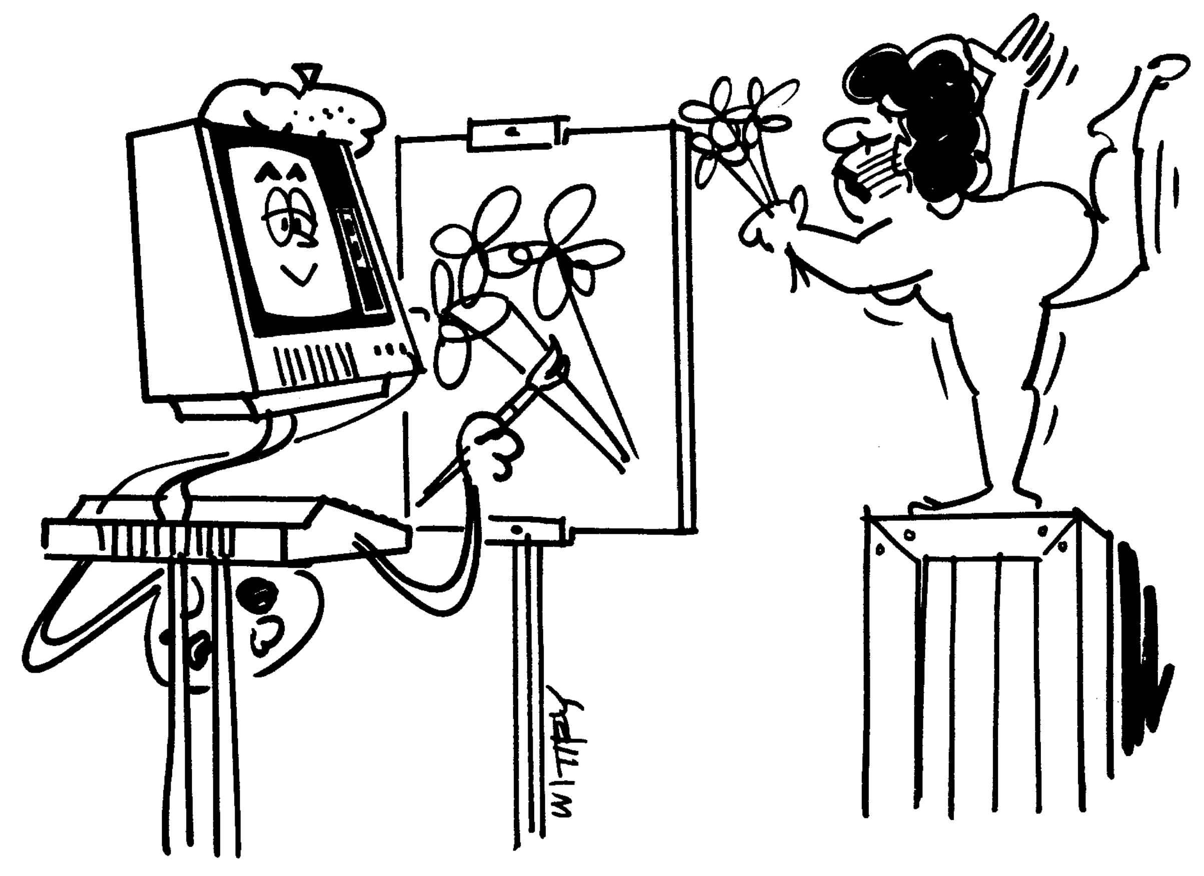
2. The cursor's "home" is:
   a. the upper-left hand corner of the display
   b. the lower-left hand corner of the display
   c. somewhere in Ivy Bend

3. PRINT @ lets you start printing __________ on the display.

4. The correct format for PRINT @ is:
   a. PRINT @*n, list-print*
   b. PRINT @ *list, n-print*
   c. PRINT @ *n, print-list*

5. (TRUE) (FALSE) You may combine PRINT USING and PRINT @.

Chapter



Twenty Three

Graphics

Graphics refers to two-dimensional art; computer graphics refers to computer-generated art which is printed on the display, printer, or other output device. Computer graphics can include tables, illustrations, and various other forms of artwork. Images printed on the display can even be animated, as we'll see later on.

In some programs, graphics techniques are used to add interest; for example, a space-war game might include a graphics routine to draw spaceships on the display. Other programs may produce "graphics for graphics' sake", for example, a program which prints out an image of Abraham Lincoln on a line printer.

Characters and Character Codes

Everything you PRINT can be thought of in two ways:

- As a sequence of characters — what you actually expect to see printed.
- As a sequence of codes — each numerical code representing one character. The computer uses these codes to represent the characters in memory and to send them to the display.

For example, the statement:

```
PRINT "ABC"
```

sends the following code sequence to the Display:

```
65, 66, 67, 13
```

When the Display receives the code 65, it creates the "A" character; 66, "B"; 67, "C"; the 13 causes the display to drop the cursor down to the beginning of the next line. (BASIC adds that last code to the text, since the PRINT statement didn't include trailing punctuation).

Strings are stored the same way. The name-it statement:

```
A$ = "ABC"
```

stores codes 65, 66 and 67 in the memory location called A$.

What's the relationship between a code and its "corresponding character"? It's a totally arbitrary one, based on a commonly accepted table called ASCII (American Standard Code for Information Interchange). ASCII associates a character or function with each of 128 codes (0-127). By extension, codes 128-255 can be assigned to additional characters or functions.

The TRS-80 Code for Information Interchange (TRSCII) is just an extension of ASCII, with a few changes in the code range 0-127. Right now, we're going to introduce TRSCII, paying special attention to how the codes affect the Display.

TRSCII includes 256 character codes (zero through 255). To store any of these codes in a string or to PRINT one, use the desired code as the argument to the CHR$(*n*) function. For example:

```
A$ = CHR$(90)
```

stores the character code 90 in A$. Since 90 is the code for "Z",

```
PRINT A$
```

will display the letter Z.

```
A$ = CHR$(14)
```

stores code 14 in A$. If you then PRINT A$ inside a program, the cursor will appear, since code 14 means "cursor on" to the display.

TRSCII can be divided into four functional groups:

- Text
- Control
- Graphics
- Space Compression

Text

Codes 32-127 represent the standard ASCII text characters: A-Z, a-z, 0-9, and punctuation symbols (see the table in the Appendix). Note that codes 97-122 are used to indicate the lowercase alphabet. But unless your computer is capable of displaying lowercase, these characters will be displayed as uppercase.

Run this program to see all the text characters with their codes:

```
NEW
10 CLS
20 FOR I=32 TO 127
30 PRINT USING "###=!   "; I; CHR$(I);
40 NEXT I
50 GOTO 50
```

Model I Users: Don't forget that your Computer will not display lowercase letters although you still have code numbers assigned to them (internally). Model III Users, however, can literally "see" what we mean when we talk about lowercase letters/codes.

In line 30, there are three spaces after the "!".

Control

Codes 0-31 represent the 32 control characters. Control characters are not displayed as such, but may perform a control function. (Some have no effect on the display.) Here's a list of control characters that have some effect on the display.

Code	Result when you PRINT CHR$(code)
8	Backspaces cursor and erases
10,11,12,13	Moves cursor to beginning of next line and erases to end of that line
14	Turns on cursor during program execution
15	Turns off cursor during program execution
23	Converts to 32 character/line display
24	Moves cursor back
25	Moves cursor forward
26	Moves cursor down
27	Moves cursor up
28	Homes cursor and converts to 64 character/line display
29	Moves cursor to beginning of line
30	Erases from current cursor position to end of line, without moving cursor
31	Erases from current cursor position to end of display, without moving cursor

You've already seen a few of these codes in action. We'll be demonstrating many more of them soon.

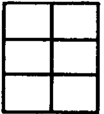
Graphics

Codes 128 to 191 correspond to the special graphics characters — a set of display patterns. You can use these for special effects, bar-graphs, etc.

To see the entire set of graphics characters, run this program:

```
NEW
10 CLS
20 FOR I=128 TO 191
30 PRINT USING "###=!      "; I, CHR$(I);
40 NEXT
50 GOTO 50
```

Each graphics character is a 2 x 3 "cell" in which various combinations of its six blocks are lit or "set":



For example, graphics character 128 has no blocks set; 129 has block 1 set; 130, block 2; 131, blocks 1 and 2; etcetera, all the way to 191, which has all blocks set.

Space Compression

Codes 192-255 have a very specialized purpose. They represent sequences of spaces (blanks):

192 zero blanks
193 one blanks
194 two blanks

And so on, up to:

255 63 blanks

These codes allow you to "compress" or abbreviate sequences of spaces into a single character. In some applications, this is very worthwhile, since storing blanks (one character for each blank) can be a waste of memory. See the demonstration program in Appendix B.

Note to Model III Users: Codes 192-255 may also be used to generate special characters. See the Model III Operation Manual for details.

For a demonstration, type in this line:

```
PRINT "1" + CHR$(254) + "2"
```

TRS-80 prints 62 spaces between the "1" and "2".

Codes vs. Characters

We've already used two functions involving character codes:

CHR$	Returns the specified character code as a string.
ASC	Returns the numerical code for the specified string character.

There's a third code-related function which sometimes involves converting a code into the corresponding character: STRING$. We demonstrated this function at the beginning of this book; now we'll explain it and put it to use.

Before trying out STRING$, type:

```
CLEAR 150
```

to reserve enough string space for our examples.

Now we're ready. Start by typing:

```
PRINT STRING$(64, "+")
```

You'll find that this prints a line of 64 plus-symbols.

You could do the same thing (less efficiently) with these statements:

```
FOR I=1 TO 64: PRINT CHR$ (43);:
  NEXT I
```

Here's another one:

```
PRINT STRING$(25, "5")
```

This prints a line of 25 "5"s. Got the idea? Now try this:

```
PRINT STRING$(50, 45)
```

As you can see, this prints out a string of 50 hyphens. It's not hard to understand why 50 of them were printed; this is because we specified 50 in the line above. What may be a bit more difficult to understand is why the line printed out a string of hyphens. If you look at the TRSCII table in the Appendix, you'll see that 45 is the code for "-". Similarly, since 88 is the code for "X",

```
PRINT STRING$(10, 88)
```

will print a string of 10 "X"s.

We can use STRING$ to print out a string of any character. For example, suppose we want to print a string of a certain graphics character, say, #150. Since we can't type in a graphics character from the keyboard, we use the code instead:

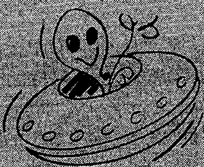
```
PRINT STRING$(100, 150)
```

which prints graphics character #150 on the display 100 times. So, when you're using STRING$, often you must specify the code rather than the character itself.

How about a string of control characters? Try this one:

```
PRINT STRING$(128, 8); "*";
```

That prints 128 backspaces — equivalent to two lines — before displaying the asterisk.



UFO #23-1

STRING$ returns a string with a length from zero to 255 characters. Every character in the string is the same. You specify the character directly or in terms of its code. Here are the two general formats for STRING$:

STRING$(*length*, *character*)
where *length* is a numeric expression with a value from zero to 255, and *character* is a string expression or constant. If a constant is used, it must be enclosed in quotes.

STRING$(*length*, *code*)
where *length* is defined above, and *code* is a numeric expression from zero to 255.

You can't do anything with STRING$ that you couldn't do in other ways. But STRING$ is more convenient than other methods.

The Real Thing

Enough of this theorizing. Let's draw some figures on the Computer's display. Type this in:

```
NEW
100 FOR K=1 TO 4
110 PRINT STRING$(4, 191);   ' SOLID BAR
120 PRINT CHR$(196);   ' FOUR SPACES
130 NEXT K
140 PRINT
```

and run it.

Remember from the graphics table that 191 is the code for "all blocks set", which produces a rectangle four times as high as it is wide. So, when we print a string of four of them, as we do in line 110, we get a solid square, with a solid square of blanks to the right.

There's not much we can do with these four squares. But if we had a few more of them . . . Don't type NEW; leave lines 100-140 in the Computer, and add these four lines:

```
200 FOR K=1 TO 4
210 PRINT CHR$(196);
220 PRINT STRING$(4, 191);
230 NEXT K
```

Type RUN. These new lines plus 100-140 will produce two staggered rows of squares.

Now are you beginning to see possibilities? Suppose we made lines 100-130 and lines 200-230 into subroutines, and then put them into a FOR/NEXT loop, so that first one subroutine, then the other, would be called. Suppose we set the FOR/NEXT loop to 4, so that this process would be repeated exactly four times. Add these lines to the ones already stored in memory:

```
10 CLS: CLEAR 100
20 FOR I=1 TO 4
30 PRINT: GOSUB 100 ' PRINT "X X X X" PATTERN
40 PRINT: GOSUB 200 ' PRINT "X X X X" PATTERN
50 NEXT I
60 END
140 RETURN
240 RETURN
```

Now run the program. If you've typed it in right, a small chessboard will be drawn on the display.

The complete program should look like this:

```
10 CLS: CLEAR 100
20 FOR I=1 TO 4
30 PRINT: GOSUB 100 'PRINT "XXXX  " PATTERN
40 PRINT: GOSUB 200 'PRINT "  XXXX" PATTERN
50 NEXT I
60 END
100 FOR K=1 TO 4
110 PRINT STRING$(4, 191); ' SOLID BAR
120 PRINT CHR$(196); ' FOUR SPACES
130 NEXT K
140 RETURN
200 FOR K=1 TO 4
210 PRINT CHR$(196);
220 PRINT STRING$(4, 191);
230 NEXT K
240 RETURN
```

The program which actually causes the Computer to play a game of chess on this chessboard we leave as an exercise to you. (But don't expect to find that one in the Appendix!)

Program. (Do It Yourself #23-1). Remember how to store cursor motion controls inside a string so we can print on several lines with a single PRINT? Run this program:

```
10 CLEAR 100
20 B$=STRING$(6, 191)   ' SOLID BAR
30 NL$=CHR$(26)+STRING$(6,24)   ' DOWN AND BACK
40 SQ$=B$+NL$+B$   ' CHECKER SQUARE
50 CLS
60 PRINT SQ$
```

Use this method to produce another checkerboard display. You'll need PRINT @ statements. Decide the starting position for each of the 32 white checker-squares. By the time you're done, you'll really know your way around the display!

Compare your program with ours in the Appendix.

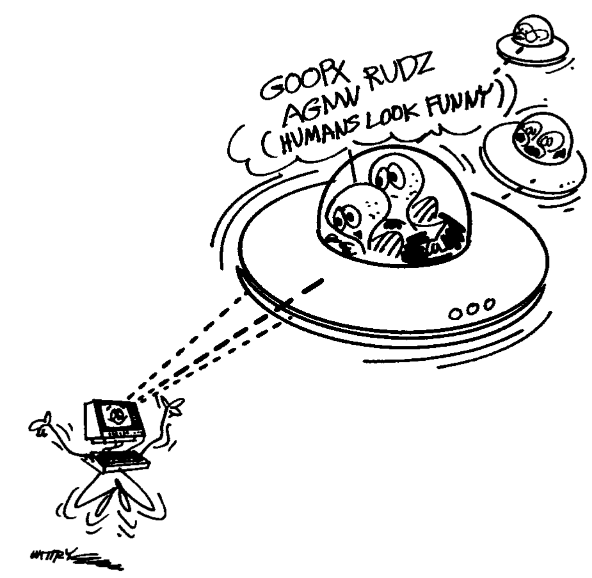
The Day The Earth Stood Still

We'll end this chapter with a demonstration of Computer animation. Then you'll probably want to start trying out some of your own neat ideas, using what you've learned in this chapter.

> But just so you don't abandon the book — we'll be presenting a whole new approach to graphics in the next chapter!

You've read so many UFO's by now, you wouldn't be surprised to see the real thing, right? So run this program.

Note: Pay particular attention to the placement of the asterisks and spaces in lines 140 and 150.




```
100 CLEAR 150: CLS
110 DEFINT A-Z
120 PRINT @ 960, STRING$(63,131);   ' LANDING SURFACE
130 CR$=STRING$(28,148)   ' CRAFT BODY
140 F1$ = "* * * * * * * * * *"   'BURN #1
150 F2$ = " * * * * * * * * * "   'BURN #2
160 BL$=CHR$(191+29)   ' 28 BLANKS
170 FOR I=0 TO 13
180   CP=I*64+17: BP=CP+68
190   PRINT @ CP, CR$;
200   FOR J=1 TO 10
```

```
210     PRINT @ BP, F1$;
220     PRINT @ BP, F2$;
230   NEXT J
240   IF I=13 THEN 260
250   PRINT @ CP, BL$;
260 NEXT I
270 GOTO 270
```

☑ Chapter Checkpoint #23

1. Which of the following doesn't belong with the others?
   a. text
   b. control
   c. appendix
   d. graphics
   e. space compression

2. Which of these STRING$ formats is incorrect?
   a. STRING$ (character length)
   b. STRING$ (length, character)
   c. STRING$ (length, code)

3. What's the difference between the characters represented by the codes 65-90 and those represented by 97-122?

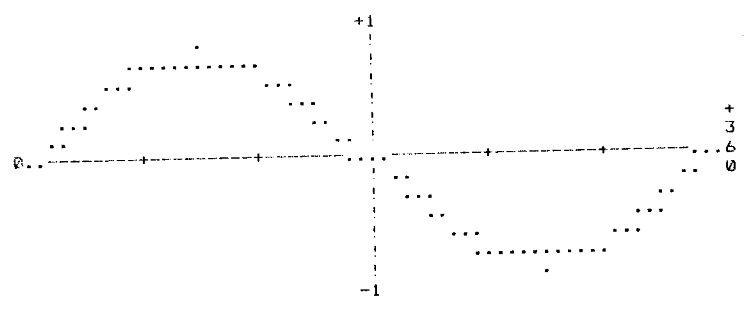
4. ____________ characters are not displayed as such, may perform a control function. An example of one is ____ which ____________.

Chapter

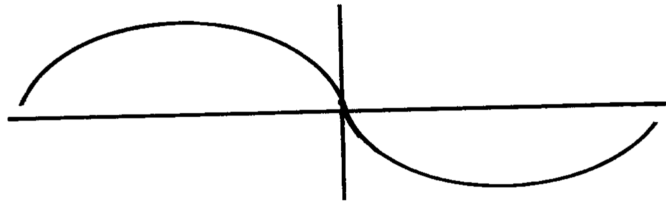
Twenty Four

Graphics — The Finer Points

Remember the sine-waving program? In it, we plotted values for SIN(X) where X ranged from zero to 360. The graph came out looking like this:



Looks good, right? But compare it with an ideal sine curve like this:



The difference is one of "resolution". The ideal sine curve has infinite points along each axis, while our TRS-80 version allows for just 60 points along the X-axis, and 14 along the Y-axis.

Although you can't attain high resolution with your TRS-80, you can significantly improve its resolution. We'll start by examining the "smallest known element" of the TRS-80 display. . .

Resolution is the ability to distinguish between objects. In our sine-wave graph, the horizontal resolution is equal to the width of one column. The vertical resolution is equal to the depth of one line.

The PRINT Cell

The PRINT cell is the space required by any single character you PRINT. Whether you're displaying a tiny period or a big capital X, the cell size is the same. RUN this program to prove it:

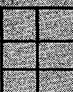
```
10 CLEAR 100: DEFINT A-Z: CLS
20 WH$=STRING$(64,191)   'SOLID WHITE LINE
30 FOR I=4 TO 12
40 PRINT @ I*64, WH$;    'PRINT A SOLID WHITE LINE
50 NEXT I
60 PRINT @ 7*64+31, ".";
70 PRINT @ 9*64+31, "X";
80 GOTO 80
```

Note: The two characters take the same size "chunk" out of the white display. That's the "cell size". Whenever you use PRINT, the resolution is limited by the size of the character cell.

Unlocking the Cell

Your Computer has three functions that let you specify or "address" subcell particles (as in subatomic particles, for you physics majors). We call these subcells "blocks."

Actually, you've already seen the "block". We described BASIC's graphics characters as 2 x 3 cells in which various combinations of its six blocks are lit or "set":



In fact, each graphics character (TRSCII codes 128-191) is composed of blocks.

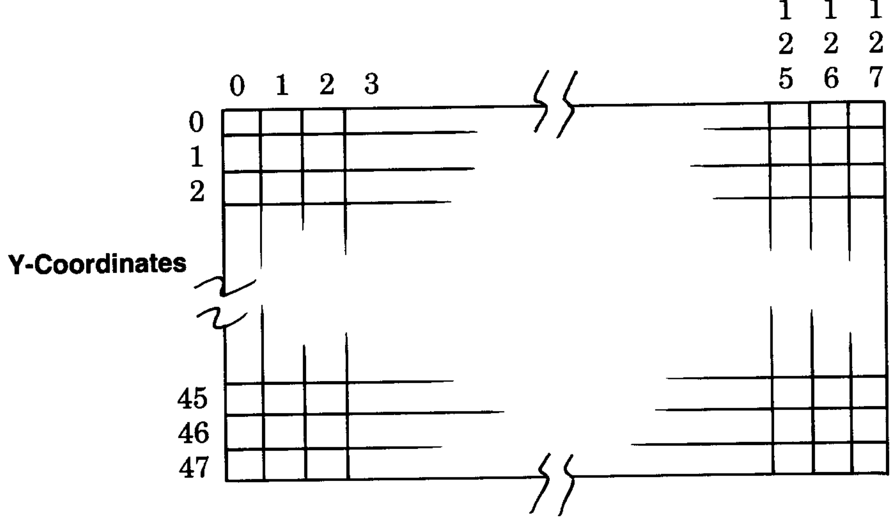
The three functions are:

- SET
- RESET
- POINT

SETting a Block

Imagine that each column on the display is made up of two subcolumns and each row is made up of three subrows. Then there are 2*64=128 subcolumns, and 3*16=48 subrows.

We number the subrows from 0 to 47; subcolumns from 0 to 127. And from now on, we'll refer to the subcolumns as X-coordinates, and subrows as Y-coordinates. This will help you think of the display as a graph:



With that layout in mind, RUN this program:

```
10 CLS
20 PRINT "SELECT THE BLOCK YOU WANT TO SET";
30 INPUT "X-COORDINATE (0-127)"; X
40 IF X<0 OR X>127 THEN 30
50 INPUT "Y-COORDINATE (0-47)"; Y
60 IF Y<0 OR Y>47 THEN 50
70 SET(X,Y)
80 FOR I=1 TO 300: NEXT I:   'DELAY
90 INPUT "PRESS <ENTER> TO SET ANOTHER BLOCK"; A
100 GOTO 10
```

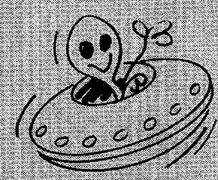
What coordinates would set the point in the exact center of the display? Test various X-Y combinations until you can predict approximately where every block will appear.

Using SET instead of PRINT @, we can smooth out that sine curve from the last chapter. Here's Sine-Wave III. Type it in and RUN it. (If you have a tape copy of Sine Wave II, load it in, and type in new lines for 420-end.)

```
100 ' DEMONSTRATION OF SET GRAPHICS (SINE GRAPH)
110 '
120 ' INITIALIZATION
130 CLS: CLEAR 100
140 DEFINT I-K
150 '
160 ' SET UP CONSTANT VALUES
170 IL=8*64+1   'IL=START OF CENTERLINE
180 IC=31   'COLUMN POSITION OF Y-AXIS
190 NL$=CHR$(26)+CHR$(24)   'CURSOR MOTION (DOWN,BACK)
200 X1$="0"   'LEFT MARKING FOR X-AXIS
210 X2$="+"+NL$+"3"+NL$+"6"+NL$+"0"   'RT. MARKING
220 Y1$="+1": Y2$="-1"   'Y-AXIS MARKINGS
230 DR=1.745329E-2   'DEGREES TO RADIANS CONVERSION FACTOR
240 '
250 ' DRAW X- AND Y-AXES
260 PRINT @ IL, STRING$(60,"-")   'X-AXIS
270 FOR I=0 TO 60 STEP 10
280 PRINT @ IL+I, "+";   'SCALE MARKINGS
290 NEXT I
300 FOR I=2 TO 14   'Y-AXIS
310 IR=I*64+IC
320 PRINT @ IR, "!";
330 NEXT I
340 '
350 ' LABEL X- AND Y-AXES
360 PRINT @ 8*64, X1$;
370 PRINT @ 6*64+62, X2$;
380 PRINT @ 64+30, Y1$;
390 PRINT @ 15*64+30, Y2$;
400 '
410 ' GRAPH HEADING
420 PRINT @ 1, "X="; TAB(20); "SINE FUNCTION ON <0,360>";
TAB(54); "SIN(X)=";
430 '
```

```
440 ' COMPUTE AND GRAPH SINE CURVE ON INTERVAL <0,360>
450 FOR I=0 TO 360 STEP 2
460 V=SIN(I*DR)
470 PRINT @ 64, USING "###"; I;
480 PRINT @ 116, USING "##.######"; V;
490 IX=I/3+2   'ADJUST RANGE OF X FOR DISPLAY
500 IY=-FIX (V*18)+24  'ADJUST RANGE OF Y FOR DISPLAY
520 SET (IX, IY)  'DISPLAY POINT
530 NEXT I
540 FOR I=1 TO 500: NEXT I: CLS: GOTO 250
```

We've doubled the resolution along the X-axis, from 60 to 120 points; and tripled it along the Y-axis, from 12 to 36 points. That's as smooth a curve as you're going to get with your TRS-80. You can draw horizontal and vertical lines smoothly and continuously, but any diagonal line is going to have a stair-step effect.



UFO #24-1

The SET statement allows you to turn on any of the graphics blocks. Each block is identified by a pair of values called X and Y coordinates. The X-coordinates give the position relative to the left of the display; the Y-coordinates give the position relative to the top of the display. The general format for SET is:

SET(x, y)

where x is a numeric expression from 0 to 127, and y is a numeric expression from 0 to 47. Use SET for applications requiring higher resolution than that afforded with PRINT.

Note: If you SET into a cell that already contains a text character, that character will be erased.

RESETting a Block

Of course, you can use CLS for a wholesale wipeout. Or you can PRINT blanks.

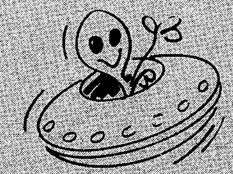
Dagwood's First Rule of Shop Practices — Don't turn anything on unless you know how to turn it off.

But even this last method lacks precision or resolution, if you will. Printing a space (or any other character) over a graphics block erases not just one block, but all of the six that may be set in that particular cell.

What's needed is an operation to specify a particular graphics block to be turned off. The TRS-80 has it, and it's called RESET.

RUN this demonstration program, in which we paint a white window on the display, and then let you reset points in the window:

```
10 CLEAR 100: CLS
20 XL=10: XR=117    'LEFT AND RIGHT X-BOUNDARIES
30 YT=15: YB=38    'TOP AND BOTTOM Y-BOUNDARIES
40 GOSUB 120    'SUBROUTINE TO PAINT WINDOW WHITE
50 PRINT @ 0, "SELECT THE BLOCK TO BE RESET"
60 PRINT @ 64, "X-COORDINATE;"; XL; "-"; XR;: INPUT X
70 IF X<XL OR X>XR THEN PRINT @ 64, CHR$(30);: GOTO 60
80 PRINT "Y-COORDINATE:"; YT; "-"; YB;: INPUT Y
90 IF Y<YT OR Y>YB THEN PRINT @ 128, CHR$(30);: GOTO 80
100 RESET(X,Y)
110 PRINT @ 64, CHR$(30): PRINT CHR$(30);: GOTO 60
120 ' PAINT WINDOW WHITE--ONE BLOCK AT A TIME
130 FOR X=XL TO XR
140 FOR Y=YT TO YB
150 SET(X,Y)
160 NEXT Y,X
170 RETURN
```



UFO #24-2

RESET works like SET, except that it turns a graphics block off. If the block is already off, it does nothing.

Note: If you RESET into a cell that contains a text character, that character will be erased.

Did you notice how long it takes to paint the window white? That's because you're painting one block at a time. Which brings us to another "Useful and Factual Observation":

UFO #24-3
For fast, efficient programs, use SET and RESET only when required. If PRINT @ can accomplish the same result, then use it.

In general, if you want to specify entire print cells, use PRINT @. It will be much faster.

To demonstrate: Notice that the boundaries of our window happen to coincide exactly with print cell boundaries:

The X-coordinates (10-117) correspond to columns 5-58
The Y-coordinates (15-38) correspond to rows 5-12

So instead of painting the window one block at a time, we can do it one line at a time. DELETE lines 120-170 and insert these new ones:

```
120 'PAINT WINDOW WHITE--ONE LINE AT A TIME
130 LW=(XR-XL)/2+1   'LINE WIDTH
140 WH$=STRING$(LW, 191)   'LINE CONTENTS (WHITE CELLS)
145 CP=XL/2   'COLUMN POSITION
150 FOR R=YT/3 TO YB/3   'ROW POSITION
160 PRINT @ R*64+CP, WH$;
170 NEXT R
180 RETURN
```

RUN it and see what a difference it makes to have a bigger paint brush.

POINTing at a Block

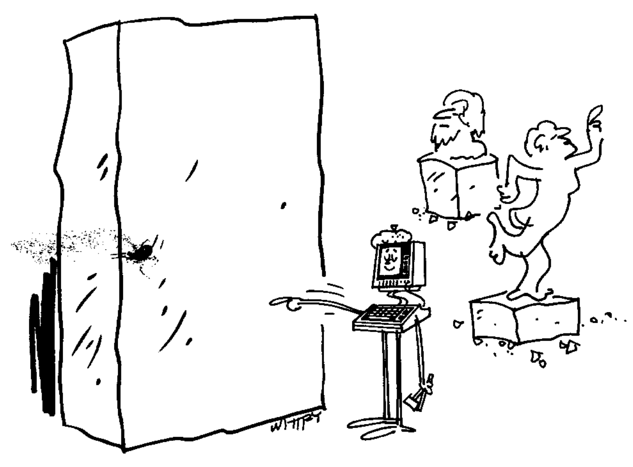
The third and final graphics operation lets you examine any block to see whether it's *on* or *off*. The function is called POINT (it points at any block so you can examine it), and it works like this:

If the block (X, Y) is *off*, POINT(X, Y) returns a 0. If it's *on*, POINT(X, Y) returns a -1.

Since POINT is a function, it has to be used in some kind of statement. Here's a typical example (don't type it in — just look):

```
IF POINT(X, Y)=-1 THEN PRINT "BLOCK"; X; Y; "IS ON"
```

However, there's an even simpler way of using the same statement, based on the fact that POINT returns a −1 or a 0.



First we've got to introduce a subtle feature of IF . . . THEN: Instead of putting a test after IF, you can put a numeric expression. If the expression has a zero value, IF acts as if a test had failed. If the expression is non-zero, IF acts as if a test had passed. The POINT function was designed to take advantage of this feature.

Try resetting a block that you've already reset.

Here's the alternate form of the last example:

```
IF POINT(X, Y) THEN PRINT "BLOCK"; X; Y; "IS ON"
```

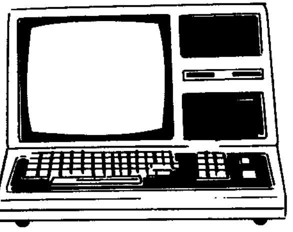
Add these lines to the main program and run it.

```
90 IF Y<YT OR Y>YB THEN PRINT @ 128, CHR$(30);: GOTO 80
92 IF POINT(X,Y) THEN 100  ' BLOCK IS ON--SKIP WARNING
94 PRINT @ 192, "THAT BLOCK IS OFF ALREADY!!!";
96 FOR DLAY=1 TO 660: NEXT DLAY: PRINT @ 192, CHR$(30);
98 GOTO 110
```

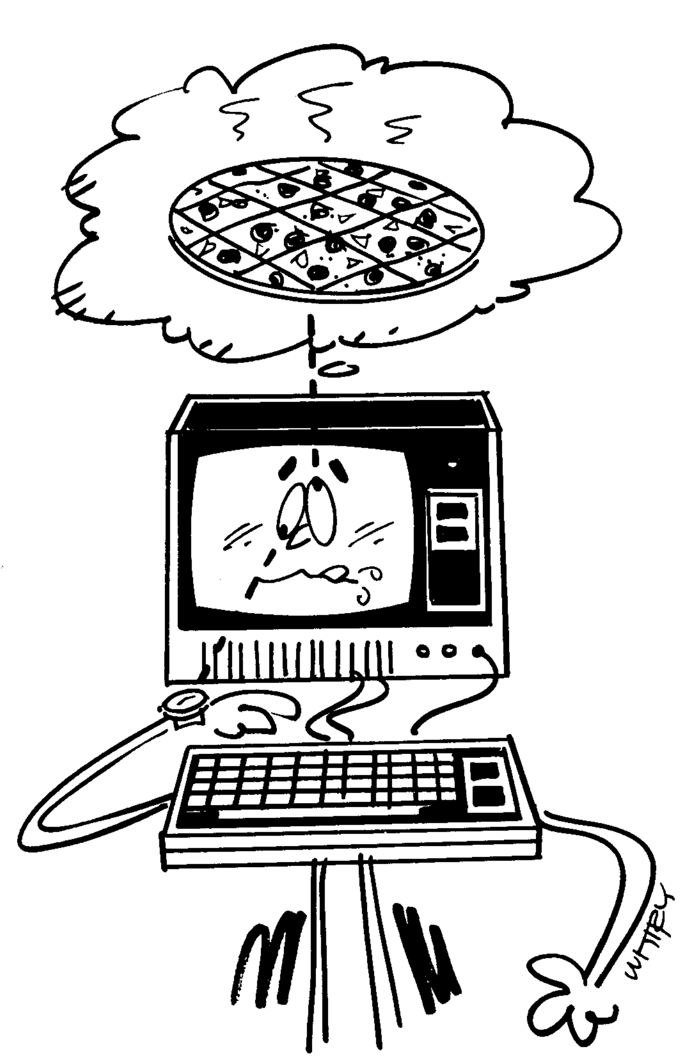
In the next chapter, we're going to put all these features to work. . . while we play! (We'll also introduce a keyboard feature that'll open up whole new programming possibilities.)

☑ Chapter Checkpoint # 24

1. The ____________ statement allows you to turn on a block while the ____________ statement enables you to turn it off.
2. You can predict where a block will appear on the Display if you know its ____________ and ____________ coordinates.
3. Which of the following doesn't belong with the others?
   a. RESET
   b. SET
   c. SINE
   d. POINT



Chapter



Twenty Five

Keys Alive!
Or, What to Do Until the Pizza's Ready

If you've ever played one of the video games while waiting for a pizza (or any other time), you've experienced the excitement of real-time interaction between man and machine: *you* position your paddle, cannon, hockey-player, etc. while the ball, target, hockey-puck, etc. is moving. *You don't have to wait your turn.*

Now, equipped with the full power of TRS-80 graphics, you'll want to write your own real-time games.

So how do you interact with a program? Through the keyboard, of course. . . with INPUT. But what happens during an INPUT? Everything comes to a halt — paddle, puck, target, and all — until you press (ENTER) to tell the Computer to look at what you've typed in. So much for real-time.

Getting past this road block will require a new key input operation — one that is able to get the input "on the fly."

INKEY$ — The Key to Interactive Programming

At all times during program execution, TRS-80 is keeping track of the last key pressed since you typed run. It stores this character in a place we'll call the "key latch." (If you haven't pressed any keys, then the latch is empty.)

The INKEY$ function returns this character as a string value and clears the latch (empties it). If no key has been pressed, INKEY$ returns the null string. Exception: pressing (BREAK) will always stop the program. INKEY$ won't normally return (BREAK) as a character.

The important thing about INKEY$ is, it doesn't wait for you to press (ENTER). INKEY$ simply gets the character already in the key latch (if there is one) and goes on.

By using INKEY$ at opportune moments in your program flow, you can make it seem like TRS-80 is doing two (or more) things at once — checking the keyboard and performing other tasks specified by the program!

Type in this little program:

```
NEW
1000 DEFINT A-Z
1010 CLS: PRINT TAB(24); "UP-DOWN COUNTER"
1020 T=1   'START WITH A POSITIVE INCREMENT
1022 'SELECT UP OR DOWN ARROW (ARW$)
1023 IF T=1 THEN ARW$=CHR$(91) ELSE ARW$=CHR$(92)
1030 PRINT @ 540, USING "! #####"; ARW$; K; 'DISPLAY ARW$ &
COUNT
1040 IF INKEY$="" THEN 1060   'IF LATCH EMPTY, CONTINUE
1050 T=-T   'REVERSE SIGN OF INCREMENT
1060 K=K+T   'COUNT BY INCREMENT T
1070 GOTO 1023
```

Now RUN it. It starts out counting up. Press any key (except (BREAK), of course) and it will begin counting down. Each time you press a key (any key) it reverses the counting increment.

So — you *can* interact with the program via the keyboard!

Lines 1023 and 1030 determine which arrow is displayed next to the count. When counting up, an up arrow or ↑ will be displayed; down, a down arrow or ↓.

Line 1040 performs the INKEY$ function in the simplest possible manner. In this program, we don't care which key has been pressed; we only want to know if any key has been pressed at all. The test:

```
IF INKEY$="" THEN 1060
```

tells whether the key latch is empty or not. If it is empty, the program keeps counting with the same increment (line 1060). But if the key latch is not empty (INKEY$ is not ""); then a key has been pressed. In that case, line 40 reverses the sign of the increment.

Note to Model III Users: So far you've had it pretty easy, but there is a program exception for you in this up-down counter.

You will have to modify line 1023 to read:

```
1023 IF T=1 THEN
ARW$=CHR$(94)ELSE
ARW$=CHR$(118)
```

Well, it has to do with the Display of CHR$(91) and CHR$(92). Model III (as you're well familiar with by now) displays a [and a \ for these two characters. (They will work in the program but they just won't look as good.)

Instead, we've inserted the character code for a ^ (code 94) and a lower-case v (code 118). They're not up-down arrows but they show direction up/down just as well.

There are no spaces between the quotes in line 1040.

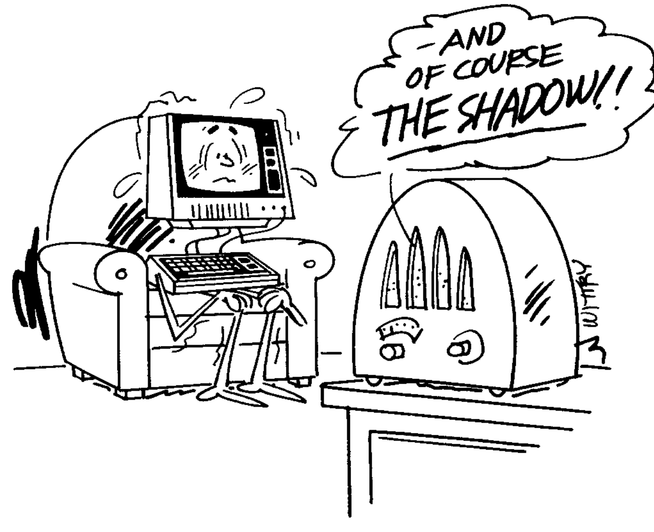
There are *no* spaces inside the quotes!

Notice something else about the program: the keys you press while it's running are not displayed. We never said they would be, but after so many INPUT operations, you may have expected it.

Once again, here are the facts about INKEY$.

> **UFO #25-1**
>
> The INKEY$ function returns the character currently stored in BASIC's key-latch. This will be the last key pressed since execution started, or since an INKEY$ was last performed.
>
> The character returned is always a string value. It can be any key except (BREAK) or (SHIFT) @. (BREAK) always interrupts the program, and (SHIFT) @ always pauses it.
>
> If no key has been pressed, INKEY$ returns the null string (INKEY$ = "").

Program (Do It Yourself #25-1). Change the up-down counter program so that it refuses to let the counter go past a certain limit (say, ± 100). Check our sample in Appendix A, but keep in mind that it's just one of an infinite number of choices.



Who Knows What Lurks in the Key Latch? INKEY$ Knows. . .

In the last program, we used INKEY$ to test whether the key latch was empty or not. Now we'll show how to retrieve a character from the keyboard so we can examine it.

Type in this program:

```
NEW
1320 DEFINT A-Z
1330 CLS
1340 A$=INKEY$   'CHECK KEY LATCH
1350 IF A$=""THEN 1340   'IF EMPTY, TRY AGAIN
1360 IF ASC(A$)<32 THEN 1390   'A$ WAS A CONTROL CHARACTER
1370 PRINT "YOU PRESSED THE ' "; A$; " ' KEY"
1380 GOTO 1340
1390 PRINT "NOT A TEXT CHARACTER--ITS CODE IS"; ASC(A$)
1400 GOTO 1340
```

> There are no spaces between the quotes in line 1350.
>
> The apostrophes in line 1370 are inside double-quotes, so they do not mean REMark.

Run the program. Try every key — each text key (A-Z, 0-9, punctuation) you press will be echoed on the display. Notice what happens when you press (ENTER). That key has a character code just like the text keys — it's 13.

Remember from our discussion of codes? Those below 32 are called control codes. In this program, we don't want to PRINT control codes since some control codes do strange things to the display. So when you press a control key, we want to display the code rather than the character represented by that code.

Now try pressing other "control keys" — the back arrow, forward arrow, down arrow, and (SHIFT) versions of all the arrows. Bet you didn't know there were so many different characters available from the keyboard!

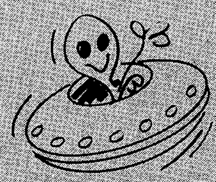
The reason you may not have noticed it is that during an INPUT, these control keys are used to control things — they do not become part of the value you're INPUTting. For example, pressing ⊖ during an INPUT erases the last character typed. It does NOT add a backspace character (code 8) to the value you're entering.

How the program works:

Line 1340 saves the contents of the key latch in A$. In this program, we can't do anything until we have an actual character to look at. So line 1350 causes control to loop back to line 1340 until a key has been pressed.

Line 1350 checks for control characters by comparing the ASCII code of A$ with 32. (32 represents the first text character.) We don't want to PRINT control characters, so line 70 simply shows you the key code.

What would happen if we *did* PRINT the control characters? Look at the table in the graphics chapter describing what happens when each control code is PRINTed.



UFO #25-2

INKEY$ can return any character the keyboard will generate except the codes for (BREAK) and (SHIFT) @. So before echoing the character, you should test to see that it is a text character. (For special applications, you may want to display control characters.)

Here's a simple way to ignore control characters returned by INKEY$ as well as null strings (assume A$ contains the character):

A$<" " (There is one blank space inside the quotes.)

If A$ precedes the space, it is either null or contains a control character. In either case, you want to wait for another character.

Parlèz-Vous Graphics?

Since INKEY$ does not echo your input automatically, this gives you the chance to "translate" keys into different characters before displaying the character. For example, you can assign graphics characters to certain keys.

We'll modify the last entered program so that each time you press SHIFT and one of the alphabet keys, a graphics character is displayed.

Add these lines to the resident program:

```
1355 IF ASC(A$)>96 THEN 1410   'A$ IS A SHIFTED A-Z
1410 A$=CHR$(ASC(A$)+33)   'MOVE CODE TO GRAPHICS RANGE
1420 PRINT A$: GOTO 1340
```

Note to Model III and Model 4 Users: To make these added lines (1355-1420) work, you must have your keyboard in the upper/lowercase mode (SHIFT 0 in Model III, and either SHIFT 0 or CAPS in Model 4. Then, press SHIFT for normal letters; don't press SHIFT for "graphics translation."

How and When to Clear the Key Latch

Suppose you want keyboard input during a specific portion of a program's execution cycle. Since INKEY$ saves the last key pressed anytime during execution, how are you going to screen out entries made at the wrong time?

Run this program to see the problem and the solution.

```
NEW
1470 CLS
1480 PRINT TAB(20); "TABLE OF SQUARE ROOTS"
1490 N=1: F$="####  ##.#####"
1500 FOR I=0 TO 39
1510 PRINT @ I*16+128, USING F$; N, SQR(N);
1520 N=N+1
1530 NEXT I
1540 PRINT: PRINT "PRESS <C> TO CONTINUE, <Q> TO QUIT"
1550 A$=INKEY$: IF A$="" THEN 1550 'LOOP UNTIL NOT EMPTY
1560 IF A$ <> "C" THEN 1590   'NOT <C>, SO CHECK FOR <Q>
1570 PRINT @ 128, CHR$(31)   'ERASE CURRENT PAGE
1580 GOTO 1500   'CONTINUE
1590 IF A$="Q" THEN END   'QUIT
1600 GOTO 1550   'INVALID KEY PRESSED--GET ANOTHER
```

Run the program. The "live keyboard" adds some class to a basically dreary number-cruncher! Press C to see the next page, Q to quit. Any other key will be ignored.

But back to clearing. . . run the program again. While the program is printing the current page of square roots—before it asks you to press (C) or (Q)—press (C) "accidentally" once or twice.

"What happened? I didn't get a chance to look at that page!" That's because the key latch contained the accidental (C) you pressed. So the INKEY$ in line 1550 returns a "C", and the program thinks you are ready to continue.

There's a quick and simple solution to this problem. It depends on the fact that INKEY$ empties the key latch. So just before you enter the key-input section of your program, insert a "dummy" INKEY$ operation. To fix the resident program, add this single line:

```
1545 A$=INKEY$   'FLUSH KEY LATCH OF PREVIOUS ENTRIES
```

Now run the program again. You'll find that to make it continue (or quit), you have to press the correct key *at the correct time.*

Numeric Entry to an INKEY$ Routine

INKEY$ returns string values only. So how do you go about entering numbers? Here's where the built-in function VAL comes in handy.

VAL accepts a string argument and evaluates it as a number. If the string characters don't make sense in a number, it returns a zero. For example:

VAL (argument)	Numerical result
VAL("8")	8
VAL("1" + "." + "3")	1.3
VAL("5" + "E" + "2")	5E+03 (5000)
VAL("XYZ" + "1")	0
VAL(" " + "100" + "A")	100

Note: VAL stops evaluating at the first "illegal" character.

So, to evaluate an INKEY$ entry as a number, use this approach (don't type this in):

```
A$ = INKEY$
A = VAL(A$)
```

Here's a program to demonstrate numeric entry with INKEY$ (and also check your reflexes).

```
NEW
1630 CLS
1640 DEFINT A-Z
1650 X=RND(5)-1: Y=RND(5): G=999
1660 Z=X+Y
1670 PRINT @ 8*64+20, "WHAT'S"; X; "+"; Y; "? ";
1680 A$=INKEY$   'FLUSH KEY LATCH OF PREVIOUS ENTRIES
1700 FOR I=100 TO 0 STEP -1   'START TIMER
1710   A$=INKEY$   'CHECK KEY LATCH
1720   IF A$<"0" OR A$>"9" THEN 1770   'EMPTY OR INVALID
1730   PRINT A$;
1750   G=VAL(A$)   'CONVERT STRING TO NUMBER
1760   I=0   'FORCE END OF LOOP
1770 NEXT I
1780 IF G=Z THEN PRINT " *** RIGHT ***": PRINT: GOTO 1820
1790 PRINT: IF G=999 THEN PRINT "TOO LATE--";: GOTO 1810
1800 PRINT: PRINT "WRONG--";
1810 PRINT "ANSWER IS"; Z: PRINT
1820 FOR I=1 TO 800: NEXT I   'DELAY
1830 CLS: GOTO 1650
```

RUN the program. For a real challenge, change line 1700 to:

```
1700 FOR I=30 TO 0 STEP -1
```

In this program we're doing two things simultaneously — running a 100-to-0 timer, and waiting for you to type in a digit. If time runs out first, a "TOO LATE" message is displayed. If you enter a digit in time, the program checks to see if it's the right one, and displays an appropriate message.

Line 1650 gets random values for X and Y. X will range from 0 to 4, and Y from 1 to 5. Therefore Z will range from 1 through 9.

Line 1720 causes the program to ignore all key-entries except characters between 0 and 9. We didn't even bother to make a separate test for A$="": wrong-key-pressed and no-key-at-all are handled the same way.

As soon as you have typed in a digit, the program assumes you're through. After all, Z has to be a single-digit number between 0 and 9.

Harder Problems Beget Smarter Programs

Oh, so you've graduated beyond one-digit addition. . . even at high speeds. Now you want to be tested on two-digit addition.

We still have to input one digit at a time. The trick is to take each digit as it's received, and save it in a "builder" string. When the builder string contains the correct number of digits, we evaluate the builder string with VAL.

Make the following changes and additions to the resident program and run it:

```
1650 X=RND(45)+4: Y=RND(46)+4: G=999
1690 B$=""   'CLEAR BUILDER STRING
1700 FOR I=500 TO 0 STEP -1   'START TIMER
1740   B$=B$+A$: IF LEN(B$)<2 THEN 1770
1750   G=VAL(B$)   'CONVERT STRING TO NUMBER
```

What to Do When the Pizza Still Isn't Ready

We started out this chapter talking about video games. And then we went and spent the whole chapter on arithmetic drills, up-down counters, character translation, and the like. We hope you found it interesting. And we know that if you put the techniques covered in this chapter to use, you can create some fantastic programs.

Video game programs tend to be rather long . . . with all the possibilities of score-keeping, special messages, etc. We've included some games in Appendix B. Type them in, run them, and modify them to your heart's content.

But just to close this chapter, here's a little program that lets you control the speed of an object while it's moving. The program was written to be improved upon (see the D.I.Y. at the end of this chapter).

Run this program:

```
NEW
1100 DEFINT A-Z
1110 CLS: PRINT @ 26, "M O T I O N"
1120 PRINT TAB(12); "PRESS '->' TO SPEED UP, '<-' TO SLOW DOWN"
1130 T=40   'INITIAL DELAY VALUE
1140 I=10   'INCREMENT FOR CHANGING SPEED
```

```
1150 FST$=CHR$(9)    'SPEED UP CHARACTER
1160 SLO$=CHR$(8)    'SLOW DOWN CHARACTER
1170 A$=CHR$(179)+CHR$(140)    'MOVING OBJECT
1180 B$=STRING$(2,8)+STRING$(8,25)    'ERASE AND ADVANCE
STRING
1190 PRINT @ 8*64, "";    'INITIAL POSITION
1200 PRINT A$;    'DISPLAY OBJECT AT CURRENT POSITION
1210 K$=INKEY$    'CHECK KEYLATCH
1220 IF K$="" THEN 1270    'IF EMPTY LEAVE SPEED AS-IS
1230 IF K$ <> FST$ THEN 1260    'WAS -> PRESSED?
1240 T=T-I: IF T<1 THEN T=0: GOTO 1280    'YES: SPEED UP
1250 GOTO 1270
1260 IF K$=SLO$ THEN T=T+I    'IF <- WAS PRESSED, SLOW DOWN
1270 FOR D=1 TO T: NEXT D    'DELAY (SPEED CONTROL)
1280 PRINT B$;    'ERASE AND ADVANCE CURSOR POSITION
1290 GOTO 1200
```

Each time you press →, the object speeds up (until it reaches maximum speed; after that → has no effect). Each time you press ←, the object slows down. It will get slower than cold molasses if you press ← enough times. Press → to speed it up again.

Program Notes:

Lines 1150-1160. In this program, we're going to check for two particular characters — the back arrow and forward arrow. These keys produce codes 8 and 9 respectively.

Line 1170. We combine two graphics characters to make this little rocket.

Line 1180. B$ is a very active string. When PRINTed, it erases the preceding two cells, then advances the print position by 8 cells. To display the rocket, we PRINT A$; to erase it and advance the print position, we PRINT B$; the next time we PRINT A$, the cursor will already be in the advanced position.

Lines 1220-1230 determine if a key has been pressed. If not, the speed is left unchanged for this go-round.

Lines 1230-1260 adjust the speed of the object by increasing or decreasing T, the delay control variable.

Line 1270. This FOR/NEXT loop determines how fast the object moves, since it determines the delay before B$ is printed (erasing A$) and A$ is re-printed at the next position.

Program (Do It Yourself #25-2). You can easily modify the program so that pressing the up arrow (key code 91) moves the print position up one line (display control code 27); and pressing the down-arrow (key code 10) moves the print position down one line (display control code 26).

This one line adds the up-arrow feature:

```
1222 IF K$=CHR$(91) THEN PRINT B$; CHR$(27);: GOTO 1200
```

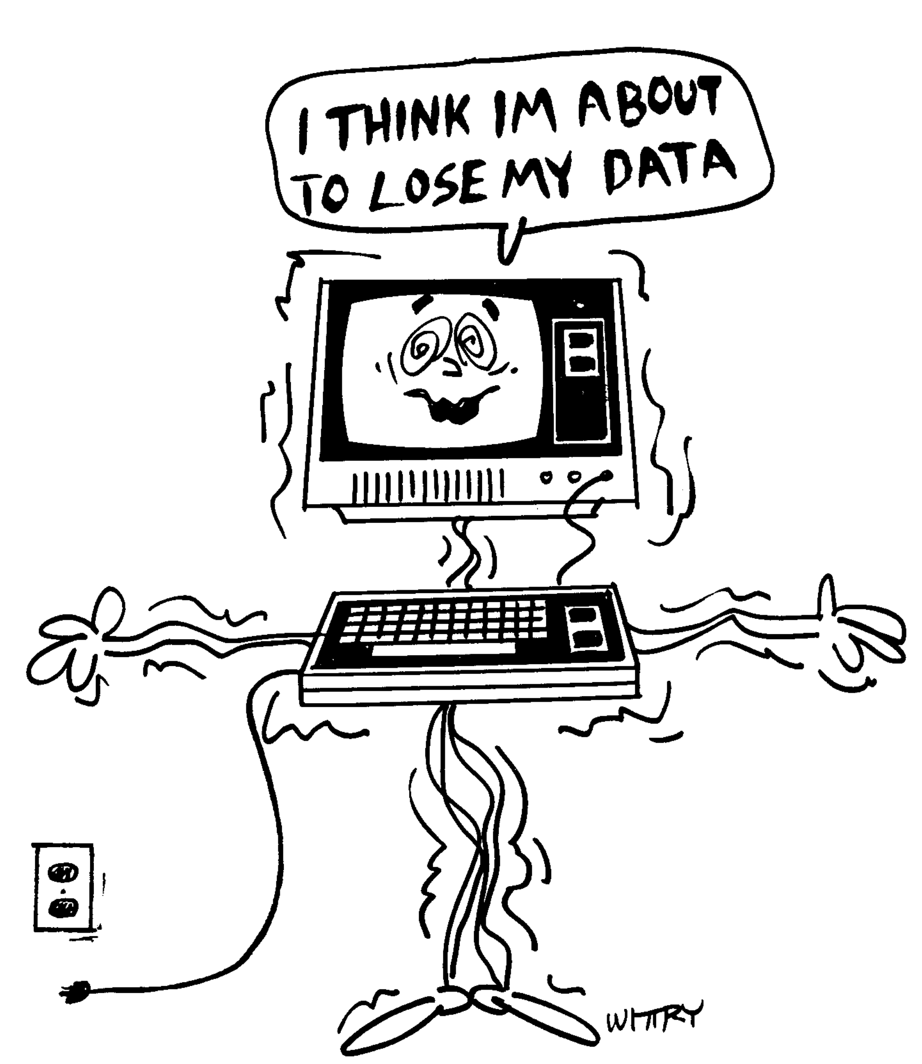
Add line 1224 to provide the down-arrow feature.

Compare your answer with ours in Appendix A.

☑ Chapter Checkpoint #25

1. INKEY$__________ the character currently stored in TRS-80's key latch.
2. There are two keys that will not be "seen" by the INKEY$ function — they are [ ] and [ ] [ ].
3. The built-in function__________ comes in handy whenever you need to enter a number within an INKEY$ function.

Chapter



Twenty Six

Does Your Data Go Away (When the Power Goes Off)?

No, that's not a song title — it's a serious question. You've been saving programs on tape for quite a while now, reloading them hours or days later. But what about the data that is input to your programs — or the results that are output by the programs? How can you save that?

Good news — you already know most of what you need, namely, INPUT and PRINT. With slight modifications, these two statements will save all your data on tape, and read it back when you need it during program execution.

Furthermore, PRINT can be slightly modified in another way to send its output to a line printer instead of to the display.

We'll Start with Tape I/O

To change PRINT and INPUT from display to cassette operations, you need only add a special "device tag," # – 1. For example, (don't type these in):

```
PRINT #-1, 12345
INPUT #-1, A
```

The device tag tells TRS-80 you are addressing the cassette rather than the display.

Punctuation for PRINTing to cassette is simple: you just add a comma after every PRINT item except the last. This comma has nothing to do with the video PRINT comma; it merely separates the items in the PRINT list.

For Model I Owners Only: If you have the Expansion Interface, you can address either of two cassettes, by using tags #-1 and #-2.

Set up your recorder by pressing the PLAY and RECORD buttons, then run this demonstrator:

```
NEW
10 PRINT "NO PUNCTUATION, PLEASE"
20 INPUT "ENTER YOUR NAME"; N$
30 INPUT "ADDRESS"; AD$
40 INPUT "CITY"; CT$
50 INPUT "STATE - ZIP CODE"; SZ$
60 INPUT "AGE"; AG%
70 INPUT "PRESS <ENTER> WHEN RECORDER IS READY TO RECORD"; X
80 PRINT "RECORDING . . . "
90 PRINT #-1, N$, AD$, CT$, SZ$, AG%
100 PRINT "THE INFO IS NOW SAVED ON TAPE"
110 PRINT "TURN OFF THE RECORDER"
```

LINE 90 stored all the information on tape. We could have used separate PRINT # - 1 statements for each variable, but PRINTing them all with a single statement uses less tape and time.

To retrieve the data, you simply execute a single INPUT #-1 statement to match the PRINT #-1. Type in this program:

Note: There is a limit to how much you should try to output with a single statement. More on this later. . .

```
NEW
200 PRINT "REWIND THE TAPE TO THE BEGINNING"
210 INPUT "PRESS <ENTER> WHEN THE RECORDER IS READY TO PLAY"; Y
220 INPUT #-1, N1$, A1$, C1$, S1$, G1%
230 PRINT "DATA HAS BEEN READ IN. TURN OFF THE RECORDER"
240 PRINT "HERE'S WHAT WAS READ"
250 PRINT N1$: PRINT A1$: PRINT C1$: PRINT S1$: PRINT G1%
```

Simple, isn't it? To keep it simple, follow these rules:

1. *Always start recording at the beginning of a "clean" tape.* Use a new one or bulk-erase an old one. Use a leaderless tape (one in which the entire tape can store recorded information).

2. *The INPUT #-1 statement must match the PRINT #-1 statement that created the tape, in terms of number of values in the item list, and type of variables.* It's okay to output both numbers and strings with a single PRINT #-1, but be sure to input them in the same order with the INPUT #-1. Compare lines 90 and 220 above: different variable names, yes, but the number of variables and the sequences are the same.

3. *Don't put too many values in the cassette PRINT list.* If you do, the extra ones will never get written (even though no error message will appear). To be specific, a single cassette PRINT statement can output a maximum of 248 characters — this includes every space, digit, letter and other symbol included in your data.

To gauge how many characters a PRINT #-1 statement will write, figure out how many characters an identical video PRINT statement would output to the display. Then add one additional character for each comma required in the cassette PRINT list.

Trouble-Shooter (Do It Yourself #26-1). Two of the following PRINT #-1 INPUT # - 1 pairs contain errors according to rules 2 and 3 above. Find them, and explain what's wrong with them. In every case, assume:

```
A$="JOHN JACKSON"
B$=STRING$(140,"#")
X=1.234567890
Y=3000.0001
```

Pair #1:

```
PRINT #-1,A$,B$,X,Y
INPUT #-1,H$,L$
```

Pair #2:

```
PRINT #-1,X,Y,X+Y
INPUT #-1,A,B,C
```

Pair #3:

```
PRINT #-1,A$,B$
INPUT #-1,H$,D$
```

Pair #4:

```
PRINT #-1,"THIS IS THE FIRST ITEM",1,2,3
INPUT #-1,T$,A,B,C
```

Pair #5:

```
PRINT #-1, X, B$, Y, B$
INPUT #-1, X1, L$, Z1, G1$
```

Hard Copy

You haven't "really" got a full-fledged data micro-processing center until you can churn out reams of paper filled with important figures.

First You Get a Printer...

There are many Radio Shack models to choose from. Some print on aluminized-paper (for the economical space-age look), others on regular paper. Some print up to 132 characters per line; others 80, etc. Some can even print the full ASCII set (upper and lower case, etc.).

... Then You Add an L

That's right, a simple L is all you need to change display-PRINT into line printer PRINT. The same L will turn display-LIST into line-printer-LIST. For example:

```
100 INPUT "WHAT DO YOU WANT TYPED"; A$
110 PRINT "HERE'S YOUR HARD COPY . . . "
120 LPRINT A$
```

Assuming you've got a printer connected and on-line, run the program. To LIST the program to the printer, type:

```
LLIST
```

Everything you know about LIST applies to LLIST, and most of what you know about PRINT applies to LPRINT . . . In fact, the only thing that absolutely doesn't work with LPRINT is the @-position operation. USING and TAB work exactly the same and commas and semi-colons have the same effect with one exception — trailing punctuation.

On the printer as on the display, trailing PRINT punctuation suppresses the automatic carriage return. But unlike the display, the printer may not "wrap around" when the line gets too wide for the paper or for the printer itself. For this reason, use trailing semi-colons only when you know there will be a carriage return before the line exceeds the limits of the paper or of the printer.

You can output control codes to the printer like this:

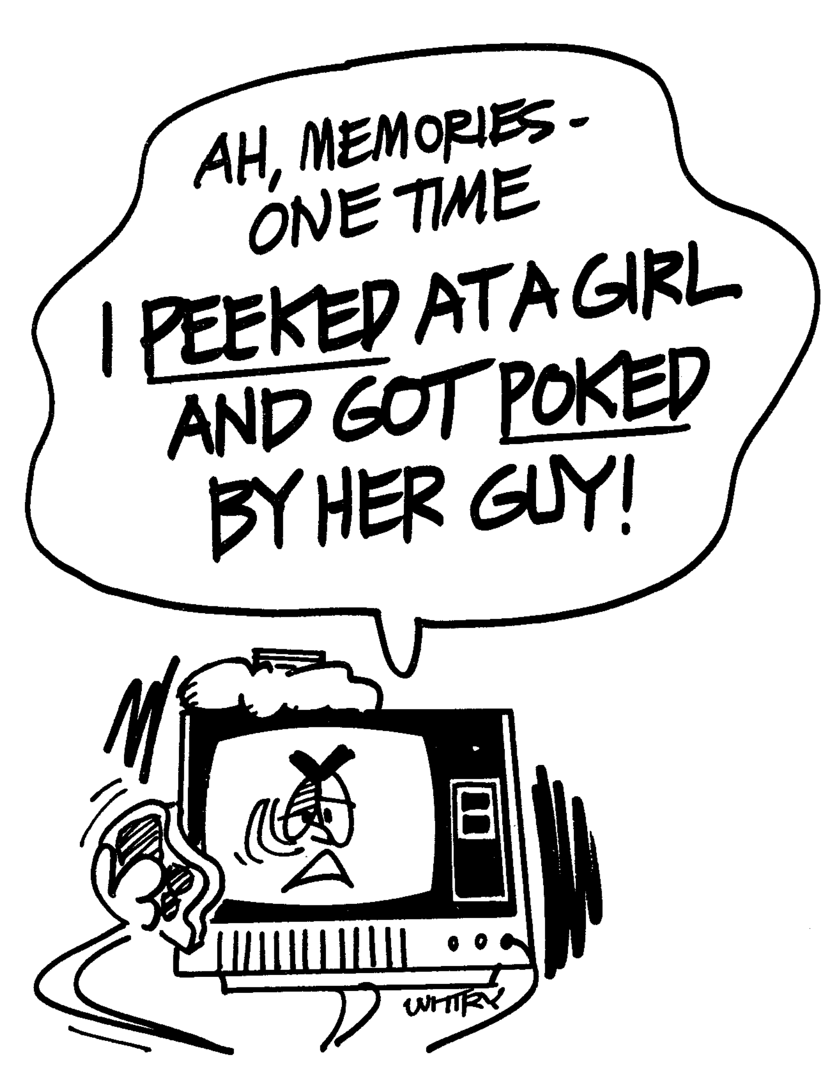
```
LPRINT CHR$ (code)
```

but the effect the code has depends on the printer. See the line printer owner's manual for details.

☑ Chapter Checkpoint #26

1. Which of the following will NOT get you in touch with the line printer?
   a. LLIST
   b. LLINE
   c. LPRINT
2. To change PRINT and INPUT from display to cassette operations, you need only add the special device tag,__________.
3. Can you identify the correct pair?
   a. PRINT #-1, 12345
      INPUT #-1, A
   b. PRINT 12345, #-1
      INPUT A, #-1
4. In order to save your data (and keep your sanity in the process), you should:
   a. always use a "clean" tape
   b. match your INPUT statement to your PRINT statement
   c. not put too many values in the cassette PRINT list
   d. all of the above
   e. none of the above

Chapter



Twenty Seven

Peeking and Poking Down Memory Lane

Back in Chapter 4, we said that a variable was a name for a place in memory. In Chapter 17, we said the CLEAR *n* command reserves space in memory for string storage. In Chapter 16, we also said that DIM reserves space in memory for array storage.

Those three references are the closest we've gotten to exploring how your Computer holds the information you input.

That's one of the advantages of BASIC (or any high-level programming language). It frees you from concerns over minute details involving the use of memory. You can write a program focusing on the problem at hand, letting BASIC take care of "internal" considerations (where will this and that value be stored, etc.).

Still, there are advantages to "direct memory addressing." In this chapter, we're going to take a quick look at some of them, focusing on a few special cases. We won't go very far though, because the subject gets very complicated very quickly. For a more in-depth treatment, we recommend a couple of Radio Shack books:

LEVEL II BASIC Programming Techniques,
by William Barden, Jr.; Catalog Number 62-2062.
TRS-80 Assembly Language Programming,
by William Barden, Jr.; Catalog Number 62-2006.

A Byte without Teeth

You've already encountered terms like 16K RAM and 12K ROM. Generalized definitions of these terms are easy to come by (see the Glossary in your Computer Owner's Manual), so we'll focus on how they apply to your TRS-80.

The "M" in RAM and ROM stands for "memory." The "RA" and "RO" stand for the two kinds of memory, "random access" and "read only." Random access is often called read/write memory, since you can read its data *and* write new data into it. Read only memory, on the other hand, has a permanently fixed contents; you can read it, but cannot write new data into it. Your TRS-80 Computer contains a BASIC language interpreter in 12K of ROM; it also contains either 4K or 16K of RAM.

You can expand your system to include an additional 16K or 32K of RAM.

6K, 12K, etc. are measures of how much memory you've got. One "K" contains 1024 units of memory, so 16K = 16384 units.

The unit of memory is the **byte**. A byte can store a value from zero to 255. (If you want a larger range of numbers, you'll need more than one byte. For example, to store an integer-type variable, BASIC always uses two bytes.)

Every byte in the Computer's memory has its own address. In your TRS-80 Computer:

- The 12K ROM occupies addresses zero through 12287.
- Addresses 12288 through 16383 are reserved for special purposes (we'll look at some of these shortly).
- Your 4K RAM (or 16K, or whatever you have) starts at 16384. However, TRS-80 takes some of this for internal "bookkeeping" functions.
- The memory that actually stores your program starts at 17129.
- In a 4K Computer, memory ends at 20479; 16K, 32767.

If you have added 16K RAM, your memory ends at 49151; if you have added 32K RAM, your memory ends at 65535. For further information, see your Owner's Manual.

How to Read a Byte of Memory

There's a simple function that tells you the contents of any memory location. The function is PEEK.

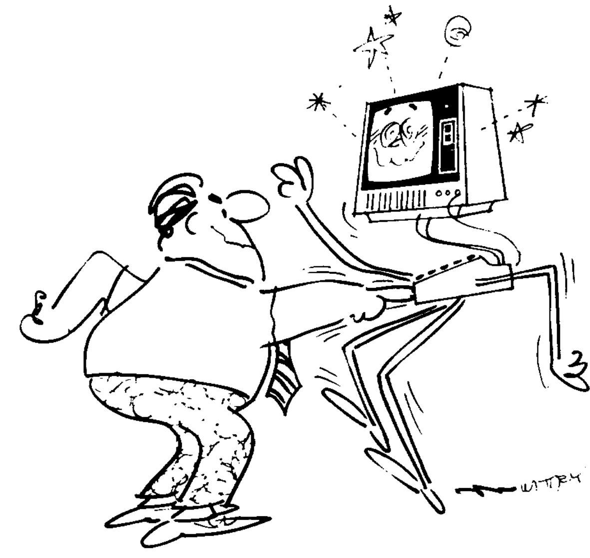
For example, the statement:

```
PRINT PEEK (15360)
```

prints out the contents of memory address 15360. (One of those special-purpose addresses.)

```
PRINT PEEK (20000)
```

This doesn't print the number 20000, it prints the contents of address 20000.



How to Write into a Byte of Memory

Before continuing, be forewarned: Some of the RAM addresses in your TRS-80 are used by the Computer to store important information. You might say these addresses help the Computer maintain its sanity. If you write new information into these addresses, your Computer is going to start behaving very strangely — or not behaving at all! In such cases, you'll have to turn the Computer off, then turn it on again a few seconds later. No harm to the Computer — but your program will be lost. (So before changing memory addresses directly, save your resident program on tape!)

In this chapter, we're going to write only to the "safe" addresses.

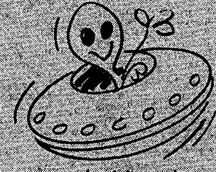
The BASIC statement that writes to memory is POKE (It's a well-chosen word, with its suggestion of intruding into someone else's business. Because when you POKE, you are bypassing BASIC. BASIC may not like that. So POKE gently and carefully!)

POKE requires two input values, an address and a value to be written to that address. For example:

```
POKE 15360, 191
```

writes 191 into address 15360

Notice anything happening on your display? Good. Read on!



UFO #27-1

BASIC allows you to read the contents of memory (ROM or RAM) or write new contents into RAM. PEEK returns the contents of an address; POKE stores a value in an address. The general formats for PEEK and POKE are:

PEEK(*address*)

address is numeric expression referring to a memory address. PEEK is a function; it cannot stand alone, but must be used in a statement.

POKE *address, value*

address is a numeric expression referring to a RAM location. If it is a ROM address, the POKE will have no effect.

value is a numeric expression between 0 and 255. This will become the new contents of the address.

To PEEK or POKE an address *above* 32767, convert the true address into a PEEK/POKE address with the formula:

PEEK / POKE *address = true address − 65536*

The resultant address will be a negative number between −32768 and −1

The last part of the UFO, about addressing locations above 32767, pertains only to readers with 32K or 48K RAM systems.

The reason for the conversion is: TRS-80 stores addresses in integer format. That means the possible ranges of addresses is from −32768 to 32767. So to address locations between 32768 and 65535, we simply start at −32768 and count up to −1.

The Visible RAM (An Educational Toy)

We have a good use for PEEK and POKE — one every TRS-80 programmer can probably appreciate, even if he or she has no interest in the technical aspects of a Computer. It has to do with the video display.

For each print cell on the display (the 1024 PRINT @ locations), there is a RAM location storing the current contents of that cell. Well, not the contents, but the TRSCII values for that content. For example, immediately after a CLS (when the entire display is "blank"), each of these RAM locations contains a value of 32, which is TRSCII for a blank space.

Remember TRSCII. That's TRS-80 Code for Information Exchange (a term derived from ASCII).

We call this portion of memory "video RAM." Its sole purpose is to store the contents of the display. When you PRINT something, BASIC changes video RAM. The display then changes automatically to match the contents of video RAM.

Video RAM occupies the addresses from 15360 to 16383. That's 1024 in all — one for each print cell, remember? The contents of cell 0 is stored in 15360, 1 in 15361, etc. In general, the contents of print cell N is stored in N + 15360.

Therefore, when you peek at an address in video RAM, you should get the TRSCII for the current contents of the corresponding print cell.

RUN this program for a demonstration:

```
NEW
10 CLS
20 PRINT @ 512, "PRESS ANY TEXT KEY";
30 A$=INKEY$: IF A$<" " THEN 30
40 PRINT @ 0, A$
50 PRINT "PEEK(15360)="; PEEK(15360)
60 GOTO 30
```

The program echoes your keyboard input (as long as you enter non-control characters) at print cell 0. Then it reads the contents of video RAM address 15360, which always contains the TRSCII for whatever is in cell 0.

PEEKing video RAM is very important, because it's the only way a program can read the current contents of the display (*You* can see the display, but your program can't).

The following program lets you type in one video "page" of information, then stores the page in an array when you press (↓). The only keys recognized are: text keys, (←), (ENTER) and (↓).

```
NEW
110 CLEAR 16*64+255   'ENOUGH FOR STRING ARRAY+SOME
120 DEFINT A-Z: DIM V$(15)   'V$() WILL STORE DSPLY CONTENTS
130 VM=15360   'START OF VIDEO RAM
140 PRINT CHR$(14);: CLS   'CURSOR ON AND CLEAR SCREEN
150 A$=INKEY$: IF A$="" THEN 150
160 IF A$>CHR$(31) THEN PRINT A$;: GOTO 150
170 IF A$=CHR$(8) OR A$=CHR$(13) THEN PRINT A$;: GOTO 150
180 IF A$=CHR$(10) THEN 200
190 GOTO 150   'IGNORE ALL OTHER KEYS
200 PRINT CHR$(15);   'CURSOR OFF BEFORE READ DSPLY
210 FOR LI=0 TO 15   'LINE NUMBERS
220 START=LI*64+VM   'RAM ADDRESSES, COLUMN 0
230 FOR CL=0 TO 63   'COLUMNS
240 AD=START+CL   'ACTUAL PEEK ADDRESS
250 V$(LI)=V$(LI)+CHR$(PEEK (AD))   'BUILD A LINE
260 NEXT CL, LI
270 CLS: PRINT "PAGE IS STORED . . ."
```

Run the program. Type in several lines — as you would on a typewriter, pressing (ENTER) to start a new line, or letting TRS-80 provide an "automatic carriage return" after column 63. Use the ⊖ to erase errors. When you are ready to save the display contents, press ⊕. There will be a short intermission while the program reads every address in video RAM into an array. (To display the contents of the array, see D.I.Y. #27-1.)

Program Notes:

Line 140 turns on the cursor for your convenience in entering characters.

Lines 160-180 check for valid key entries. If you have entered a text key, ⊖, or (ENTER), TRS-80 prints the character. If you entered a ⊕, TRS-80 drops out of the loop and starts reading video RAM. Any other key you press is ignored.

Line 210 lets LI ("line") range from zero to 15. LI will be the index for the current array element in V$(). It will also be used to compute the current video RAM address.

Line 220: START will count by 64's from 15360 to 16320. These are the RAM addresses of the column 0 cells on each display line.

Line 230: CL ("column") ranges from zero to 63, and will be used to pinpoint the exact memory address for PEEKing.

Line 240: AD is the current address in video RAM.

Line 250 builds up each element in V$(). Each element will contain 64 characters.

Program (Do It Yourself #27-1). Add lines to the end of the program to print out the array contents. Then run it again; type in new text; press ⊕; then watch the Computer save the screen, CLS, and re-print it.

Special challenge for look-ahead experts: Try to prevent scrolling when the last line is printed.

How to Write into Video RAM, or POKE, the Ultimate Weapon Against the Auto-Scroll

Remember the POKE example:

```
POKE 15360, 191
```

Knowing about the visible RAM, you can now understand why that white block appears at cell 0. Aha! An alternative to PRINTing!

Program (Do It Yourself #27-2). Write a program to POKE TRSCII codes 32 through 191 into video RAM addresses 15360 through 15519. Be sure not to POKE outside video RAM!

Don't get too carried away with POKE as a video output operation — PRINT is still the most efficient way to output lines to the display. POKE can output only one character at a time to the display, while a single PRINT statement could fill the display.

There's one major reason to POKE video RAM — to output to the display without moving the cursor. More to the point, POKE lets you output a character into the bottom right corner of the display without causing a scroll.

The address of that corner is 16383. So try this immediate line:

```
POKE 16383, 191: PRINT @ 0, "LOOK - NO SCROLL!!!"
```

Now make these changes and additions to the resident program (the one that reads the display contents into an array). You will then be able to fill the entire display with the array contents without producing a scroll:

```
280 INPUT "PRESS <ENTER> TO CONTINUE"; X
290 FOR LI=0 TO 14
300 PRINT V$(LI);
310 NEXT LI
320 PRINT LEFT$(V$(LI), 63);
330 POKE 16383, ASC(RIGHT$(V$(LI), 1))
340 GOTO 340
```

Run the complete program.

Repeating Keyboard Trick (Model I Only)

When you want to repeat a key, it takes two separate keystrokes, right? That prevents accidental key entries from occurring when you hold a key too long. But for some special applications, it's nice to have a repeating keyboard.

You can "install" this feature in your TRS-80 without opening the case (and voiding your pedigree). It's all done with — no, not mirrors — POKE.

Addresses 16437-16444 store "flags" telling whether the keys have been released since they were last pressed. If you POKE a value of zero into each of these addresses before doing an INKEY$, TRS-80 will think each key has been released and is active again. So you can fool your Computer into providing a repeat-key function!

Try this program:

```
NEW
400 CLS
410 A$=INKEY$: IF A$<" " THEN 410
420 PRINT A$;
430 FOR I=16437 TO 16444: POKE I,0: NEXT I
440 GOTO 410
```

Remember the test, A$< " "? It passes when A$ contains a *text* character. Null string or control characters will always fail this test.

Run it. Try holding down any text key (all others are ignored).

Can you think of ways this technique can improve your game programs? Sure you can!

Other Direct Memory Access Functions

VARPTR *Variable Pointer*. Returns the address of the bookkeeping area for a specified variable.

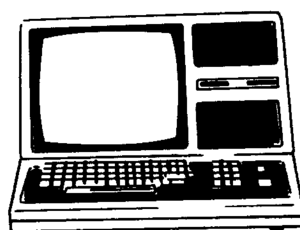
USR *User*. Transfers control from your BASIC program to a non-BASIC "machine-language" program.

INP and OUT *Input/Output* to any of TRS-80's 256 "ports".

But all these subjects require a considerably more technical level of understanding than we've tried to give in this book. Not that they're that difficult. . . To investigate this programming frontier, take a look at the books we referred to at the beginning of the chapter.

☑ Chapter Checkpoint #27

1. PEEK and POKE are:
   a. a couple of stand-up comics
   b. ways to examine and change contents of a memory location
   c. ways to increase screen resolution
2. Which of the following are correct formats?
   a. PEEK (value)
      POKE address, value
   b. PEEK (address)
      POKE address, value
   c. PEEK (value)
      POKE (address, value)
3. POKE 15360, 191 might be interpreted to read:
   a. Insert the character symbolized by ASCII code number 191 into memory slot 15360.
   b. Insert the character symbolized by ASCII code number 15360 into memory slot 191.
   c. The RAM address is malfunctioning.
4. (RAM) (ROM) is TRS-80's permanently fixed memory.
5. You now know three ways to output to the video display.
   They are ________, ________, and ________.



Chapter

Twenty Eight

Lion-Tamers, Fire-Eaters and Error-Handlers

By the time you've finished this chapter, you'll know what error-handlers have in common with the first two occupations. But first, what is an error-handler?

It's a routine that takes control when an error occurs during program execution. Normally, errors cause TRS-80 to end your program and display an error message. But with an error-handling routine, your program can tell the Computer, "Calm down, I can handle that!" when an error occurs.

The benefit of such a routine is that it lets your program try to take care of the error and keep going, avoiding at all costs the dread message:

```
?XY ERROR IN LLLL
```

Before giving you the keys to this powerful technique, we had better make it clear how and why we've managed without error-handlers thus far. It's all summed up in two words:

Preventive Programming

We've come a long way without error-handlers, simply by making sure errors won't occur. For example, RUN this program:

```
10 INPUT "INPUT A NON-ZERO NUMBER"; X
20 PRINT "THE RECIPROCAL OF "; X; "IS "; 1/X
30 PRINT: GOTO 10
```

What happens if you ignore the prompt's instructions and input the number zero? Blows your program, doesn't it? Preventive programming tells us to add the following line:

```
15 IF X = 0 THEN 10
```

Now the program won't let you input an undefined value.

UFO #28-1 Each time a program inputs a value from the keyboard, the program should make sure the value is in the expected range.

An Unpreventable Error

The above resident program isn't really error-proof. Can you think of an input that will still generate an error?

Typing in a non-numeric value won't do it; TRS-80 is smart enough to ask for a proper value and repeat the INPUT operation. What other input value might cause an error?

You are inputting to a numeric variable. Remember the range for numbers? It's approximately −1E38 to +1E38. So, try inputting a value like 1E50.

```
? OV ERROR IN 10
```

So there you have it — an error that can't be prevented. . . or can it?

Challenge (Do It Yourself #28-1). Think of a way in which even this last kind of error could be prevented. Don't accept this challenge unless you enjoy working on impractical answers to unlikely problems.

Check Appendix A for our answer.

Hint: Input the number as a string, and go from there.

Well, maybe there is a way to prevent this kind of error. But as sure as you figure it out, another big error will come along, requiring its own deluxe prevention program. At that rate, you'll never complete a practical program before running out of one of the three basic programming resources: time, RAM, and patience.

Introducing Dr. Sledge the Unsinkable

Now you're ready to appreciate Dr. Sledge, our first error-handler. He refuses to be undone by any overflow (or other) error that occurs.

Type in this new version of the program:

```
NEW
5 ON ERROR GOTO 100
10 INPUT "INPUT A NON-ZERO NUMBER"; X
15 IF X=0 THEN 10
20 PRINT "THE RECIPROCAL OF "; X; "IS "; 1/X
30 PRINT: GOTO 10
100 'DR SLEDGE, ERROR HANDLER
110 PRINT "*** VALUE OUT OF RANGE ***"
120 PRINT "NUMBER MUST BE BETWEEN -1E38 AND +1E38"
130 RESUME
```

Now run the program, and try entering any out-of-range number. Sledge has it under control, right?

UFO #28-2 Error-handling involves a pair of statements: one tells TRS-80 to transfer control to the start of your error-handler in case an error occurs. The other terminates your error-handling routine and tells the Computer where to continue. The statements are:

ON ERROR GOTO *line*

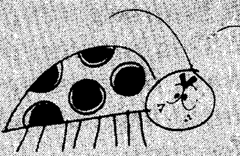
where *line* is the starting line of your error-handling routine.

When TRS-80 executes the ON ERROR GOTO statement, it does NOT transfer control to the routine; it simply enables (activates) the routine for use in case an error occurs.

RESUME *line* or RESUME NEXT

where *line* tells where execution resumes after exit from the error-handling routine. If *line* is omitted, execution resumes at the statement in which the error occurred. If RESUME NEXT is used, the program resumes execution at the next statement after the one causing the error.

This statement ends the error handling routine and returns control to the specified line in your program.



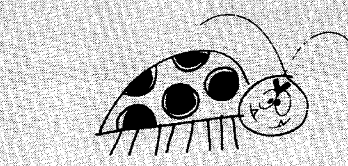
Sledge has a reputation for solving problems by obliterating them.

What better place to have a note on error but in a chapter about error-handling ("bug-handling" perhaps? Maybe this chapter should be entitled "Insecticides and the TRS-80" instead.)

Anyway, the most common error you'll run into with error-handling is:

```
?RW ERROR
```

RW stands for "RESUME without error". (You'll find out that every error-handler must end with a RESUME statement in one of its forms.)



Turn on the trace (TRON) and RUN it again. Notice the line sequence when you enter an out-of-range number:

<10> <100> <110> <120> <130> <10>

Error here. . . — So branch to here. . . — Execution resumes at point of error.

Program (Do It Yourself #28-2). Sledge handles errors by asking for another keyboard input. This is a very forgiving response (rather uncharacteristic of Sledge, by the way). Suppose we want to be heavy-handed about it: "You had your chance to input a decent value and you blew it. So we're going to provide our own value for you. . ."

Seriously, such approaches are sometimes justified.

After saving the existing program, modify it so that the error-handler sets X = 1E38, displays a message to that effect, and resumes execution at the next statement in the main program. (Sounds like a good application for RESUME NEXT. . .)

Check your answer against ours in Appendix A.

Having a routine like Dr. Sledge really puts your mind at ease, right? If anything goes wrong, the doctor will take care of it. . .

Yes, he will. But will his solution be appropriate to the problem?

Notice that the Doc has no way of knowing what kind of error brought him into power. Well, we're just assuming it was an OV ERROR. What happens if another kind of error occurred?

For example, change line 20 of the resident program to contain a deliberate typo:

```
20 PINT "THE RECIPROCAL OF"; X; "IS"; 1/X
```

Now RUN the program again.

That kind of error handling can give the practice a bad name: Dr. Sledge can't tell the difference between a syntax and an overflow error!

Pinpointing the Error

TRS-80 has two functions which are especially useful inside error-handlers. One identifies what type of error occurred; the other tells you where (in what line) the error occurred.

ERR/2 +1 (It's a Lot Easier Than E = M C²)

Every TRS-80 error condition has an error message. The message is what the Computer prints when it handles an error. For example;

```
?SN ERROR IN XXXX
```

Every error condition also has its own error code. The code for a syntax error is 1. See Appendix C for a complete list of error codes and messages.

TRS-80 always stores the code of the last error that has occurred since you typed RUN (the code = 0 if no error has occurred). If your error handler can look at this value, it will know what kind of error it's supposed to handle.

The function ERR/2 + 1 returns this current error code.

Back to our example. We're after overflow errors; looking at the appendix, we see that code 6 means overflow. So make these changes and additions to the program:

```
101 IF ERR/2+1=6 THEN 110
105 RESUME
```

Line 101 checks the current error code. If it is 6 (overflow), the program will GOTO line 110 and PRINT the overflow-related warning. Any other code drops control to line 105, which ENDS the error-handler and returns control to the error-causing statement.

Put a syntax error in line 20, and run the program. When you type in a good value for X, the Computer appears to stop — but READY never comes back. Press (BREAK), turn on the trace (TRON), then run it again, typing in a good value for X. Notice that the program is looping like this:

<20> <100> <101> <105> <20> <101> <105> <20>. . . .

Can you figure out what's wrong?

The syntax error activates Dr. Sledge. Then line 105 returns control to the point of the error. But the error occurs again! It happens over and over.

Why the /2 + 1? That's just a little eccentricity of TRS-80's. (Every machine needs a little eccentricity!)

Here's the complete program:

```
5 ON ERROR GOTO 100
10 INPUT "INPUT A NON-ZERO
   NUMBER"; X
15 IF X=0 THEN 10
20 PINT "THE RECIPROCAL OF ";
   X; "IS "; 1/X
30 PRINT: GOTO 10
100 'DR SLEDGE, ERROR HANDLER
101 IF ERR/2+1=6 THEN 110
105 RESUME
110 PRINT "*** VALUE OUT OF RANGE
   ***"
120 PRINT "NUMBER MUST BE
   BETWEEN -1E38 AND +1E38"
130 RESUME
```

Press (BREAK) then (ENTER) to get out of the program.

Turning Off the Error-Handler

This is one error Dr. Sledge cannot handle. We might as well let TRS-80 interrupt the program and take its standard approach to such errors (start you in the edit mode for that line). But how do we tell Dr. Sledge to give up and let TRS-80 take over? We use a special form of the statement that empowers Dr. Sledge in the first place:

```
ON ERROR GOTO 0
```

It doesn't matter whether your program contains a line 0; the 0 in this line simply disables the error-handler. After disabling Sledge, we can resume normal execution in the main program. Any error that occurs will now be handled by TRS-80. Dr. Sledge is powerless until we execute another ON ERROR GOTO 100 statement.

To try this out, change line 105 to

```
105 ON ERROR GOTO 0: RESUME
```

Now run the program. It now handles overflow errors, and lets TRS-80 handle all others.

Just Where Did the Error Occur, Ma'am?

The function ERL ("error line") returns a number telling you the program line where the error occurred. (If no error has occurred, or if the error occurred in the immediate mode, it returns a value of 65535.)

For example, fix the typo in line 20 (PRINT instead of PINT), then make these additional changes:

```
110 PRINT "ERROR OCCURRED IN LINE"; ERL
```

Now RUN the program.

In some applications, your error handler routine may need this information. For example, add these new lines and changes:

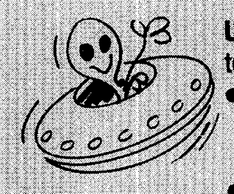
```
30 PRINT X; "TO THE 10TH POWER IS"; X ^ 10
40 PRINT: GOTO 10
110 IF ERL <> 30 THEN 120
112 PRINT "  TOO LARGE"
114 RESUME 10
```

Your program should look like this:

```
5 ON ERROR GOTO 100
10 INPUT "INPUT A NON-ZERO
    NUMBER"; X
15 IF X=0 THEN 10
20 PRINT "THE RECIPROCAL OF
     "; X; "IS "; 1/X
30 PRINT X; "TO THE 10TH POWER
    IS"; X ^ 10
40 PRINT: GOTO 10
100 'DR SLEDGE, ERROR HANDLER
101 IF ERR/2+1=6 THEN 110
105 RESUME
110 IF ERL <> 30 THEN 120
112 PRINT " TOO LARGE"
114 RESUME 10
120 PRINT "NUMBER MUST BE
     BETWEEN -1E38 AND +1E38"
130 RESUME
```

Now RUN the program. Enter small values like 5, 10, −5. So far, so good. Enter an out-of-range value like 5E50. That provokes the original portion of the error-handler. Now, enter a value like 50, which causes line 30 to compute 10 to the 50th power which is too large for the Computer to handle.

Line 110 determines whether the error occurred in the keyboard input line (20) or in the computation line (30), and takes appropriate action.



UFO #28-3 BASIC has two functions to make your error-handler "smarter":

- ERR/2 + 1
  Returns the error code of the most recent error.
  Returns 0 if no error has occurred.
- ERL
  Returns the number of the line where the most recent error occurred.
  Returns 65535 if no error has occurred during program execution.

Testing an Error-Handler

Suppose you have just installed a beautiful new error-handler in a program. There's absolutely no error that can blow your program. Every possible error invokes a different portion of your error-handler.

Now you want to see the error-handler at work. So go ahead, TRS-80, start erring! You wait patiently and no error occurs. Of course — your main program will perform perfectly when you want it to fall to pieces. You could rewrite the program, forcing the various errors to occur. But that takes time — and when you're done, you no longer have your original program!

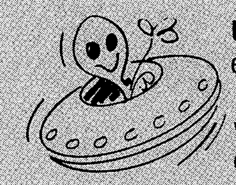
TRS-80 has the answer: a statement that simulates errors, so your error-handler can show its stuff.

Type in these lines in the **immediate** mode:

```
ON ERROR GOTO 0
ERROR 6
ERROR 2
ERROR 11
```

ON ERROR GOTO 0 is just in case your error-handler is still "on."

You should have received error messages for overflow, syntax, and division by zero, respectively.



UFO #28-4 To provoke one of BASIC's error messages, use the appropriate error code, like this:

ERROR *code*

where *code* is a numeric expression for a valid error code. A complete table of errors is in Appendix C.

To test an error-handler, you simply insert the ERROR statement at the point in your program where the specified error is likely to occur. The program will behave just as if that error had occurred in the line containing the ERROR statement.

Two Useful Error-Handlers

The program below shows two practical uses for error-handlers. Notice that you can have more than one error-handler in a single program, simply by executing an ON ERROR GOTO statement each time you are ready to disable one routine and enable another.

```
10 'STRING INPUT ROUTINE
20 CLEAR 25
30 ON ERROR GOTO 100
40 PRINT "FAVORITE TYPE OF FRUIT (25 CHARACTERS OR LESS)"
50 A$="": INPUT A$
60 GOTO 200
100 'ERROR HANDLER #1
110 IF ERR/2+1=14 AND ERL=50 THEN 130 'PINPOINT IT
```

```
120 ON ERROR GOTO 0: RESUME   'LET BASIC HANDLE OTHERS
130 PRINT "OUT OF STRING SPACE--TYPE IN 25 LETTERS MAX"
140 RESUME 40   'RESUME AT PRINT STATEMENT
200 'READ-DATA WITH ERROR HANDLING
210 ON ERROR GOTO 300
220 READ B$
230 IF A$=B$ THEN PRINT "WE HAVE SOME!!!": RESTORE: GOTO 40
240 GOTO 220
250 PRINT "WE DON'T HAVE ANY": RESTORE: GOTO 30
270 DATA APPLE, CHERRY, PEAR, PEACH, WATERMELON, ORANGE
280 'YOU CAN ADD YOUR OWN FAVORITES HERE
300 'ERROR HANDLER #2
310 IF ERR/2+1=4 AND ERL=220 THEN 330
320 ON ERROR GOTO 0: RESUME   'LET BASIC HANDLE OTHERS
330 PRINT "OUT OF DATA"
340 RESUME 250
```

Run the program. Type in the name of a type of fruit. The program will compare what you typed with its DATA list of fruits. If it finds a match before exhausting the list, it'll tell you so. If it exhausts the DATA list first, it'll handle the OD error, then continue.

> To provoke error handler #1, type in:
> `PICKLED RUMANIAN WATERMELON`
> as the name of your favorite fruit.

The program handles OS and OD errors — errors that are sometimes impossible or impractical to prevent.

Program Notes

Line 20: For illustration purposes, we want only 25 characters of string space available. (Be sure to put CLEAR statements ahead of ON ERROR GOTO, since CLEAR cancels the effect of a prior ON ERROR GOTO.)

Line 30: Enables error-handler #1.

Line 50: Clears out the current contents of A$ and inputs a new value. (Otherwise, when you press (ENTER) without a fruit name, A$ will retain its previous value.)

Line 110: If this test passes, we know an OS (out of string space) error occurred in line 50.

Line 120: If the test in line 110 fails, the error is one we didn't plan for. So we let TRS-80 handle it.

Line 210 enables error-handler #2 and automatically disables the one at line 100.

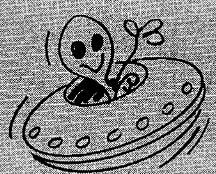
Lines 220-240 READ a value, compare it with A$, and continue doing this until a match is found or the DATA list is exhausted. We don't care how many items are in the data list, since the error-handler #2 will take care of OD (out of data) errors.

Line 300 finds out if the error is the one we expect. If it's not, line 310 lets TRS-80 handle it. If it is, line 330 lets execution resume at line 250.

Now what has error-handling got to do with lion-taming and fire-eating? If you haven't discovered the similarity yet. . . either

A. You haven't experimented with ON ERROR GOTO and RESUME yet, or
B. You are a very careful programmer and have avoided the dangers quite well (or maybe you've never tamed a lion, for that matter).

It's exciting to watch an error-handler work, provided he doesn't get careless; in which case the whole program may lose its head or go up in smoke (figuratively speaking, of course — contrary to rumor, an errant error-handler will not destroy your machine!).



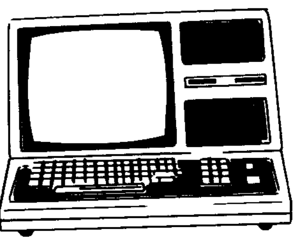
UFO #28-5

Tips for error-handlers

1. Don't use them when preventive programming will do the job better.
2. The first thing an error-handler should do is pinpoint the error (with ERR/2 + 1 and ERL).
3. If an unexpected error occurs, the error-handler should disable itself and let the Computer handle it (ON ERROR GOTO 0: RESUME).
4. Every error-handler must end with a RESUME statement. Use
   RESUME
   to re-try the error-causing statement.
   RESUME NEXT
   to skip the error-causing statement.
   RESUME *line*
   to resume execution at another point in the main program.
5. When an error-handler is no longer needed in your program, turn it off with ON ERROR GOTO 0.
6. Make sure TRS-80 doesn't execute an error-handler when no error has occurred. Put an END statement, a GOTO, or some other "road-block" just in front of the start of the error-handler.

☑ Chapter Checkpoint #28

1. Simply put, error-handlers allow you and your program to:
   a. ignore the prompt's instruction
   b. stroll on down the avenue
   c. continue when an error occurs during execution
2. (TRUE) (FALSE) ON ERROR GOTO 100 terminates your error-handling routine and tells TRS-80 where to continue.
3. A common error message related to improper error-handling is:
   a. NF
   b. OD
   c. RW
   d. JR
4. When you don't need an error-handler in your program anymore, you can turn it off by:
   a. telling a few bad jokes
   b. typing ON ERROR GOTO 0
   c. typing TURN OFF
   d. typing RESUME NEXT



Now You've Done It . . .

That's the end of our exploration of TRS-80 BASIC. We hope it's just the beginning of your own. In Appendix B, you'll find some additional programs for amusement and illustration. (And if you haven't checked our D.I.Y. answers in Appendix A, now's as good a time as any . . . There are several complete, stand-alone programs you may enjoy.)

Appendixes

- A/Suggested Answers to D.I.Y.'s and Checkpoints
- B/Sample Programs
- C/Error Messages
- D/Tables
- E/Display Worksheet

Appendix A

Do-It-Yourself and Checkpoint Answers

D.I.Y. # 1-1

Press the ⇧ key. Whether you get "↑" or "[" depends on which Model TRS-80 you've got. When we ask for a "↑," press ⇧.

D.I.Y. # 1-2

Press ⇧ or ENTER. ⇧ does not "end" the line; ENTER does. More on this in Chapter 2.

D.I.Y. # 1-3

The number zero is slashed "Ø" to help you tell it apart from the letter "O." We'll use slashed zeros when Computer information is displayed.

Chapter Checkpoint # 1

1. a 2. ">" 3. c 4. CLEAR
5. double-size 6. SHIFT ← 7. ←

D.I.Y. # 2-1

Try these:

```
PRINT 1, 2, 3, 4 ENTER
PRINT 1; 2; 3; 4 ENTER
PRINT 1, 2; 3, 4 ENTER
```

etc.

D.I.Y. # 2-2

```
PRINT 1/3 ENTER
```
etc.

D.I.Y. # 2-3

```
PRINT 83 + 11/21 - 33.7 ENTER
```

D.I.Y. # 2-4

```
PRINT 12345000 * 70010.101 ENTER
```

D.I.Y. # 2-5

```
PRINT "100 + 200 ="; 100 + 200 ENTER
```

Chapter Checkpoint # 2

1. NEW 2. error 3. ENTER 4. BREAK 5. PRINT

D.I.Y. # 3-1

```
10 CLS
20 PRINT "I LIKE STARS"
30 PRINT STRING$(50, "*")
```

D.I.Y. # 3-2

```
10 CLS
20 PRINT "1/2="; 1/2
30 PRINT "1/3="; 1/3
```
etc., up to
```
90 PRINT "1/9="; 1/9
```

Chapter Checkpoint # 3

1. instructions 2. LIST 3. line 4. RUN
5. line number 6. c

D.I.Y. # 4-1

```
10 CLS
20 PRINT "HI! I'M YOUR TRS-80 COMPUTER!!"
30 PRINT "WHAT'S YOUR NAME?"
40 INPUT N$
50 PRINT "HELLO "; N$
60 PRINT "HOW OLD ARE YOU TODAY?"
70 INPUT AG
80 D = AG * 365 + 365
90 PRINT N$; " WILL BE"; D; "DAYS OLD ON NEXT
   BIRTHDAY."
```

Chapter Checkpoint # 4

1. Statement: name-it, input/output, graphics, program flow, and declare-it.
2. c 3. True 4. alphanumeric information
5. Input/Output 6. Expression

D.I.Y. # 5-1

```
10 CLS
20 A = 10
30 PRINT 1/A
40 A = A + 1
50 IF A = 20 THEN END
60 GOTO 30
```

D.I.Y. # 5-2

Add one line:

```
65 IF N$ = "THAT'S ALL" THEN END
```

This is our first example of a string comparison. If N$ is exactly equal to "THAT'S ALL," the program ends.

D.I.Y. #5-3

```
10 CLS
20 A = 1
30 PRINT 1/A
40 A = A + 1
50 IF 1/A <= .0125 THEN END
60 GOTO 30
```

Chapter Checkpoint # 5

1. IF-THEN 2. It wouldn't be fair for us to say.
3. SHIFT @ 4. GOTO 5. comma and semi-colon
6. b 7. a

D.I.Y. # 6-1

```
100 INPUT "YES OR NO"; R$: IF R$ = "YES" THEN
    400
200 IF R$ = "NO" THEN 500
300 GOTO 100
400 PRINT "THAT'S BEING POSITIVE!": GOTO 100
500 PRINT "WHY SO NEGATIVE?": GOTO 100
```

Chapter Checkpoint # 6

1. c 2. colon 3. remark 4. a-AUTO, b-BREAK
5. DELETE 6. LIST

D.I.Y. # 7-1

You can do this one with the three edit commands you know. Start like this:

```
EDIT 10 ENTER
I PRINT: SPACE SHIFT ↑
```

That takes care of the first change. You take it from there...

Chapter Checkpoint # 7

1. I to insert, SHIFT ↑ to stop inserting.
2. syntax 3. D 4. b

Chapter Checkpoint # 8

1. 1 0 SPACE
2. H to hack. SHIFT ↑ to quit.
3. X to extend. SHIFT ↑ to quit.
4. c 5. ENTER, Q and E

D.I.Y. # 9-1

1 = 1E0   1234.56 = 1.23456E3   1000.1 = 1.0001E3
0.000123 = 0.123E-3   -300.0031 = -3.000031E2

1.1E1 = 11   8.7654321E8 + 876543210   1E6 = 1000000
3141593E-6 = 3.141593   0.123456E6 = 123456

D.I.Y. # 9-2

```
180 AVG% = TTL% / 25 + .5
```

D.I.Y. # 9-3

Accepted Answer:

```
10 A# = 1/3
20 B# = A# * 3
30 PRINT B#
```

Clever Sidestep:

Change line 10 to:

```
10 A# = 0.33333333333333333
```

(17 three's)

D.I.Y. # 9-4

```
10 CLS
20 N# = 1
30 N# = N# * 5
40 PRINT N#
50 GOTO 30
```

Chapter Checkpoint # 9

1. a. −3.8E7   b. 1.111111E6   2. a and b   3. b
4. double-precision

D.I.Y. #10-1

```
10 CLS
20 Y# = 1#: X# = 5#
30 PRINT X# ↑ Y#
40 Y# = Y# + 1#
50 GOTO 30
```

To prove a point, we've made everything double-precision. But as you'll find out, there's really no reason to use double-precision here.

D.I.Y. # 10-2

We start with a good guess that is too small.

```
10 GUESS=1: WANT=371.08: NEAR=1
20 VOL = GUESS * GUESS * GUESS
30 IF VOL < WANT THEN 50
40 PRINT "EACH SIDE ="; GUESS: END
50 GUESS = GUESS + NEAR
60 GOTO 20
```

This gives the smallest side measured to the nearest unit. For the measurements to the nearest .25 unit, change NEAR to .25 (line 10). For the nearest .001 unit, change NEAR to .001; etc.

D.I.Y. # 10-3

↑
* and /
+ and −

D.I.Y. # 10-4

```
PRINT 6 ^ 5 ^ (4 / (3 + 2 - 1)
```

D.I.Y. # 10-5

No answer needed.

D.I.Y. # 10-6

```
PRINT A%, B#, C$
```

Chapter Checkpoint # 10

1. a. `PRINT 73.2 ^ 2`   b. `PRINT 16 ^ 3`
   c. `PRINT 43 ^ 9`
2. a. `PRINT SQR (5358.24)`   b. `PRINT 4096 ^ (1/3)`
   c. `PRINT 5.025926E14 ^ (1/9)`
3. b.

D.I.Y. # 11-1

```
10 K = Ø: TTL = Ø
20 INPUT "ENTER THE NEXT NUMBER (999 TO
   QUIT)"; A
30 IF A = 999 THEN 6Ø
40 TTL = TTL + A: K = K + 1
50 GOTO 2Ø
60 IF K = Ø THEN END
70 PRINT "TOTAL ="; TTL, "AVERAGE ="; TTL/K
```

D.I.Y. # 11-2

```
10 N=Ø
20 N = N+1
30 IF N*N*N >= 111Ø THEN 5Ø
40 GOTO 2Ø
50 PRINT N-1; "IS THE LARGEST N SUCH THAT"
60 PRINT "N CUBED IS LESS THAN 111Ø."
```

Can be done with a simple statement:

```
PRINT INT (111Ø ^ (1/3))
```

D.I.Y. # 11-3

```
10 A=Ø: INC=2: LIM=256
20 PRINT A
30 A=A+INC
40 IF A>LIM THEN END
50 GOTO 2Ø
```

D.I.Y. # 11-4

```
10 A=Ø: INC=.1: LIM=1Ø
20 PRINT A
30 A=A+INC
40 IF A>LIM THEN END
50 GOTO 2Ø
```

But with this program, we see round-off errors at 7.8, etc., and we never get to print out the limit value, 10.

For "perfection", change line 30:

```
30 A=INT ((A+INC)*1Ø+.5)/1Ø
```

D.I.Y. # 11-5

```
10 A=Ø: INC=1: LIM=59
20 PRINT A
30 A=A+INC
40 IF A>LIM THEN 1Ø
50 GOTO 2Ø
```

D.I.Y. # 11-6

```
10 FOR I = Ø TO 256 STEP 2
20 PRINT I
30 NEXT I
```

D.I.Y. # 11-7

```
10 FOR I=Ø TO 1Ø STEP .1
20 PRINT I
30 NEXT I
```

This has the round-off problem. We can solve it by changing line 20:

```
20 PRINT INT(I*1Ø+.5)/1Ø
```

But we're still not reaching the limit value, 10. So make these changes and additions:

```
15 I=INT(I*10+.5)/1Ø
20 PRINT I
```

D.I.Y. # 11-8

```
5 FOR MN =Ø TO 59
10 FOR SC=Ø TO 59
20 PRINT MN; ":"; SC
25 FOR K=1 TO 33Ø
27 NEXT K
30 NEXT SC
40 NEXT MN
50 GOTO 5
```

D.I.Y. # 11-9

Make these changes to the "clock":

```
40 FOR HR=HR TO 23
120 NEXT HR: HR=Ø
```

D.I.Y. # 11-10

```
10 FOR I=1 TO 2
20 I=1
30 PRINT "I LIKE TO RUN",
40 NEXT I
```

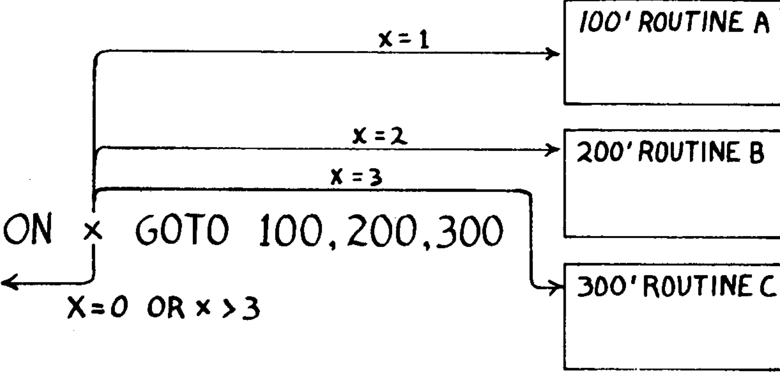
D.I.Y. # 11-11

The program is in the main text.

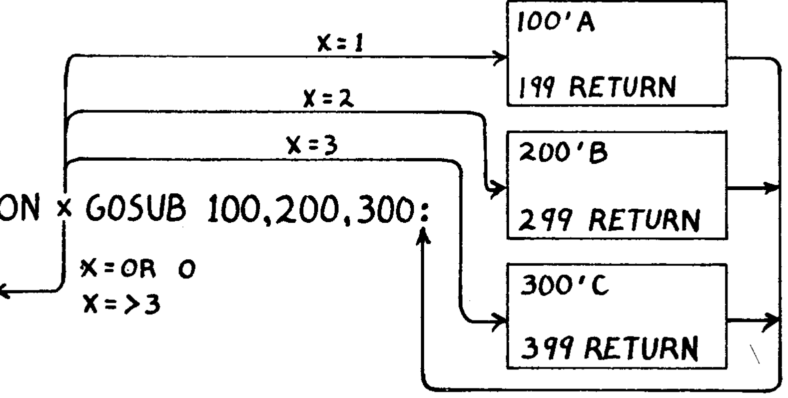
Chapter Checkpoint # 11

1. a-2 b-3 c-1 2. TRON 3. endless

D.I.Y. # 12-1



D.I.Y. # 12-2



D.I.Y. # 12-3

In D.I.Y. # 12-2, substitute C for X; subroutine A is square root, sub-routine B is cube root, subroutine C is exponentiation.

Chapter Checkpoint #12

1. ELSE 2. b 3. GOSUB, RETURN

D.I.Y. # 13-1

```
120 IF RICK = YES AND ELLA =NO THEN 500
```

D.I.Y. # 13-2

```
130 IF SUE = YES AND ELLA = YES THEN 500
```

Chapter Checkpoint # 13

1. b, c, e 2. a-2 b-3 c-1

D.I.Y. # 14-1

The answer is in the main text.

D.I.Y. # 14-2

```
304 R$ = "X MUST BE POSITIVE"
306 IF X <= 0 THEN GOSUB 900: GOTO 300
```

Chapter Checkpoint # 14

1. c
2. Sine, log, and exponentiation are available. Cosine, tangent, exponential, and other built-in math functions could be added.
3. It is accurate only to seven digits.

D.I.Y. # 15-1

With Bob's no-array method, you would need 20 separate statements for inputting data, and 20 separate statements for printing out each item.

Change Ray's program like this:

```
10 DIM G(20)
DELETE 20
30 FOR I = 1 TO 20
DELETE 60
70 INPUT "WHICH ITEM DO YOU WANT TO RE-ORDER
   (1-20)"; I
75 IF I<1 OR I>20 THEN 70
```

D.I.Y. # 15-2

Milk quantity = *G(m* * *3* – *2)* = *G(m* * *3* – *3* + *item #)*
(Milk is item #1.)

D.I.Y. # 15-3

Eggs quantity = *G(m* * *3* – *1)* = *G(m* * *3* – *3* + *item #)*
Wax quantity = *G(m* * *3)* = *G(m* * *3* – *3* + *3)*
(Eggs = item #2, Wax = item #3.)

```
10 DIM G(36)
20 FOR M=1 TO 12
30 FOR I=1 TO 3
40 PRINT "FOR MONTH #"; M;
50 PRINT "ENTER THE QUANTITY SOLD OF ITEM #"; I
60 INPUT G(M*3-(3-I))
70 NEXT I,M
80 CLS
90 INPUT "ENTER THE MONTH # AND ITEM #"; M,I
95 IF M<1 OR M>12 OR I<1 OR I>3 THEN 90
100 PRINT "ORDER THIS MUCH:"; G(M*3-(3-I))
110 GOTO 90
```

D.I.Y. # 15-4

L stores the month number of the worst month.
LT stores the total sales of the worst month.
GT stores the grand total of all sales.

For the best-selling item in the best month, pick the largest value printed by this:

```
PRINT G(H, 1), G(H, 2), G(H, 3)
```

For the worst-selling item in the worst month, pick the smallest value printed by this:

```
PRINT G(L, 1), G(L, 2), G(L, 3)
```

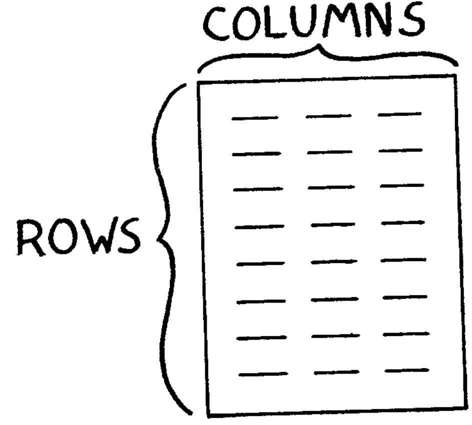
D.I.Y. #15-5

We computed the totals simultaneously:

```
220 T1 = 0: T2 = 0: T3 = 0
230 FOR M = 1 TO 12
240 T1 = T1 + G(M, 1)
250 T2 = T2 + G(M, 2)
260 T3 = T3 + G(M, 3)
270 NEXT M
280 PRINT "TOTAL MILK SALES = "; T1
290 PRINT "TOTAL EGGS SALES = "; T2
300 PRINT "TOTAL WAX SALES = "; T3
310 GOTO 51
```

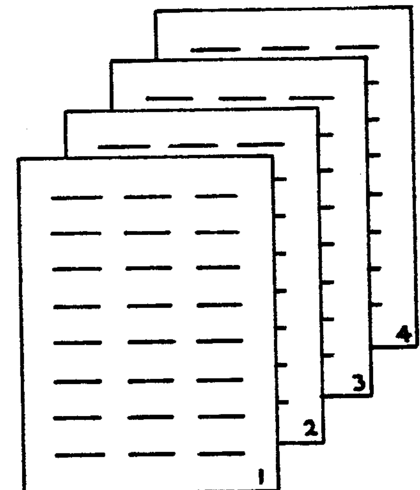
Chapter Checkpoint #15

1. b, c
2. A table, like this:



D.I.Y. # 16-1

1. Sales at store #4, in December, of item 3 (wax).
2. 4 * 12 * 3 = 144
3. Three-dimensional arrays may be visualized as stacks of pages, each page containing a table:



D.I.Y. # 16-2

```
51 PRINT "PRESS <1> FOR RE-ORDER, <2> FOR SALES
   ANALYSIS"
52 INPUT A
53 ON A GOTO 54,100
80 GOTO 51
100 FOR S = 1 TO 4
102 PRINT "SALES ANALYSIS FOR STORE #"; S
110 H = 1: L = 1: GT = 0
115 HT = G(S,I,1) + G(S,I,2) + G(S,I,3): LT =
    HT
120 FOR M = 1 TO 12
130 MT =0
140 FOR I = 1 TO 3: MT = MT + G(S,M,I): NEXT I
150 IF MT > HT THEN H = M: HT = MT
160 IF MT < LT THEN L = M: LT = MT
170 GT = GT + MT
180 NEXT M
190 PRINT "THE GRAND TOTAL OF ITEMS SOLD IS";
    GT
200 PRINT "MONTH"; H; "WAS BEST, WITH TOTAL
    SALES OF"; HT
210 PRINT "MONTH"; L; "WAS WORST, WITH TOTAL
    SALES OF"; LT
215 NEXT S
220 GOTO 51
```

D.I.Y. # 16-3

```
5 CLEAR 640    'ENOUGH FOR TEN 64-CHARACTER
   NAMES
10 DIM TX$(10)
20 FOR N=1 TO 10
30 PRINT " ENTER NAME #"; N
40 INPUT TX$(N)
50 NEXT N
60 PRINT: PRINT "HERE THEY ARE AGAIN:"
70 FOR N = 1 TO 10
80 PRINT TX$(N)
90 NEXT N
100 GOTO 100
```

D.I.Y. # 16-4

```
FOR I = 1 TO 5: ? NN$(I): NEXT
FOR I = 1 TO 5: ? VB$(I): NEXT
FOR I = 1 TO 4: ? FT$(I): NEXT
```

D.I.Y. # 16-5

```
10 CLEAR 500    'ENOUGH FOR 25 20-LETTER WORDS
20 READ NN, NJ, NV, NI
30 DATA 10, 5, 5, 5
40 DIM N$(NN), AJ$(NJ), AV$(NV), VB$(NI)
50 FOR I = 1 TO NN: READ N$(I): NEXT I
55 DATA RIVER, MOUNTAIN, SKY, WATER, WIND
56 DATA RAIN, VALLEY, FOREST, DESERT, SEA
60 FOR I = 1 TO NJ: READ AJ$(I): NEXT I
65 DATA BLUE, HOT, WISTFUL, UNSURE, STILL
70 FOR I = 1 TO NV: READ AV$(I): NEXT I
```

```
75 DATA SOFTLY, QUICKLY, BAREL Y, NEVER, AL-
   WAYS
80 FOR I = 1 TO NI: READ VB$(I): NEXT I
85 DATA WHISPER, REMAIN, GROW, GO AWAY, EN-
   DURE
90 W1 = RND(NN): W2 = RND(NJ): W3 = RND(NN)
100 W4 = RND(NN): W5 = RND(NV): W6 = RND(NJ)
110 W7 = RND(NN): W8 = RND(NI)
120 PRINT N$(W1);" IN THE ";AJ$(W2);"  ";N$(W3)
130 PRINT N$ (W4);"  "; AV$(W5);"  "; AJ$(W6)
140 PRINT "WHEN WILL THE "; N$(W7); " ";
    VB$(W8)
150 PRINT: PRINT
160 FOR T = 1 TO 330: NEXT T
170 GOTO 90
```

D.I.Y. # 16-6

1.
```
CLEAR
DIM I%(1)
I%(1)=1.5
? I%(1)
I%(1)=-1.5
? I%(1)
```

2.
```
CLEAR
DIM I#(1)
I#(1)=1.234567891
? I#(1)
```

3.
```
CLEAR
DIM I! (1)
I!(1)=1.234567891
? I!(1)
```

Chapter Checkpoint # 16

1. DIM G(1, 2, 6) 2. How much RAM is available.

D.I.Y. # 17-1

Add this line:

```
126 IF LEN(P$) = 4 THEN P$ = "REJECT"
```

D.I.Y. # 17-2

Notice the "instring" subroutine beginning at line 1000. It's going to come in handy many times in this and later chapters.

Given S1$ (the source text), S2$ (the target text), and SP (the start position), it searches for the first occurrence of S2$ in S1$. The search begins at SP. Upon return from the subroutine, SF contains the position where the string starts. If the string is not found, SF=0.

```
10 CLEAR 300
20 PRINT
40 INPUT "ENTER THE TEXT TO BE CENSORED": S1$
50 S2$ = "SCRUBDEK": SP = 1
60 GOSUB 1000 'FIND OCCURRENCE
70 IF SF = 0 THEN 100
80 PRINT "THIS TEXT IS NOT ACCEPTABLE, WATCH
   IT!"
90 GOTO 20
100 PRINT "THIS TEXT IS ACCEPTABLE...
    CONTINUE":GOTO 20
1000  'INSTRING SUBROUTINE
1010  'INPUT VALUES: S1$ = TO BE SEARCHED S2$ =
      STRING TO FIND
1020  '    SP = START POSITION FOR SEARCH
1030  'INTERNAL VALUES: S3 = LEN(S1$)    S4 =
      LEN(S2$)
1040  '    SL = LAST POSITION TO CHECK
1042  '    SI = COUNTER
1050  'OUTPUT VALUE: SF = POSITION WHERE S2$
      WAS FOUND IN S1$
1060  '    IF NOT FOUND, SF = 0
1070  '
1080  SF = 0: S3 = LEN(S1$): S4 = LEN(S2$)
1090  IF S4 > S3 THEN PRINT " PARAMETER
      ERROR": STOP
1100  SL = S3 - S4 + 1
1110  FOR SI = SP TO SL
1120  IF MID$(S1$, SI, S4) = S2$ THEN SF = SI: SI
      = SL
1130  NEXT SI
1140  RETURN
```

Chapter Checkpoint # 17

1. a 2. "and a right portion." 3. 50, CLEAR

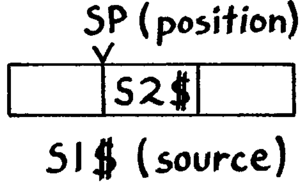
D.I.Y. # 18-1

Make these changes and additions to the program from D.I.Y. # 17-2.

```
80 S2$ = " NOBODY ": REM 8 CHARACTERS LONG
81 PRINT "CENSOR-PROGRAM ACTIVATED..."
82 GOSUB 2000
85 PRINT "HERE IS THE CENSORED MESSAGE:"
87 PRINT S1$
2000 'MIDSTRING REPLACEMENT SUBROUTINE
2010 'INPUT VALUES:    S1$ = ORIGINAL STRING
2020 '    S2$ = REPLACEMENT SUBSTRING
2030 '    SF = START POSITION FOR REPLACEMENT
2040 'INTERNAL VALUES:    S3 = LEN(S1$)    S4
       = LEN(S2$)
2045 '    SR = NO. OF TRAILING CHARACTERS
2050 'OUTPUT VALUE: S1$ = STRING WITH RE-
       PLACEMENT MADE
2060 '
2070 S3 = LEN(S1$): S4 = LEN(S2$): SR = S3 - SF -
       S4+1
2080 IF SF < 1 OR SR < 0 THEN RETURN
2090 S1$ = LEFT$(S1$, SF-1) + S2$ +
       RIGHT$(S1$, SR)
2100 RETURN
```

Notice the "midstring" subroutine beginning at line 2000. This is our string modification tool.

Given S1$ (the source string), S2$ (the replacement substring), and SF (the position in S1$ where the replacement begins, this subroutine changes S1$. This picture shows the result:



Chapter Checkpoint # 18

1. "+" (concatenation)
2. string, integer, single-precision, double-precision

D.I.Y. # 19-1

1. The DATA statement contains the characters to be treated as word-delimiters. For example, "," marks the end of a word just as surely as does a space " ".
2. Type in a line using "/" instead of spaces between words.
   Here's a more rigorous delimiter list:

```
110 DATA " ",",",".","/","?","<",">",";",
    "+", "!", "#", "$", "%", "&", "'", "(",
    ")", ":", "*", "-", "=", "@", 0
```

3. K$( ) stores the position of each delimiter in this text. For n equal 1 to the length of the text:
   K$($n$) = The nth character in the string if it is a delimiter. K$($n$) = "" if the nth character is not a delimiter.

   To see an example of this, add one line to the program:

```
145 STOP
```

   and run it. When the program stops, type in the immediate mode:

```
FOR I = 1 TO LEN(S1$): ? K$(I);: NEXT
```

   You'll see a list of the delimiters that were found.

D.I.Y. # 19-2

A$ contains the list of words. The Computer goes through the list, comparing each word with its successor. If they are out of order, they are swapped. The alphabetically "lighter" elements gradually float to the top of the list. When the Computer goes through the entire list without making a single swap, the list is sorted.

Because of the floating process, this method is known as a "bubble sort."

```
100 CLEAR 300
110 DIM A$(15)
120 FOR I = 1 TO 15
130 PRINT "TYPE IN WORD #"; I;
140 INPUT A$(I)
150 NEXT I
160 T = 0     'WHEN T = 1, A SWAP HAS OCCURRED
170 FOR I = 1 TO 14
180 IF A$(I) <= A$(I+1) THEN 200
185 'THE NEXT LINE PERFORMS THE SWAP AND SETS T
    = 1
190 X$ = A$(I): A$(I) = A$(I+1) = X$: T=1
200 NEXT I
210 IF T=1 THEN 160    'REPEAT UNTIL NO SWAPS
    ARE MADE
220 PRINT "HERE IS THE ALPHABETIZED LIST"
```

```
230 FOR I = 1 TO 15
240 PRINT A$(I)
250 NEXT I
260 INPUT "PRESS (ENTER) TO END"; X
```

D.I.Y. # 19-3

```
10 CLEAR 1000
15 DEFINT A-Z
20 DIM K$(255)     'K$() STORES DELIMITER IM-
     AGE
25 DIM K(256)     'K() STORES DELIMITER POSI-
     TIONS
27 DIM N(256)    'N() STORES PERIOD POSITIONS
30 PRINT "TYPE IN THE TEXT, STARTING WITH A
     QUOTE. TYPE IN UP"
35 PRINT "TO 240 CHARACTERS. YOU MAY TYPE MORE
     THAN ONE SENTENCE."
37 INPUT S1$
38 IF S1$="" THEN 37
39 NW=0: LW=0: NS=0: LS=0     '# AND LENGTH OF
     WORDS AND SENTENCES
40 SS = 15: SW = 4     'SUGGESTED WORD AND SENT-
     ENCE LENGTH
100 FOR I=1 TO 255: K$(I) = "": NEXT I     'CLEAR
     OUT K$( )
110 DATA " " , "," , ";" , "." , "!" , "?" , "0"
115 READ S2$: IF S2$ = "0" THEN 190     'HAVE
     CHECKED FOR DELIMS.
120 SP = 1
125 GOSUB 1000
130 IF SF = 0 THEN 115     'GET NEXT DELIMITER
135 K$(SF) = S2$
140 SP = SF + S4: IF SP > S3 THEN 115     'END OF
     TEXT
145 GOTO 125
190 KT = 0: K(0) = 0     'KT COUNTS DELIMS, K(0)
     IS ALWAYS ZERO
191 NP = 0: N(0) = 0     'NP COUNTS PERIODS. N(0)
     IS ALWAYS ZERO
192 FOR I=1 TO 255
193 IF K$(I)="." OR K$(I)="!" OR K$(I)="?"
     THEN NP = NP + 1: N(NP)=I
194 IF K$(I) <> "" THEN KT = KT + 1: K(KT) = I
196 NEXT I
200 K(KT + 1) = S3 + 1     'END OF TEXT IS A DE-
     LIMITER, TOO
205 N(NP + 1) = S3 + 1     'END OF TEXT IS A DE-
     LIMITER, TOO
210 FOR I = 1 TO KT = 1
220 IF K(I) - K(I-1) > 1 THEN NW = NW + 1: LW = LW +
     K(I) - K(I-1)
290 NEXT I
292 FOR I = 1 TO NP + 1
294 IF N(I) - N(I-1) > 1 THEN NS = NS + 1: LS = LS +
     N(I) - N(I-1)
296 NEXT I
297 VS = LS/NS: VW = LW/NW
300 PRINT "YOU TYPED"; NS; "SENTENCE(S)
     CONTAINING"; NW; "WORD(S)."
310 PRINT "AVERAGE SENTENCE LENGTH:"; VS;
     "CHARACTER(S)"
320 PRINT "AVERAGE WORD LENGTH:"; VW;
     "CHARACTER(S)"
330 IF VS > SS THEN PRINT "USE SHORTER
     SENTENCES!"
340 IF VW > SW THEN PRINT "USE SHORTER WORDS!"
350 IF VS <= SS AND VW <= SW THEN PRINT
     "EXCELLENT!"
420 RESTORE: GOTO 30
1000 'INSTRING SUBROUTINE
1010 'INPUT VALUES: S1$ = TO BE SEARCHED
             S2$ = STRING TO FIND
1020 '     SP = START POSITION FOR SEARCH
1030 'INTERNAL VALUES: S3 = LEN(S1$)     S4 =
     LEN(S2$)
1040 '     SL = LAST POSITION TO CHECK
1042 '     SI = COUNTER
1050 'OUTPUT VALUE: SF = POSITION WHERE S2$
     WAS FOUND IN S1$
1060 '     IF NOT FOUND, SF = 0
1070 '
1080 SF = 0: S3 = LEN(S1$): S4 = LEN(S2$)
1090 IF S4 > S3 - SP + 1 THEN PRINT " PARAMETER
     ERROR": STOP
```

```
1100 SL = S3 - S4 + 1
1110 FOR SI = SP TO SL
1120 IF MID$(S1$, SI, S4) = S2$ THEN SF = SI: SI
     = SL
1130 NEXT SI
1140 RETURN
2000 'MIDSTRING REPLACEMENT SUBROUTINE
2010 'INPUT VALUES:    S1$ = ORIGINAL STRING
2020 '    S2$ = REPLACEMENT SUBSTRING
2030 '    SF = START POSITION FOR REPLACEMENT
2040 'INTERNAL VALUES:     S3 = LEN(S1$)
        S4 = LEN(S2$)
2045 '    SR = NO. OF TRAILING CHARACTERS
2050 'OUTPUT VALUE:    S1$ = STRING WITH RE-
     PLACEMENT MADE
2060 '
2070 S3 = LEN(S1$): S4 = LEN(S2$): SR = S3 - SF -
     S4 + 1
2080 IF SF < 1 OR SR < 0 THEN RETURN
2090 S1$  =  LEFT$(S1$,  SF-1)  +  S2$  +
     RIGHT$(S1$, SR)
2100 RETURN
```

D.I.Y. # 19-4

```
10 CLEAR 20 * 64 'ENOUGH FOR 2 COPIES OF 10 64-
   CHAR. ENTRIES
20 DIM TX$(9)  '10 ENTRIES
30 CLS
40 FOR EN=1 TO 10
50 PRINT:PRINT "TYPE IN A QUOTE, THEN A LINE OF
   TEXT."
60 INPUT TX$(EN-1)
70 PRINT "THAT WAS ENTRY #"; EN; ". YOU HAVE";
   10-EN; "ENTRIES LEFT."
80 NEXT EN
90 FOR EN=1 TO 10
100 PRINT:PRINT "HERE IS ENTRY #"; EN; "."
110 PRINT TX$(EN-1)
120 INPUT "OK (Y/N)"; R$: IF R$<>"N" THEN 220
130 S1$ = TX$(EN-1)
140 INPUT "TYPE IN THE POSITION WHERE THE
    CHANGE STARTS"; SF
150 INPUT "TYPE IN THE NUMBER OF CHARACTERS TO
    CHANGE"; SC
160 PRINT "TYPE A QUOTE, THEN THE TEXT:"
170 INPUT S2$
180 GOSUB 2000
190 TX$(EN-1) = S1$
200 PRINT "HERE IS THE CORRECTED LINE.":
    PRINT S1$
210 GOTO 120
220 NEXT EN
230 PRINT "HERE IS THE ENTIRE TEXT."
240 FOR EN=1 TO 10: PRINT TX$(EN-1): NEXT
250 END
2000 'MIDSTRING REPLACEMENT SUBROUTINE
2010 'INPUT VALUES:    S1$ = ORIGINAL STRING
2020 '    SC = # OF CHARACTERS TO REPLACE
2030 '    S2$ = REPLACEMENT SUBSTRING
2040 '    SF = START POSITION FOR REPLACEMENT
2050 'INTERNAL VALUES:    S3 = LEN(S1$)    S4
     = LEN(S2$)
2060 '    SR = NO. OF TRAILING CHARACTERS
2070 'OUTPUT VALUE:    S1$ = STRING WITH RE-
     PLACEMENT MADE
2080 '
2090 S3 = LEN(S1$): S4 = LEN(S2$): SR = S3 - SF -
     SC + 1
2100 IF SF < 1 OR SR < 0 THEN RETURN
2110 S1$  =  LEFT$(S1$,  SF-1)  +  S2$  +
     RIGHT$(S1$,SR)
2120 RETURN
```

Chapter Checkpoint # 19

1. b 2. a-2 b. 1 3. 77 + 32 = 99 4. alphabetical

D.I.Y. # 20-1

A. 64 columns per line
B. Whenever a character is printed in column 63, the cursor drops down to the beginning of the next line (this is called a "carriage return"). When we leave off the semi-colon at the end of PRINT, another carriage return is done, leaving a blank line on the display. Putting a semi-colon at the end of PRINT suppresses the extra carriage return, eliminating the blank line.
C. 16 lines. The last line must be printed without a carriage return or else the display will scroll. So we have a semi-colon on this one line.
D. Every number is printed with a leading and trailing space. For negative numbers, the leading space is filled by the minus sign. Leading zeros to the left of the decimal point are not printed. Trailing zeros to the right of the decimal point are not printed.

The display should look like this:

```
Ø....5....Ø....5....Ø....5....Ø....5
 1  1Ø  5.3 -1 -1Ø -5.3 -5 -5.3  .1
```

E. Here's our demo program:

```
10 CLS
30 PRINT "Ø....5....Ø....5....Ø....5....
   Ø....5"
35 READ A$: IF A$="END" THEN END
40 PRINT A$;
50 GOTO 35
60 DATA HELLO, TWO WORDS, 1, -1, END
```

It produces this output:

```
Ø....5....Ø....5....Ø....5....Ø....5
HELLOTWO WORDS1-1
```

Strings are printed with no added spaces.
F. See D.
G. and H. The "cursor" is the current print position. The normal PRINT statement always resumes printing where the cursor was last seen.

D.I.Y. # 20-2

Try this version of line 50:

```
50 PRINT X; "      ";
```

D.I.Y. # 20-3

Line 90 contains two TAB instructions, TAB(MN) and TAB(MX). If MN > MX, the second TAB won't be done, because you can't TAB backwards. So we find out which value is smaller and always do that tab first.

Chapter Checkpoint # 20

1. a 2. comma and semi-colon

D.I.Y. #21-1

Change lines 20 and 60 of the average rainfall program:

```
20 PRINT TAB(3); "A"; TAB(13); "B"; TAB(23);
   "C"; TAB(33); "D"; TAB(43); "E"
60 PRINT TAB(J); USING "###.#"; X;    'NOTE
   TRAILING SEMI-COLON
```

D.I.Y. # 21-2

```
90 PRINT USING "$$#,###.##"; M
```

D.I.Y. #21-3

Add this line:

```
27 INPUT "WHAT DAY OF THE MONTH WERE YOU BORN
   ON"; D
28 IF D = INT(D/2) * 2 THEN 30 'EVEN NUMBER?
29 PRINT "SORRY-ONLY EVEN # PEOPLE GET PRO-
   CESSED TODAY": GOTO 100
```

Chapter Checkpoint # 21

1. c 2. d 3. a

D.I.Y. # 22-1

Change line 30:

```
30 FOR I = 1 TO 7 * 64 + 31
```

D.I.Y. # 22-2

Change line 30

```
30 FOR I = 1 TO 15 * 64 + 31
```

D.I.Y. # 22-3

Change line 30

```
30 FOR I = 1 TO 15 * 64 + 62
```

D.I.Y. # 22-4

Change line 30

```
30 FOR I = 1 TO 15 * 64 + 63
```

D.I.Y. # 22-5

16 * 64 = 1024

Chapter Checkpoint #22

1. On: ? CHR$(14)
   Off: ? CHR$(15)
2. a 3. anywhere 4. c 5. True

D.I.Y. #23-1

```
10 CLEAR 100
20 B$=STRING$(6,191)    'SOLID BAR
30 NL$=CHR$(26)+STRING$(6,24) 'DOWN AND
   BACK
40 SQ$=B$+NL$+B$   'CHECKER SQUARE
50 CLS
60 THIS=-1
70 FOR ROW=0 TO 7
80 IF THIS=-1 THEN BASE = 8 ELSE BASE = 14
90 FOR BLCK=0 TO 3
100 AT=ROW*128+BLCK*12+BASE
110 PRINT@AT,SQ$;
120 NEXT BLCK: THIS=-THIS: NEXT ROW
130 GOTO 130
```

Chapter Checkpoint #23

1. c 2. a 3. u/lc
4. Control, CHR$(14), turns the cursor on.

Chapter Checkpoint #24

1. SET, RESET 2. X and Y coordinates 3. c

D.I.Y. #25-1

Make these changes and additions:

```
1060 IF ABS(K) < 100 THEN K = K + T: GOTO 1023
1062 IF SGN(K) = SGN(T) THEN K = SGN(K) * 100:
     GOTO 1023
1070 K = K + T: GOTO 1023
```

D.I.Y. # 25-2

```
1224 IF K$ = CHR$(10) THEN PRINT B$;
     CHR$(26);: GOTO 1200
```

Chapter Checkpoint #25

1. gets or returns
2. (BREAK) (interrupts program) and (SHIFT) @ (pauses program).
3. VAL

D.I.Y. #26-1

Pair #1 The number of variables doesn't match.
Pair #5 There will be more than 248 characters output.

Chapter Checkpoint #26

1. b 2. #-1 3. b 4. d

D.I.Y. #27-1

```
280 CLS
290 FOR LI = 0 TO 15
300   PRINT V$(LI);
310 NEXT LI
```

See the text for the answers to the challenge.

D.I.Y. #27-2

```
10 CLS
20 VM=15360
30 FOR CD=0 TO 159
40 POKE VM+CD,CD+32
50 NEXT CD
60 GOTO 60
```

Chapter Checkpoint #27

1. b 2. b 3. a 4. ROM 5. PRINT, SET / RESET, POKE

D.I.Y. #28-1

Run the program listed below. Try inputting these values:

1E37 (Okay)
100E40 (Reject!)
−350E37 (Reject!)
Any number with more than 38 digits (Reject!)
1D50 (Reject!)
31D−39 (Reject!)

For any x, x is rejected if $\log 10(ABS(x)) > 37$ or if x contains more than 38 digits.

```
10 CLEAR 1000
20 INPUT "INPUT A NON-ZERO NUMBER"; S1$
30 IF LEN(S1$) > 38 THEN PRINT "TOO MANY
   DIGITS": GOTO 20
40 S2$ = "E": SP = 1
50 GOSUB 1000
60 IF SF > 0 THEN GOSUB 140
70 S2$ = "D": SP = 1
80 GOSUB 1000
90 IF SF > 0 THEN GOSUB 140
100 X = VAL(S1$)
110 PRINT "THE RECIPROCAL OF "; X; "IS  "; 1/X
120 GOTO 20
130 'SUBROUTINE TO CHECK FOR EXPONENTS > 37
140 EX = ABS(VAL(RIGHT$(S1$,S3-SF)))
150 CH = ABS(VAL(LEFT$(S1$, SF-1)))
     'CHARACTERISTIC
160 MC = LOG(CH)/LOG(10)   'LOG BASE 10 OF CH
170 TM = EX + MC   'TOTAL MAGNITUDE
180 IF TM < 38 THEN RETURN
190 PRINT "NUMBER OUT OF RANGE, VALUE SET = 1"
200 S1$ = "1"
210 RETURN
1000 'INSTRING SUBROUTINE
1010 SF = 0: S3 = LEN(S1$): S4 = LEN(S2$)
1020 IF S4 > S3 - SP + 1 THEN RETURN
1030 SL = S3 - S4 + 1
1040 FOR SI = SP TO SL
1050   IF MID$(S1$, SI, S4) = S2$ THEN SF = SI:
     SI = SL
1060 NEXT SI
1070 RETURN
```

D.I.Y. #28-2

Add these lines and changes:

```
125 PRINT "VALUE SET TO 1E38"
127 X = 1E38
130 RESUME NEXT
```

Chapter Checkpoint #28

1. b 2. False 3. c 4. b

Appendix B

Sample Programs

The programs in this section illustrate many of the techniques discussed in the main text. We hope you'll enjoy them. Studying how they work may help you to put your own ideas into practice.

Note: All of the programs in this section are computer printouts of functioning programs. If they don't work, it's probably due to a typing error. Compare what you've typed with what's in the program listing.

Program #1. The Count

Who says TRS-80 can handle a maximum of only 16 significant digits? And who says TRS-80 will overflow if a number exceeds 10^{38}? (We did, in the chapter on numbers.)

In the following program, we show how to make TRS-80 keep track of up to 255 significant digits, and handle numbers as large as $10^{256}-1$!

It's all done with . . . strings.

Type in the program and run it. It'll ask you where you want to start counting. Start with a very large whole number, say, 49 nines followed by a zero:

```
99999999999999999999999999999999999999999999999990
```

Normally, such a number would overflow instantly. But watch The Count count! There is a pause when the computer hits all-nines.

Note: Because you can input a maximum of 240 characters, you won't be able to input 255 nines, which is the number $10^{256}-1$. The largest number you can input is 240 nines, which is the number $10^{240}-1$. But the program will count up to $10^{256}-1$ (be prepared for a very long wait!).

To see the program reach its ultimate overflow quickly, make the following changes:

```
30 A$=STRING$(254, "9") + "Ø"
35 PRINT "I'LL START COUNTING AT "
```

and run it.

Here's the program:

```
10 CLS
20 CLEAR 11ØØ
30 PRINT "WHERE DO YOU WANT TO START COUNTING?"
35 INPUT A$
4Ø P=LEN(A$)
5Ø PRINT:PRINT A$
6Ø C=VAL(MID$(A$,P,1))+1
7Ø MS$=A$: MR$=RIGHT$(STR$(C),1): PS=P: GOSUB 2ØØ: A$=MS$
8Ø IF C<1Ø THEN 4Ø
9Ø P=P-1
1ØØ IF P=Ø  THEN IF LEN(A$)=255 THEN PRINT "OVERFLOW":END:
    ELSE A$="1"+A$: GOTO 4Ø
11Ø GOTO 6Ø
2ØØ LS=LEN(MS$)
21Ø IF LS <> LEN(MR$) + LS - 1 OR PS < 1 THEN ERROR 5
22Ø MS$=LEFT$(MS$,PS-1) + MR$ + RIGHT$(MS$,LS-PS)
23Ø RETURN
```

Program #2. Diced Graphics

This program makes extensive use of graphics and cursor control characters to accomplish a form of animation (rolling dice).

Here's the program:

```
100 '   DICE WITH GRAPHICS
110 '
120 '  CLEAR SCREEN
130 CLS
140 '
150 '   RESERVE SPACE FOR STRING CHARACTERS
160 CLEAR 1000
170 '
180 '   GRAPHICS DATA:
190 '   OUTLINE OF THE DIE
200 DATA  160, 176, 176, 176, 144, 26, 24, 149, 24, 24,
    24, 24, 24, 170, 26, 24, 170, 25, 25, 25, 149
210 DATA  26, 24, 149, 24, 24, 24, 24, 24, 170, 26, 24,
    130, 131,131, 131, 129
220 DATA -1
230 '
240 '   BLANK INSIDE OF DIE. ('-1' MEANS END)
250 DATA  32, 32, 32, 26, 8, 8, 8, 26, 32, 32, 32, -1
260 '
270 '   FACES OF THE DIE  ('-1' MEANS END)
280 '   ONE
290 DATA 26, 25, 42, -1
300 '   TWO
310 DATA 25, 25, 42, 26, 26, 24, 24, 24, 42, -1
320 '   THREE
330 DATA  26, 42, 42, 42, -1
340 '   FOUR
350 DATA  42, 25, 42, 26, 24, 24, 24, 26, 42, 25, 42, -1
360 '   FIVE
370 DATA 42, 25, 42, 26, 24, 24, 42, 26, 42, 24, 24, 24, 42, -1
380 '   SIX
390 DATA 42, 42, 42, 26, 26, 24, 24, 24, 42, 42, 42, -1
400 DEFINT A-Z     'INTEGER ALL VARIABLES
410 DIM DF$(6)  'DF$() WILL HOLD THE SIX DIE FACES
420 '
430 'READ OUTLINE INTO OT$
440 READ IC: IF IC = -1 THEN 480
450 OT$ = OT$ + CHR$(IC): GOTO 440
460 '
470 '   READ BLANKING FIELD INTO BL$
480 READ IC: IF IC = -1 THEN 520
490 BL$ = BL$ + CHR$(IC): GOTO 480
500 '
510 '   READ DIE FACES INTO DF$()
520 FOR I = 1 TO 6
530     READ IC: IF IC = -1 THEN 550
540     DF$(I) = DF$(I) + CHR$(IC): GOTO 530
550 NEXT I
560 '
570 '   INITIALIZE DIE POSITIONS
580 F1 = 404: D1 = F1 + 65: F2 = 424: D2 = F2 + 65
590 '
600 '   MAIN PROGRAM
610 '
620 '   PRINT OUTLINES
630 PRINT @ F1, OT$;: PRINT @ F2, OT$;
640 '
650 '   ROLL THE DICE
660 FOR I = 1 TO RND(11) + 9    'ROLL 10 TO 20 TIMES
670     GOSUB 760   : V1 = IR 'GET 1ST RND VALUE AND SAVE IT
680     PRINT @ D1, BL$;: PRINT @ D1, DF$(V1);
690     GOSUB 760   : V2 = IR 'GET 2ND RND VALUE AND SAVE IT
700     PRINT @ D2, BL$;: PRINT @ D2, DF$(V2);
705    FOR DL = 1 TO 66: NEXT
710 NEXT I
720 PRINT @ 788, "TOTAL IS"; V1 + V2
730 INPUT "PRESS <ENTER> FOR NEXT ROLL"; X: CLS
740 GOTO 630
750 END
760 '      RANDOM NUMBER GENERATOR
770 '  ON EXIT, IR = RANDOM VALUE FROM 1 TO 6
780 IR = RND(6): RETURN
```

Program #3. Homing Missile

This program simulates the homing missile situation. One object, "XY", moves in a straight line and bounces off the wall the way a real object bounces, i.e., the angle of deflection equals the angle of inflection. The other object, "AB", always homes in on its target, XY.

The result of this chase can be interesting. As you'll find out, AB eventually catches XY even if XY is given a significant speed advantage.

Note: To change the object speeds, change line 6. VXY is the speed of XY, VAB is the speed of AB. Notice we've given XY a 5:3 speed advantage over AB.

The paths of XY and AB are not erased as the objects move. For an interesting effect, change the program so that the trail of AB is not erased. Simply make line 594 a remark by inserting a single apostrophe at the beginning of the line.

Here's the program:

```
0 'SET UP BOUNDARIES
4 CLS
5 CLEAR 192
6 VXY = 5: VAB = 3
10 PRINT CHR$(151) + STRING$(62,131) + CHR$(171);
20 FOR I = 1 TO 14
30 PRINT CHR$(149) + STRING$(62,32) + CHR$(170);
40 NEXT I
50 PRINT CHR$(181) + STRING$(62,176);
60 POKE 16383,186
100 ' MAIN PROGRAM
110 GOSUB 200  'INITIALIZE XY OBJECT
120 GOSUB 300  'INITIALIZE AB OBJECT
130 GOSUB 400 'MOVE XY OBJECT
140 GOSUB 500 'MOVE AB OBJECT
150 GOTO 130 'KEEP MOVING THEM
200 'INITIALIZE XY OBJECT
210 X = RND(126): Y = RND(46)
220 IF POINT(X,Y) THEN 210
230 XM = RND(VXY * 5) / 5 * (-1 ↑ RND(2))
240 YM = SQR(VXY ↑ 2 - XM ↑ 2) * (-1 ↑ RND(2))
250 SET(X,Y)
299 RETURN
300 'INITIALIZE AB OBJECT
310 A = RND(126): B = RND(46)
320 IF POINT(A,B) THEN 310
330 SET(A,B)
399 RETURN
400 'MOVE XY OBJECT
410 XT = X: YT = Y
420 X = X + XM: Y = Y + YM
430 TX = X: TY = Y: T1 = XM: T2 = YM
440 GOSUB 1000 'CHECK POSITION, BOUNCE IF NECESSARY
450 X = TX: Y = TY: XM = T1: YM = T2
460 RESET(XT, YT)
470 SET(X,Y)
499 RETURN
500 'MOVE AB OBJECT
510 AD = SGN(X-A): BD = SGN(Y-B)
520 IF Y = B THEN AM = VAB * AD: BM = 0: GOTO 560
530 K = ABS((X-A)/(Y-B))
540 AM = VAB * K / SQR(K ↑ 2 + 1) * AD
550 BM = VAB / SQR(K ↑ 2 + 1) * BD
560 AT = A: BT = B
570 A = A + AM: B = B + BM
580 TX = A: TY = B: T1 = AM: T2 = BM
590 GOSUB 1000 'CHECK POS., BOUNCE IF NECESSARY
592 A = TX: B = TY: AM = T1: BM = T2
594 RESET(AT, BT)
596 SET(A,B)
599 RETURN
1000 'CHECK POSITION AND BOUNCE IF NECESSARY
1010 IF TX > 126 THEN TX = 126: T1 = -T1
1020 IF TX < 1 THEN TX = 1: T1 = -T1
1030 IF TY > 46 THEN TY = 46: T2 = -T2
1040 IF TY < 1 THEN TY = 1: T2 = -T2
1099 RETURN
```

Program #4. Drawing Board

This program requires at least 16K of RAM. It turns your Video Display into a graphics drawing board. The program will even let you save your pictures on tape, and load them in later.

The program is long, but we think you'll find it worth the trouble to type in. It contains some functions you'll probably find useful in your own programs. For example, lines 190-350 "redefine" certain keys on the keyboard. Lines 1040-1220 check the keyboard and take appropriate actions depending on the redefinitions.

There are two versions of the program, one for Model I, one for Model III. The only differences are in lines 100-580. Lines 1000-end are identical for both Computers, so we've printed these lines only once.

Here are the instructions for using the program:

The cursor starts out moving one space each time you press a motion key. However, you can select an auto-draw mode, in which the cursor moves continually.

The cursor starts out in the center of the screen. Normally, it draws as it moves. This is called the SET writemode. However, you can select the RESET writemode, in which case the cursor erases as it moves.

There are also two cursormodes, WRITE and POSITION ONLY. With the cursormode = WRITE, the cursor changes (sets or resets points) as you move it. With the cursor mode = POSITION ONLY, the cursor has no effect on the points; it simply moves around.

Keyboard Controls

Direction of Motion

Key	Action
↑	Cursor up.
↓	Cursor down.
←	Cursor left.
→	Cursor right.
Q	Cursor northwest.
W	Cursor northeast.
A	Cursor southwest.
S	Cursor southeast.

Modes

Key	Action
SHIFT W	Switch writemodes (SET/RESET).
SHIFT C	Switch cursormodes (SET/POSITION ONLY).
SHIFT A	Switch manual/auto draw.

Special commands

Key	Action
SHIFT R	Read the display into an array (takes about one minute).
SHIFT S	Save the display into an array (takes about one minute) and then write it onto tape. Be sure to have recorder ready before issuing this command. The process of writing to tape takes several minutes.
SHIFT L	Load a display from tape (takes several minutes). Rewind tape and get recorder ready first.
SHIFT P	Print the array contents onto the screen.
CLEAR	Erase the display.

Here's the program:

Lines 100-580 for Model I

```
100 '    DRAWING BOARD    MODEL I VERSION
110 '
120 '*** INITIALIZATION - COLD START *************
130 CLEAR 2050                          'LOTS OF SPACE TO STORE DSPLY
140 DEFINT A-Z
150 DIM V$(15)                          'ARRAY TO STORE DISPLAY
160 T = 1: F = -1: Z = 0                 'TRUE/FALSE/ZERO
170 VM = 15360                          'START OF VIDEO MEMORY
180 '
190 '    KEY-DEFINITIONS
200 UP$ = CHR$(91)                        'UP ARROW
210 DN$ = CHR$(10)                        'DOWN ARROW
220 LF$ = CHR$(8)                         'LEFT ARROW
230 RT$ = CHR$(9)                         'RIGHT ARROW
240 NE$ = "Q"                             'NORTH-EAST
250 NW$ = "W"                             'NORTH-WEST
260 SE$ = "A"                             'SOUTH-EAST
270 SW$ = "S"                             'SOUTH-WEST
280 WR$ = CHR$(ASC("W") + 32)             'SHIFT-W
290 AD$ = CHR$(ASC("A") + 32)             'SHIFT-A
300 CP$ = CHR$(ASC("C") + 32)             'SHIFT-C
310 CL$ = CHR$(31)                        'CLEAR
320 ST$ = CHR$(ASC("S") + 32)             'SHIFT-S
330 LT$ = CHR$(ASC("L") + 32)             'SHIFT-L
340 PR$ = CHR$(ASC("P") + 32)             'SHIFT-P
350 RD$ = CHR$(ASC("R") + 32)             'SHIFT-R
360 XH = 127: XL = 0                      'LIMITS ON X-COORDINATE
370 YH =  47: YL = 3                      'LIMITS ON Y-COORDINATE
380 '   READOUT FORMAT
390 FM$ = "WRITEMODE = %   %  CURSORMODE = %             %" + CHR$(30)
395 WN$ = "ARRAY EMPTY - USE <SHIFT-S> FIRST"
397 QU$ = CHR$(34)                       'DOUBLE QUOTES FOR TAPE OUTPUT
400 D = 1                                 'INCREMENT
410 CLS
420 '
500 '*** INITIALIZATION - WARM START **********
510 PX = (XH - XL) / 2 + 1                'START AT CENTER OF WINDOW
520 PY = (YH - YL) / 2 + 1
530 SB = POINT(PX, PY)                    'SAVE BLOCK-STATUS
540 W = T                                 'WRITE = ON
550 C = F                                 'CURSOR-MOTION-ONLY = OFF
560 A = F                                 'AUTO-DRAW = OFF
570 DX = 0: DY = 0                        'ZERO X AND Y VECTORS
580 '
```

Lines 100-580 for Model III

```
100 '   DRAWING BOARD   MODEL III VERSION
110 '
120 '*** INITIALIZATION - COLD START *************
130 CLEAR 2050                   'LOTS OF SPACE TO STORE DSPLY
140 DEFINT A-Z
150 DIM V$(15)                   'ARRAY TO STORE DISPLAY
160 T = 1: F = -1: Z = 0          'TRUE/FALSE/ZERO
170 VM = 15360                   'START OF VIDEO MEMORY
180 POKE 16409,0    'SET MODEL III KEYBOARD IN ULC MODE
190 '   KEY-DEFINITIONS
200 UP$ = CHR$(91)                'UP ARROW
210 DN$ = CHR$(10)                'DOWN ARROW
220 LF$ = CHR$(8)                 'LEFT ARROW
230 RT$ = CHR$(9)                 'RIGHT ARROW
240 NE$ = "q"                     'NORTH-EAST
250 NW$ = "w"                     'NORTH-WEST
260 SE$ = "a"                     'SOUTH-EAST
270 SW$ = "s"                     'SOUTH-WEST
280 WR$ = "W"
290 AD$ = "A"
300 CP$ = "C"
310 CL$ = CHR$(31)                'CLEAR
320 ST$ ="S"
330 LT$ = "L"
340 PR$ = "P"
350 RD$ = "R"
360 XH = 127: XL = 0              'LIMITS ON X-COORDINATE
370 YH =  47: YL = 3              'LIMITS ON Y-COORDINATE
380 '   READOUT FORMAT
390 FM$ = "WRITEMODE = %    %  CURSORMODE = %              %" + CHR$(30)
395 WN$ = "ARRAY EMPTY - USE <SHIFT-S> FIRST"
397 QU$ = CHR$(34)               'DOUBLE QUOTES FOR TAPE OUTPUT
400 D = 1                         'INCREMENT
410 CLS
420 '
500 '*** INITIALIZATION - WARM START **********
510 PX = (XH - XL) / 2 + 1        'START AT CENTER OF WINDOW
520 PY = (YH - YL) / 2 + 1
530 SB = POINT(PX, PY)            'SAVE BLOCK-STATUS
540 W = T                         'WRITE = ON
550 C = F                         'CURSOR-MOTION-ONLY = OFF
560 A = F                         'AUTO-DRAW = OFF
570 DX = 0: DY = 0                'ZERO X AND Y VECTORS
580 '
```

Lines 1000-end for Models I-III

```
1000 '*** START MAIN PROGRAM **********
1010 K$ = INKEY$            'FLUSH PRIOR ENTRIES
1020 GOSUB 7000             'DISPLAY SWITCH STATUS
1030 GOSUB 6010             'BLINK BLOCK
1040 '   KEYBOARD ROUTINE
1050 K$ = INKEY$: IF K$ = "" THEN 1240  'IF NONE, CHK AUTODRAW
1060 IF K$ = UP$ THEN DX = Z: DY =-D: GOTO 1270 'DRAW
1070 IF K$ = DN$ THEN DX = Z: DY = D: GOTO 1270
1080 IF K$ = LF$ THEN DX =-D: DY = Z: GOTO 1270
1090 IF K$ = RT$ THEN DX = D: DY = Z: GOTO 1270
1100 IF K$ = NE$ THEN DX = D: DY =-D: GOTO 1270
1110 IF K$ = NW$ THEN DX =-D: DY =-D: GOTO 1270
1120 IF K$ = SE$ THEN DX = D: DY = D: GOTO 1270
1130 IF K$ = SW$ THEN DX =-D: DY = D: GOTO 1270
1140 IF K$ = WR$ THEN W =-W: GOTO 1240   'CHK AUTODRAW
1150 IF K$ = AD$ THEN A =-A: GOTO 1240
1160 IF K$ = CP$ THEN C =-C: GOTO 1240
1170 IF K$ = RD$ THEN GOSUB 2010         'READ DSPLY INTO ARRAY
1180 IF K$ = PR$ THEN 5000               'PRINT ARRAY TO DSPLY
1190 IF K$ = ST$ THEN GOSUB 2010 : GOTO 3010 'SAVE DSPLY ON TAPE
1200 IF K$ = LT$ THEN 4010              'LOAD DSPLY FROM TAPE
1210 IF K$ = CL$ THEN CLS: GOTO 500     'CLEAR SCREEN & RESTART
1220 GOTO 1030            'UNDEFINED KEY - IGNORE AND GET ANOTHER
1230 '   CHECK AUTO-DRAW
1240 GOSUB 7000                  'UPDATE SWITCH READOUT
1250 IF A = F THEN 1030          'NO AUTO-DRAW, SO GET NEW K$
1260 '   DRAW ROUTINE
1270 IF C = T THEN 1310          'RESTORE BLOCK, MOVE CURSOR
1280 IF W = T THEN SET(PX,PY): GOTO 1340 ''SET BLOCK & MV CSR
1290 RESET(PX,PY): GOTO 1340     'ERASE BLOCK & MV CSR
1300 '   RESTORE BLOCK
1310 IF SB THEN SET(PX,PY): GOTO 1340   'MV CSR
1320 RESET(PX,PY)
1330 '   MOVE CURSOR
1340 PX = PX + DX: PY = PY + DY         'INCREMENT X AND Y
1350 IF PX < XL THEN PX = XH: GOTO 1370  'CHECK PX AND PY
1360 IF PX > XH THEN PX = XL
1370 IF PY < YL THEN PY = YH: GOTO 1390
1380 IF PY > YH THEN PY = YL
1390 SB = POINT(PX,PY)           'SAVE BLOCK-STATUS
1400 SET(PX,PY)                  'SET NEW POSITION
1410 GOTO 1030                   'GET NEW K$
1420 '
2000 '*** SAVE DISPLAY INTO ARRAY *********
2010 A = F: RESET(PX,PY)          'STOP AUTODRAW & BLOCK OFF
2020 PRINT @ 0, "READING DISPLAY INTO ARRAY..."; CHR$(30);
2025 GOSUB 2200              'EMPTY EXISTING ARRAY
2030 FOR LI = 0 TO 15
2040    ML = LI * 64 + VM         'MEM ADDR OF CURRENT LINE
2050    FOR CL = 0 TO 63
2060      AD = ML + CL
2070      V$(LI) = V$(LI) + CHR$(PEEK(AD))
2080 NEXT CL, LI
2090 RETURN
2100 '
2200 'EMPTY THE ARRAY
2210 FOR I = 0 TO 15: V$(I) = "": NEXT: RETURN
3000 '*** SAVE ARRAY TO TAPE **********
3010 PRINT @ 0, "SAVING DISPLAY ONTO TAPE..."; CHR$(30);
3020 FOR LI = 0 TO 15
3030    PRINT #-1, QU$; V$(LI); QU$
3040 NEXT LI
3050 GOTO 1010                   'GET NEW K$
3060 '
4000 '*** READ DISPLAY FROM TAPE
4010 PRINT @ 0, "READING DISPLAY FROM TAPE..."; CHR$(30);
4020 FOR LI = 0 TO 15
4030    INPUT#-1, V$(LI)
4040 NEXT LI
4050 '
5000 '*** WRITE ARRAY TO DISPLAY **********
5010 CLS: ON ERROR GOTO 9000   'PREVENT FC ERROR
5020 FOR LI = 0 TO 14
5030    PRINT V$(LI);
5040 NEXT LI
5050 PRINT LEFT$(V$(LI), 63);
5060 POKE 16383, ASC(RIGHT$(V$(LI),1))   'PREVENT SCROLL
5070 GOTO 500                            'WARM START
5080 '
6000 '*** BLINK BLOCK **********
```

```
6010 IF POINT(PX,PY) THEN RESET(PX,PY): RETURN
6020 SET(PX,PY): RETURN
7000 '*** DISPLAY SWITCH STATUS **********
7010 IF W = 1 THEN SM$ = "SET" ELSE SM$ = "RESET"
7020 IF C = 1 THEN SC$ = "POSITION ONLY" ELSE SC$ = "WRITE"
7030 PRINT @ 0, USING FM$; SM$, SC$;
7040 RETURN
9000 'ERROR HANDLER
9010 IF ERR/2 + 1 <> 5 THEN ON ERROR GOTO 0
9020 PRINT @ 0, WN$; CHR$(30);
9030 FOR DL = 1 TO 500: NEXT DL
9040 RESUME 1010
```

Program #5. Automatic Censor, Broadaxe Version

Here is the complete listing of the program developed in Chapter 19.

```
10 CLEAR 1000
15 DEFINT A-Z
20 DIM K$(255)                  'K$() STORES DELIMITER IMAGE
25 DIM K(256)                      'K() STORES DELIMITER POSITONS
30 CLS
40 PRINT "MEKLOVAKIAN OFFICIAL AUTOMATIC CENSOR."
45 PRINT "BROADAXE VERSION--YOU TYPE IT IN--I CUT IT UP"
50 PRINT "TYPE A QUOTATION MARK, THEN THE TEXT."
60 INPUT P$: S1$ = P$
70 FOR I=1 TO LEN(S1$)          'CHANGE LOWERCASE TO UPPERCASE
75      S2$ = MID$(S1$, I, 1)
80      IF S2$ < CHR$(97) OR S2$ > CHR$(122) THEN 95
85      S2$ = CHR$( ASC(S2$) - 32)
90      SF = I: SC = 1: GOSUB 2000
95 NEXT I
100 FOR I=1 TO 255: K$(I) = "": NEXT I      'CLEAR OUT K$( )
110 DATA " " , "," , ";" , "." , "!" , "?" , "-", "0"
115 READ S2$: IF S2$ = "0" THEN 190  'CHECK ALL DELIMS.
120 SP = 1
125 GOSUB 1000
130 IF SF = 0 THEN 115  'GET NEXT DELIMITER
135 K$(SF) = S2$
140 SP = SF + S4: IF SP > S3 THEN 115    'END OF TEXT
145 GOTO 125
190 KT = 0: K(0) = 0   'KT COUNTS BLANKS, K(0) IS ALWAYS ZERO
192 FOR I=1 TO 255
194   IF K$(I) <> "" THEN KT = KT + 1: K(KT) = I
196 NEXT I
200 K(KT + 1) = S3 + 1      'END OF TEXT MARKER
210 FOR I = 1 TO KT + 1
220     IF K(I) - K(I-1) <> 5 THEN 290
230     S2$ = "****": SC = 4: SF = K(I - 1) +1
240     GOSUB 2000
290 NEXT I
300 IF S3 < 8 THEN 400
305 SP=1
310 S2$ = "SCRUBDEK": GOSUB 1000
320 IF SF = 0 THEN 400
330 S2$ = "NOBODY": SC = 8: GOSUB 2000
340 IF SR < S4 THEN 400
350 SP = SF + 1: GOTO 310
400 CLS: PRINT "AUTHORIZED TEXT FOLLOWS...": PRINT S1$
420 END
1000 'INSTRING SUBROUTINE
1080 SF = 0: S3 = LEN(S1$): S4 = LEN(S2$)
1090 IF S4 > S3 - SP + 1 THEN RETURN
1100 SL = S3 - S4 + 1
1110 FOR SI = SP TO SL
1120   IF MID$(S1$, SI, S4) = S2$ THEN SF = SI: SI = SL
1130 NEXT SI
1140 RETURN
2000 'MIDSTRING REPLACEMENT SUBROUTINE
2070 S3 = LEN(S1$): SR = S3 - SF - SC + 1
2080 IF SF < 1 OR SR < 0 THEN RETURN
2090 S1$ = LEFT$(S1$, SF-1) + S2$ + RIGHT$(S1$,SR)
2100 RETURN
```

Appendix C

C / Error Codes

CODE	ABBREVIATION	ERROR
1	NF	NEXT without FOR
2	SN	Syntax error
3	RG	Return without GOSUB
4	OD	Out of data
5	FC	Illegal function call
6	OV	Overflow
7	OM	Out of memory
8	UL	Undefined line
9	BS	Subscript out of range
10	DD	Redimensioned array
11	/0	Division by zero
12	ID	Illegal direct
13	TM	Type mismatch
14	OS	Out of string space
15	LS	String too long
16	ST	String formula too complex
17	CN	Can't continue
18	NR	NO RESUME
19	RW	RESUME without error
20	UE	Unprintable error
21	MO	Missing operand
22	FD	Bad file data
23	L3	Disk BASIC only

Explanation of Error Messages

NF NEXT without FOR: NEXT is used without a matching FOR statement. This error may also occur if NEXT *variable* statements are reversed in nested loop.

SN Syntax Error: This usually is the result of incorrect punctuation, open parenthesis, an illegal character or a mis-spelled command.

RG RETURN without GOSUB: A RETURN statement was encountered before a matching GOSUB was executed.

OD Out of Data. A READ or INPUT # statement was executed with insufficient data available. DATA statement may have been left out or all data may have been read from tape or DATA.

FC Illegal Function Call: An attempt was made to execute an operation using an illegal parameter. Examples: square root of a negative argument, negative matrix dimension, negative or zero LOG arguments, etc. Or USR call without first POKEing the entry point.

OV Overflow: The magnitude of the number input or derived is too large for the Computer to handle. NOTE: There is no underflow error. Numbers smaller than ±1.701411E−38 single precision or ±1.701411834544556E−38 double precision are rounded to 0. See /0 below.

OM Out of Memory: All available memory has been used or reserved. This may occur with very large matrix dimensions, nested branches such as GOTO, GOSUB, and FOR-NEXT Loops.

UL Undefined Line: An attempt was made to refer or branch to a non-existent line.

BS Subscript out of Range: An attempt was made to assign a matrix element with a subscript beyond the DIMensioned range.

DD Redimensioned Array: An attempt was made to DIMension a matrix which had previously been dimensioned by DIM or by default statements. It is a good idea to put all dimension statements at the beginning of a program.

/0 Division by Zero: An attempt was made to use a value of zero in the denominator. **Note**: If you can't find an obvious division by zero check for division by numbers smaller than allowable ranges. See OV above.

ID Illegal Direct: The use of INPUT as a direct command.

TM Type Mismatch: An attempt was made to assign a non-string variable to a string or vice-versa.

OS Out of String Space: The amount of string space allocated was exceeded. Try to CLEAR more.

LS String Too Long: A string variable was assigned a string value which exceeded 255 characters in length.

ST String Formula Too Complex: A string operation was too complex to handle. Break up the operation into shorter steps.

CN Can't Continue: A CONT was issued at a point where no continuable program exists, e.g., after program was ENDed or EDITed.

NR No RESUME: End of program reached in error-trapping mode.

RW RESUME without ERROR: A RESUME was encountered before ON ERROR GOTO was executed.

UE Unprintable Error: An attempt was made to generate an error using an ERROR statement with an invalid code.

MO Missing Operand: An operation was attempted without providing one of the required operands.

FD Bad File Data: Data input from an external source (i.e., tape) was not correct or was in improper sequence, etc.

L3 DISK BASIC only: An attempt was made to use a statement, function or command which is available only with the Disk System.

Appendix D

TRSCII (TRS-80 Code for Information Interchange)

Code Dec.	Hex.	Key(s)*	PRINT CHR$(*code*)
1	00		No effect
1	01	BREAK SHIFT ↓ A	No effect
2	02	SHIFT ↓ B	No effect
3	03	SHIFT ↓ C	No effect
4	04	SHIFT ↓ D	No effect
5	05	SHIFT ↓ E	No effect
6	06	SHIFT ↓ F	No effect
7	07	SHIFT ↓ G	No effect
8	08	← SHIFT ↓ H	Backspace and erase
9	09	→ SHIFT ↓ I	No effect
10	0A	↓ SHIFT ↓ J	Move cursor to start of next line and erase line
11	0B	SHIFT ↓ K	No effect
12	0C	SHIFT ↓ L	No effect
13	0D	ENTER SHIFT ↓ M	Move cursor to start of next line and erase line
14	0E	SHIFT ↓ N	Cursor on
15	0F	SHIFT ↓ O	Cursor off
16	10	SHIFT ↓ P	No effect
17	11	SHIFT ↓ Q	No effect
18	12	SHIFT ↓ R	No effect
19	13	SHIFT ↓ S	No effect
20	14	SHIFT ↓ T	No effect
21	15	SHIFT ↓ U	Swap space compression/ special characters
22	16	SHIFT ↓ V	Swap special/alternate characters
23	17	SHIFT ↓ W	Double-size characters
24	18	SHIFT ← SHIFT ↓ X	Backspace without erasing
25	19	SHIFT ↓ Y	Advance cursor
26	1A	SHIFT ↓ Z	Move cursor down
27	1B	SHIFT ↑	Move cursor up
28	1C	SHIFT ↓ ,†	Move cursor to upper left corner
29	1D	SHIFT ↓ 9†	Move cursor to beginning of line
30	1E	SHIFT ↓ .†	No effect
31	1F	CLEAR SHIFT ↓ /†	No effect

* Many of these key-sequences are to be used with INKEY$, not INPUT.

In Model 4, the CTRL (Control) key works the same way as SHIFT ↓.

†Not available in Model I.

Code Dec.	Code Hex.	Key(s)	PRINT CHR$(*code*)
32	20	SPACEBAR	b̸
33	21	!	!
34	22	”	”
35	23	#	#
36	24	$	$
37	25	%	%
38	26	&	&
39	27	’	’
40	28	(	(
41	29	)	)
42	2A	*	*
43	2B	+	+
44	2C	,	,
45	2D	–	–
46	2E	.	.
47	2F	/	/
48	30	0	0
49	31	1	1
50	32	2	2

Code Dec.	Code Hex.	Key(s)	PRINT CHR$(*code*)
51	33	3	3
52	34	4	4
53	35	5	5
54	36	6	6
55	37	7	7
56	38	8	8
57	39	9	9
58	3A	:	:
59	3B	;	;
60	3C	<	<
61	3D	=	=
62	3E	>	>
63	3F	?	?
64	40	@	@
65	41	A	A
66	42	B	B
67	43	C	C
68	44	D	D

Code Dec.	Hex.	Key(s)	PRINT CHR$(*code*)
69	45	E	E
70	46	F	F
71	47	G	G
72	48	H	H
73	49	I	I
74	4A	J	J
75	4B	K	K
76	4C	L	L
77	4D	M	M
78	4E	N	N
79	4F	O	O
80	50	P	P
81	51	Q	Q
82	52	R	R
83	53	S	S
84	54	T	T
85	55	U	U
86	56	V	V
87	57	W	W
88	58	X	X
89	59	Y	Y
90	5A	Z	Z
91	5B	↑	[or ↑
92	5C		\ or ↓
93	5D		] or ←
94	5E		∧ or →
95	5F		–
96	60		@
97	61	A	a
98	62	B	b
99	63	C	c
100	64	D	d
101	65	E	e
102	66	F	f
103	67	G	g
104	68	H	h
105	69	I	i

Note: For all-capitals Model I's, codes 96-127 display the same characters as codes 64-95.

Code Dec.	Code Hex.	Key(s)	PRINT CHR$(*code*)
106	6A	J	j
107	6B	K	k
108	6C	L	l
109	6D	M	m
110	6E	N	n
111	6F	O	o
112	70	P	p
113	71	Q	q
114	72	R	r
115	73	S	s
116	74	T	t
117	75	U	u
118	76	V	v
119	77	W	w
120	78	X	x
121	79	Y	y
122	7A	Z	z
123	7B		{
124	7C		¦
125	7D		}
126	7E		~
127	7F		±
128	80	Codes 128-191 output graphics characters. See the graphic display table in this Appendix.	
192 255	C0 FF	Codes 192-255 output either space compression codes or special characters when used with PRINT CHR$(*code*). **Model III Only:** They always output special characters when used with POKE *vidram, code*.	

Model III BASIC Reserved Words*

@
ABS
AND
ASC
ATN
AUTO
CDBL
CHR$
CINT
CLEAR
CLOCK
CLOSE
CLS
CMD
CONT
COS
CSNG
CVD
CVI
CVS
DATA
DEFDBL
DEFFN
DEFINT
DEFSNG
DEFUSR
DEFSTR
DELETE
DIM
EDIT
ELSE
END
EOF
ERL
ERR
ERROR
EXP
FIELD
FIX
FN
FOR
FORMAT
FRE
FREE
GET
GOSUB
GOTO
IF
INKEY$
INP
INPUT
INSTR
INT
KILL
LEFT$
LET
LSET
LEN
LINE
LIST
LLIST
LPRINT
LOAD
LOC
LOF
LOG
MEM
MERGE
MID$
MKD$
MKI$
MKS$
NAME
NEW
NEXT
NOT
ON
OPEN
OR
OUT
PEEK
POINT
POKE
POS
POSN
PRINT
PUT
RANDOM
READ
REM
RENAME
RESET
RESTORE
RESUME
RETURN
RIGHT$
RND
RSET
RUN
SAVE
SET
SGN
SIN
SQR
STEP
STOP
STRING$
STR$
SYSTEM
TAB
TAN
THEN
TIME$
TO
TROFF
TRON
USING
USR
VAL
VARPTR
VERIFY

*Some of these words have no function in BASIC; they are reserved for use in Disk BASIC. None of these words can be used inside a variable name. You'll get a syntax error if you try to use these words as variables.

Graphics Characters (Codes 128-191)

128 129 130 131 132 133 134 135

136 137 138 139 140 141 142 143

144 145 146 147 148 149 150 151

152 153 154 155 156 157 158 159

160 161 162 163 164 165 166 167

168 169 170 171 172 173 174 175

176 177 178 179 180 181 182 183

184 185 186 187 188 189 190 191

Appendix E

Graphic Worksheet

TAB: 0, 5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60

PRINT AT

X

Y

Rows (PRINT AT): 0–47

Left scale: 0, 64, 128, 192, 256, 320, 384, 448, 512, 576, 640, 704, 768, 832, 896, 960

Right scale: 63, 127, 191, 255, 319, 383, 447, 511, 575, 639, 703, 767, 831, 895, 959, 1023

Index

! 231-2, 241
" 16, 186
$ 45, 227, 228, 230, 232-3
% 102-3, 223, 230, 232
' 70, 71, 273
() 21, 116
* 18
+ 197, 225-227
, 17, 53, 54, 186, 224-5, 229, 241, 286
– 225-227, 229-230
. 69
/ 16
: 64, 66, 186
; 17, 52, 53, 213, 214, 224, 286
< 59-60
> 59-60
? 63

AAGF, 100
AAW, 203
Abnormal programs, 37
ABS, 118
Absolute value, 114, 118
AND, 147, 151
Animation 257, 327
Ajax, 87
Aristotle, 190
Array, 163, 164, 289
  integer, 178, 181
  one-dimensional, 164
  two-dimensional, 168
  three-dimensional, 174-5
  single-precision, 96-106
ASC, 204, 253
ASCII, 203, 237, 250, 274, 286, 292
Asterisk (*), 18
AUTO, 66, 68

Babson, Bob, 161
Backspace, 8, 77, 83-4, 254
BREAK, 13, 244, 271, 303
Broadaxe, 202
Byte, 181, 290-1

Calculator, 153-5
Cassette, 211, 283-4
Chapter Checkpoint Answers 312-25
Chess, 256
CHR$, 203, 204, 237-8, 240, 251, 253
CLEAR, 176, 187, 289
CLEAR, 9, 238
CLS, 20, 238-9, 265, 292
Character, 249
  codes, 249, 250, 253, 272, 274
  control, 237, 240, 251
  graphics, 252, 275, 339
CINT, 118
Colon (:), 64, 66, 186
Columns, 53
Comma, 17, 53, 54, 186, 224-5, 229, 241, 286
  floating, 228, 230
  trailing, 54, 215-6
Command, 13, 15, 37
Concatenation, 97
Conditional branch, 56
Control codes, 251, 274
COS, 219
Cosine, 219
Counter, 125, 128
CSAVE, 219, 245
cummings, e.e., 205
Cursor movement 8, 77, 83-4, 215, 217, 237, 240-1, 249, 250-1

DATA, 98-9, 100, 125, 307
Decimal point, 224, 230
Declare-it statement (see Statements) 38, 119, 120
DEFDBL, 120
DEFINT, 119-20
DEFSNG, 120
DEFSTR, 120
DELETE, 29, 69-70, 77-8, 81
Delimiter, 195
Dewey Decimal System, 154
DIM, 164, 168, 180, 289
D-notation, 229
Dollar sign ($), 45, 227, 228, 230, 232-3
Do-It-Yourself #, 1-1, -4, 10; 2-1, -5, 18; 3-1, 31; 3-2, 32; 4-1, 47; 5-1, 56; 5-2, 57; 5-3, 60; 6-1, 66; 7-1, 81; 9-1, 98; 9-2, 103; 9-3, 108; 9-4, 110; 10-1, 113; 10-2, 115; 10-3, 117; 10-4, -5, 118; 10-6, 120; 11-1 -2, 125; 11-3 -4, 126; 11-6, -7, 128; 11-8, 130; 11-9 -10, 131; 11-11, 132; 12-1 -2, -3, 142; 13-1, 148; 13-2, 149; 14-1, 155; 15-1, 165; 15-2, 166; 15-3, 167; 15-4, 170; 15-5, 171; 16-1 -2, 174; 16-3 -4, 176; 16-5, 177; 16-6, 178; 17-1, 189; 17-2, 192; 18-1, 198; 19-1, 203; 19-2 -3, 207; 19-4, 208; 20-1, 213; 20-2, 216; 21-1, 223; 21-2, 229; 21-3, 233; 22-1 -5, 239; 23-1, 257; 25-1, 273; 25-2, 280; 26-1, 285; 27-1, 294; 27-2, 295; 28-1, 300; 28-2, 302
Do-It-Yourself Answers, 312-25
Double-precision, 107-8, 178, 181, 214, 229, 230
Double-size mode, 10

EDIT, 30, 69, 70, 75-93
END, 303
ENTER 13, 14, 91, 92
E-notation, 229
Erase, 8, 30
ERL, 304
Error-handling, 157-8, 211, 299-309
Error messages 14, 20, 332-3
Exclamation point (!), 231-2, 244
EXECUTE, 26
Exponential (E) format, 97-8
Exponentiation, 114
Expressions, 42
Field specifiers (see PRINT USING), 221-36
FIX, 118
FOR/NEXT, 128, 129-33, 255, 280
Googol, 19
GOSUB, 140-1
GOTO, 51, 54, 133, 137, 303
Graphics (also see TRSCII and Statements), 38, 211, 219, 249-51, 254, 261, 265, 271, 275, 328
Greater Than (>), 59-60

Hierarchy, 117
Hyphen (-), 201

IF. . . END, 57
IF. . . THEN, 55, 56, 59, 133, 151, 233, 268
IF. . . THEN. . . ELSE, 135-6
Increment, 125, 126, 128
INKEY$, 271, 280, 295
Initial value, 125, 128
INP, 296
INPUT, 58, 215, 274, 283-5
Input/Output (I/O) (see Statements), 38, 45-6, 283
INT, 118
Integer variable, 103
Integer array, 178, 181
Keyboard, 7

LEFT$, 190
LEN, 188
Level II BASIC Reference Manual, 3
Level II Programming Techniques, 289
Less than (<), 59-60
Limit value, 125, 128
Lincoln, Abraham, 249
Line, 69
Line number, 28, 66-8
Line printer, 211, 249
LIST, 27, 77
LLIST, 286
LOG, 118
Logarithm, 154-9
Logical expression, 219
Loops, 52, 123-33
Lower/uppercase, 250, 275
LPRINT, 286

Marginalia, 4
Meklovakia, 185-92
Memory, 181-2, 211
Menu, 153-5
MID$, 190
Minus sign (–), 225-7, 229-30
Model I, 3, 76, 201-2, 204, 217, 228, 251, 283, 295, 329
Model III, 3, 76, 114, 229, 251, 272, 275, 329
Model III Operation and BASIC Language Reference Manual, 3
Multiple Branches, 137-8
Multi-statement lines, 64-6

Nth Root, 114
Name-it statement (see Statements), 37-9
Negative numbers, 98, 103, 108
NEW, 30, 255
Nobody, 197
NOT, 150-1
Numeric variables, 43, 96, 107
Numbers, tables, 215, 221
  double-precision, 214
  single-precision, 96-106, 178, 181, 214
  whole-numbers, 223
  positive, 225
  negative, 225
ON ERROR GOTO, 301
ON ERR GOTO, 307
ON. . . GOSUB, 142

ON. . . GOTO, 138
Operators, arithmetic, 117
  exponentiation, 116
  mathematical, 116-7, 218
  relational, 60
OR, 150-1
OUT, 296
Overflow, 19

Parentheses, 21, 116
Pause, 54
PEEK, 289, 290, 293-4
Percent sign (%), 102-3, 223, 230, 232
Period (.), 69
Plus sign (+), 197, 225-7
POINT, 262-7
POKE, 289, 291, 294
Pope, Alexander, 87
Power, 154-9
PRINT, 16, 18, 53, 64, 237-8, 241, 249, 274
  @, 241, 264, 267, 292
  advanced, 211, 213, 215
  cell, 262
  TAB, 217, 218, 223, 286
  tables, 216
Print zones, 17, 54
Program, 25
Program Flow, 38
Prompt, 7, 58

Question mark (?), 63
Quotation marks (" "), 16, 186

Radians/degrees, 154
RAM, 290-4, 300
Random, 167
READ, 98-100, 125
READY, 7
REM, 70-1, 273
RESET, 262, 265
Reserved words, 337
Resolution, 261, 265
RESUME, 301
RETURN, 140-1
Re-usable instructions, 25
RIGHT$, 190
RND, 167, 215-6
ROM, 186, 187, 290
Round-off, 103
Routines, 138
RUN, 26

Sample programs, 326-31
SAVE, 33
Scientific notation (see E-notation), 229
Scrolling, 21
Scrubdek, 187
Semi-colon, 17, 52, 224, 286
  trailing, 53, 213, 214, 224
SET, 262, 265, 329
SGN, 118
(SHIFT), 329
  @, 54, 245, 273, 274
  < >, 10
Significant digits, 97
SIN, 118, 219, 261
Sine, 154-9, 219, 242, 245, 261, 264
(SPACEBAR), 76, 77, 84
Space compression, 251
Spaces, 13, 46-7
SQR, 118
Square root, 118
Statements, 37-9, 45-6, 119, 120
STRING$, 253-4
Strings, 43, 44, 185
  arrays, 175-8, 181-2
  comparison, 206
  expressions, 197
  format, 227, 231
  null, 241, 273
  variables, 43-5
  vs. numbers, 241, 217, 254
Subroutines, 138-42
Subscript, 163, 178, 179
Syntax, 19

(TAB), 9
Tape, 217
Test action, 56, 135
  true/false, 151
Text, 185, 250
TIMER, 127
Trigonometry, 219, 245
TROFF, 123-4
TRON, 123-4
TRS-80 Assembly Language Programming, 289
TRS-80 Reference Manual, 219
TRSCII, 250, 253, 262, 292, 334-6
Type-declaration tags, 119-20

UFO (defined), 9
UFO#, 1-1, 9; 1-2, 10; 2-1 -2, 14; 2-3, 16; 3-1, 26; 3-2, 27; 3-3, 28; 3-4, 30; 4-1, 39; 4-2, 40; 4-3, 42; 4-4, 43; 4-5, 45; 5-1, 53; 5-2, -3, 54; 5-4, 55; 5-5, 56; 5-6, 58; 5-7, 60; 6-1, 64; 6-2, 66; 6-3, 68; 6-4, 69; 6-5, 70; 6-6, 71; 7-1, 76; 7-2, 77; 7-3, 81; 8-1, 84; 8-2, 85; 8-3, 86; 8-4, 88; 8-5, 90; 8-6, 91; 8-7, 92; 9-1, 96; 9-2, 98; 9-3, 100; 9-4, 103; 9-5, 107; 9-6, 108; 9-7, 110; 10-1, 114; 10-2, 117; 10-3, 120; 11-1, 124; 11-2, 128; 11-3, 133; 12-1, 136; 12-2, 138; 12-3, 141; 13-1, 151; 14-1, 156; 15-1, 164; 15-2, 168; 16-1, 175; 16-2, 179; 17-1, 186; 17-2, 187; 17-3, 188; 17-4, 190; 18-1, 197; 19-1, 204; 20-1, 217; 21-1, 222; 21-2, 223; 21-3, 225; 21-4, 226; 21-5, 230; 21-6, 233; 22-1, 238; 22-2, 241; 22-3, 254; 24-1, 265; 24-2, 266; 24-3, 267; 25-1, 273; 25-2, 274; 27-1, 292; 28-1, 300; 28-2, 301; 22-3, 305; 28-4, 306; 28-5, 308
Unconditional branch, 55
Up-arrow (↑), 229-30
Upper/lowercase, 250, 275
USR, 296

VAL, 276, 278
Variables, 39, 40, 101, 103, 163-4, 168, 224
VARPTR, 296
Video display, 9

Woodpecker method, 8
Wrap-around, 64, 217

X-Y axis, 244-5, 261

Zeros, 214

RADIO SHACK, A DIVISION OF TANDY CORPORATION

U.S.A.: FORT WORTH, TEXAS 76102
CANADA: BARRIE, ONTARIO L4M 4W5

TANDY CORPORATION

AUSTRALIA	BELGIUM	U. K.
91 KURRAJONG ROAD MOUNT DRUITT, N.S.W. 2770	PARC INDUSTRIEL DE NANINNE 5140 NANINNE	BILSTON ROAD WEDNESBURY WEST MIDLANDS WS10 7JN

8749414 283 S-L

PRINTED IN U.S.A.